# Trust the clones

Sophia Drossopoulou[1] and James Noble[2]

[1] Department of Computing,
Imperial College, London
`scd@doc.ic.ac.uk`
[2] School of Mathematics, Statistics, and Computer Science,
Victoria University of Wellington,
Wellington, New Zealand
`kjx@mcs.vuw.ac.nz`

**Abstract.** Most object-oriented programming languages provide some way to clone objects — to produce a new object that is a copy of an existing object. Typically this is either a shallow clone that clones only one object, or a deep clone that clones objects recursively. In practice, programmers have to write custom cloning methods for each of their classes, and this boilerplate code is tedious to write and requires care to write correctly. Inspired by ownership types, we propose a system that annotates classes to express how their instances should be cloned. Furthermore, we show how these annotations can be used to generate cloning methods for these classes.

## 1 Introduction

*Background* Ownership types were first suggested in 1998; their aim was to characterise aliases, and thus to control the topology of objects on the heap [19]. Since then, they have attracted tremendous interest; they have been developed in several brands and variations, and they have been put to several different uses, e.g. memory management, encapsulation, effect systems, avoidance of race conditions, locations, parallel programming, program verification etc.

One example given in the seminal ECOOP paper [19], was cloning, whereby all the objects “inside” a certain other object would automatically be copied when the enclosing object was copied.

The aim of supporting cloning, although appealing, has — to our knowledge — not been further pursued in the context of static ownership types. Interestingly, this aim was recently pursued in the static analysis world [14], whereby copy policies are expressed by annotating code so as to specify the maximally allowed sharing between an object and its clone, and a type and effects system checks whether the copy policy is adhered to.

In this paper we propose the opposite approach: the copy code is *generated* out of *clone annotations* given with the types of fields, and thus, adherence to the copy policy is automatically guaranteed.

*Overview of our approach* The key problem we aim to address is determining which objects to copy. Generally, the set of objects which have to be cloned when cloning an object $o$ does not solely depend on $o$ — it also depends on the object which started the the cloning process — the "originator" of the cloning.

Consider the object structure shown in Fig. 1, where the small boxes represent objects of a class as written in the box, and the labelled arrows represent fields. For example, we have an object of class `Union`, with a field called `students` which points to an object of class `StudentList`.

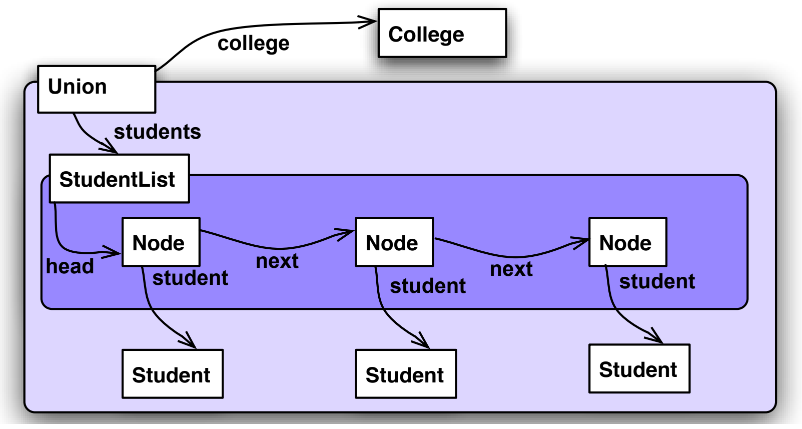

**Fig. 1.** An example of cloning domains

Cloning a `Node` object when the originator is itslef requires only the current object (the `Node`) to be cloned. Cloning a `Node` as a result of cloning a `StudentList` requires cloning all the successors of that node too, so that all the `Nodes` in the list are cloned. On the other hand, cloning the `StudentList` should not clone the `Students` pointed to by the `Nodes` in the list — however, if the `StudentList` is cloned as part of cloning the `Union` object, then the `Students` should also be cloned. Therefore, we say that the `Node` objects belong to the "cloning domain" of the `StudentList` object, as well as that of the `Union` object, while the `StudentList` and the `Student` objects belong to the cloning domain of the `Union` object.

The two cloning domains are depicted graphically in Fig. 1, through the two shaded rounded boxes. The cloning domains introduce a hierarchy among objects: when e.g. $o$ belongs to the cloning domain of $o'$ and $o'$ belongs to the cloning domain of $o''$, then $o$ belongs to the cloning domain of $o''$.

The hierarchy shown in Fig. 1 is very similar to that shown in various works on ownership types. In both cases, objects are put into boxes, and the boxes belong to objects. The difference is in the meaning of these boxes. In some cases, the boxes guarantee encapsulation[24], while in others they guarantee domination[8], or restrict who may modify the objects[16], or guarantee common de-allocation[1]. In our current work, the boxes guarantee that cloning the object which owns the box will also clone all objects in that particular box.

Note that the diagram in Fig. 1 presents a “ghost view” of the objects; the boxes, *i.e.,* the cloning domain information is *not* available at runtime. All the information we have is that `StudentList` will have a pointer field called `head` that points to its nodes; those `Nodes` have two pointer fields, one called `next` that points to the next `Node` and another called `student` that points to its `Student`.

To clone an object, we have to answer this question: for each object we reach, considering each of its fields in turn, *without* the runtime topological ownership information shown in Fig. 1: should we clone the object to which the field refers?

*Organisation of our paper* In Sect. 2 we describe our solution — clone annotated types — and give an example. In Sect. 3 we describe how clone annotated types can be used to decide whether or not a particular pointer should be cloned, and then in Sect. 4 we show how to generate the appropriate cloning methods. In Sect. 5 we describe the guarantees provided by our approach. In Sect. 6 we discuss related work and conclude.

## 2 Cloning Annotations

We propose to answer the question by introducing *cloning annotations* on field types. These annotations follow the tradition of ownership types [6, 8]. Every class is defined so as to take one or more formal *cloning parameters*, *c*. Thus, cloning annotated types have the form `C<c1,...cn>`, where `C` is a class, and `c1`, ... `cn` are the cloning parameters. The type `C<c1,..,cn>` expresses that the object has class `C`, and that it will be cloned whenever the object standing for `c1` is cloned; the remaining copy annotations, `c2`,..,`cn` may be used to annotate fields from that object and so describe when they are to be cloned.

The class definition determines the scope of these parameters — similarly to generic parameters. As in traditional ownership types, these parameters stand for objects. Again as in traditional ownership type systems, types are formed by classes followed by *actual cloning parameters*, *ca*, which may be any cloning parameter which is in scope, or `this`. The syntax is shown in Fig. 2.

$$\begin{array}{lll}
\mathit{ClassDecl} & ::= \texttt{class C}\langle\overline{c}\rangle & \\
 & \quad \{\ \overline{\mathit{FieldDecl}}\ \overline{\mathit{MethDecl}}\ \} & \\
\mathit{FieldDecl} & ::= \mathit{FieldType}\ \texttt{f} & \\
\mathit{FieldType} & ::= \texttt{C}\langle\overline{ca}\rangle & \\
c & ::= \mathit{Identifier} & \text{clone parameters} \\
ca & ::= c \mid \texttt{this} & \text{clone arguments}
\end{array}$$

**Fig. 2.** Syntax extracts

Note that the syntax of clone arguments does not allow any annotations which indicate that the cloning domain is unknown; thus, we do not support annotations such as `norep` from [10], or `any` from [9] or existentially bound arguments as in [2].

We use the term *cloning domain* of an object $o$ to describe all the objects which have to be cloned when $o$ is cloned. The first cloning parameter, $\mathtt{c}_1$, in the annotation of a field `f` determines that the object pointed at by `f` belongs to the cloning domain of $\mathtt{c}_1$: thus `f` must be cloned whenever $\mathtt{c}_1$ is cloned. The remaining cloning parameters are used to determine cloning for the fields of `f`.

```
class Node<c1, c2>{
   Node<c1,c2> next;
   Student<c2> student;
}
class StudentList<c>{
   Node<this,c> head;
}
class Union<c>{
   StudentList<this> students;
   College<c> college;
}
```

**Fig. 3.** Lists example

Fig. 3 shows cloning annotated class declarations that match the structure from Fig. 1. When a `Node` is cloned, then a new object of class `Node` needs to be created, and its fields need to be initialised according to those of the old object. When a `StudentList` is cloned, then a new object of class `StudentList` needs to be created, and *all* accessible `Node`s will have to be cloned — this is denoted by the `head` field's first cloning parameter being `this`. Finally, when a `Union` is cloned, then the `StudentList` will be cloned, all accessible `Node`s and all accessible `Student`s will have to be cloned.

## 3 Cloning Domains and Cloning Paths

Cloning domains have no runtime representation. In order to clone objects, we can only access their subordinate objects via *cloning paths* traced through objects' fields.

Inspecting Fig. 1 and Fig. 3 we can see how cloning paths lead to objects in different cloning domains. Consider paths beginning from a `StudentList` object. The objects at the paths `this.head`, or `this.head.next`, or `this.head.next.next` belong to the cloning domain of that `StudentList` object, but the object reachable through `this.head.next.student` does not. For paths starting at a `Union` object, objects reachable at `this.students`, or at `this.students.head`, or through `this.students.head.next`, and finally also at `this.students.head.next.next.student` belong to the cloning domain of the `Union` object.

Note that cloning domains are nested, and thus object may belong to more than one cloning domain. For example, for a `Union` object, `this.students`, `this.students.head` and `this.students.head.next` all belong to the `Union`'s cloning domain. On the other hand, the objects at `this.students.head` and `this.students.head.next` also belong to the cloning domain of the `StudentList` at the `this.students`, as well as to the `Union`'s cloning domain. Furthermore, it is possible for an object to be reachable from only one other object, but not belong to its cloning domain. For example, `this.head.next` is only reachable via `this.head`, but it does not belong to `this.head`'s cloning domain.

We now formalise the notion of cloning paths and cloning domains. We first define paths and path types as follows:

| | |
|---|---|
| $p \;\; ::= \texttt{this} \mid p.\texttt{f}$ | paths |
| $PT ::= \texttt{C}\langle\overline{cap}\rangle$ | path types |
| $cap ::= ca \mid p$ | actual path cloning parameters |

Path types express types which are relative to the current object `this`, and where paths can be used as actual cloning parameters. For example $\texttt{Node}\langle\texttt{this.students}, \texttt{this}\rangle$ describes an object of class `Node`, which belongs to the cloning domain of the object at `this.students`, and whose `student` field belongs to the cloning domain of `this`.

In Fig. 4 we define a judgment of the form $\texttt{C} \vdash p : \texttt{D}\langle cap_1, ...cap_n\rangle$ which says that from an object of class `C`, the path $p$ leads to an object in the cloning domain of $cap_1$. Furthermore, the actual path cloning parameters $cap_1$, ... $cap_n$ may be used to characterize the cloning domain of $p.f$, where $f$ is a field of $p$.

$$\frac{\texttt{class C}\langle \texttt{c}_1, ...\texttt{c}_\texttt{n}\rangle\{\;\; ... \;\;\}}{\texttt{C} \vdash \texttt{this} : \texttt{C}\langle \texttt{c}_1, ...\texttt{c}_\texttt{n}\rangle}$$

$$\frac{\begin{array}{c}\texttt{class D}\langle \texttt{c}_1, ...\texttt{c}_\texttt{m}\rangle\{\;\; ...\; \texttt{E}\langle \texttt{ap}_1, ...\texttt{ap}_\texttt{n}\,\rangle\; \texttt{f} \;\;\; ... \;\;\} \\ \texttt{C} \vdash \texttt{this}.\overline{\texttt{f}} : \texttt{D}\langle \texttt{cap}_1, ...\texttt{cap}_\texttt{m}\rangle\end{array}}{\texttt{C} \vdash \texttt{this}.\overline{\texttt{f}}.\texttt{f} : \texttt{E}\langle\, \texttt{ap}_1, ...\texttt{ap}_\texttt{n}[\texttt{cap}_1, ...\texttt{cap}_\texttt{m}/\texttt{c}_1, ...\texttt{c}_\texttt{m}][\texttt{this}.\overline{\texttt{f}}/\texttt{this}]\,\rangle}$$

**Fig. 4.** Path types for paths

The first rule says that in class `C` the path `this` has the type given by the cloning parameters of the class `C`. This gives, for example, that $\texttt{Union} \vdash \texttt{this} : \texttt{Union}\langle\overline{\texttt{c}}\rangle$.

The second rule says that from class `C` the path $\texttt{this}.\overline{\texttt{f}}.\texttt{f}$ has the type of `f` as given in `f`'s declaration, but where $\texttt{c}_1, ...\texttt{c}_m$, the cloning parameters of the class declaring field `f`, are replaced by $\texttt{cap}_1, ...\texttt{cap}_m$, the actual cloning parameters for $\texttt{this}.\overline{\texttt{f}}$. Moreover, any occurrence of `this` in the type of `f` is replaced by $\texttt{this}.\overline{\texttt{f}}$. The first replacement is standard in the literature, and the second replacement is novel. This rule gives $\texttt{Union} \vdash \texttt{this.students} : \texttt{StudentList}\langle\texttt{this}\rangle$ and $\texttt{Union} \vdash \texttt{this.students.head} : \texttt{Node}\langle\texttt{this.students}, \texttt{this}\rangle$.

Here are some more examples of the path type judgment:

$$\begin{gathered}
\texttt{Node} \vdash \texttt{this.next.next} : \texttt{Node}\langle\texttt{c1, c2}\rangle \\
\texttt{StudentList} \vdash \texttt{this.head.next.next} : \texttt{Node}\langle\texttt{this}, \texttt{c}\rangle \\
\texttt{StudentList} \vdash \texttt{this.head.student} : \texttt{Student}\langle\texttt{c}\rangle \\
\texttt{Union} \vdash \texttt{this.students.head} : \texttt{Node}\langle\texttt{this.students}, \texttt{this}\rangle \\
\texttt{Union} \vdash \texttt{this.students.head.student} : \texttt{Student}\langle\texttt{this}\rangle \\
\texttt{Union} \vdash \texttt{this.college} : \texttt{College}\langle\texttt{c}\rangle
\end{gathered}$$

Note that the judgments only talk about paths, and makes no requirement about consistent views of the heap. So far, nothing precludes two paths $\texttt{p}_1$ and $\texttt{p}_2$, which are aliases at runtime, but which have different path types, eg $\texttt{C} \vdash \texttt{p}_1 : \texttt{D}\langle\texttt{cap}_1, ...\texttt{cap}_\texttt{n}\rangle$ and $\texttt{C} \vdash \texttt{p}_2 : \texttt{D}\langle\texttt{cap}'_1, ...\texttt{cap}'_\texttt{n}\rangle$ and $\texttt{cap}_1 \neq \texttt{cap}'_1$. However, as we see in section 5.3, we require consistent views on the heap in order to obtain completeness, *i.e.,* that all objects which should be cloned, are, indeed, cloned.

We can now characterise for a given class C the set of paths to the objects in their cloning domain, $\mathit{ClnDom}(\texttt{C})$. This set consists of `this`, and all these paths p.f whose path-type has as first argument another path $\texttt{p}'$ (remember that all paths start at `this`), *i.e.,* $\texttt{C} \vdash \texttt{p.f} : \texttt{D}\langle\texttt{p}',...\rangle$, and which lie exclusively within the cloning domain of C, *i.e.,* $\texttt{p} \in \mathit{ClnDom}(\texttt{C})$.

$$\mathit{ClnDom}(\texttt{C}) \;=\; \{\,\texttt{this}\,\} \cup \{\,\texttt{p.f} \mid \texttt{C} \vdash \texttt{p.f} : \texttt{D}\langle\texttt{p}',...\rangle \text{ for a class D, and a path } \texttt{p}', \text{ and where } \texttt{p} \in \mathit{ClnDom}(\texttt{C})\,\}$$

Thus, we obtain that

$$\begin{gathered}
\texttt{this.next.next} \notin \mathit{ClnDom}(\texttt{Node}) \\
\texttt{this.head.next.next} \in \mathit{ClnDom}(\texttt{StudentList}) \\
\texttt{this.college} \notin \mathit{ClnDom}(\texttt{Union}) \\
\texttt{this.students.head.next.next.student} \in \mathit{ClnDom}(\texttt{Union})
\end{gathered}$$

The requirement that the complete path should lie within the cloning domain (i.e that $\texttt{p} \in \mathit{ClnDom}(\texttt{C})$) is relevant for the case where the cloning-owners are not dominators. For example, if the `College` object had a field `f` to some object which lied under the `StudentList` object, then `this.college.f` would not belong to $\mathit{ClnDom}(\texttt{Union})$.

## 4 Generating cloning methods

We have seen that the cloning behaviour of an object depends on the originator of the cloning action. For example, cloning a `Node` behaves differently when called as a result of cloning a `Union` or cloning a `StudentList`. This means that a single `clone()` method defined on each object cannot clone that object correctly in all contexts.

One approach would be to write a cloning method for every class that not only clones the object itself, but also clones every object within its cloning domain. Such a clone method would have to navigate through the appropriate paths and duplicate the appropriate objects. This approach is highly non-modular, since it exposes the internals of potentially every class.

Instead, we propose an overloaded parametric `clone(...)` method for each class, with additional Boolean actual parameters to describe whether or not its additional cloning domains should be cloned. For a class $\mathtt{C}\langle \mathtt{c}_1, ..\mathtt{c}_n \rangle$, we generate one overloaded parametric method, $\mathtt{clone}(\mathtt{Boolean}\ \mathtt{s}_1, ...\mathtt{Boolean}\ \mathtt{s}_\mathtt{n}, \mathtt{Map}\ \mathtt{m})$. The value of $\mathtt{s}_\mathtt{i}$ determines whether objects from the cloning domain $\mathtt{c}_i$ are to be cloned too. The original `clone` method can then be reimplemented to call the parametric cloning method, requesting only the object itself is cloned, by passing **false** to the additional cloning parameters.

The `Map m` formal parameter to the cloning method is to ensure that we clone the current object only if it has not already been cloned. We use the conventional solution of maintaining a table mapping original to cloned objects — something like a Smalltalk `IdentityDictionary` or Java 1.4's `IdentityHashMap` is ideal. If the object is not yet in the map, then the method will call the corresponding `clone` methods for all fields which need to be cloned, and will make aliases to all other fields — otherwise we simply use the clone stored in the map.

We show some examples of cloning methods in subsection 4.1 and then define the general case in subsection in 4.2.

## 4.1 A `clone` method for our example

For class Node, the basic `clone` method is called when the originator is the node itself, and simply delegates to the parametric cloning method. This method will be part of the interface of the cloning library.

```
Node clone( ){
    this.clone(false, false, new IdentityHashMap())
}
```

The parametric cloning method distinguishes whether the cloning domains of the clone parameters are to be cloned too, and accepts a `Map` to ensure each object is only cloned once. This method is internal to our system, and will not be visible outside the cloning library.

```
Node clone(Boolean s1, Boolean s2, Map m){
    Object n = m.get(this);
    if ( n != null) then {
       return (Node)n;
    } else {
       Node clone=new Node();
       m.put(this,clone);
       clone.next=  s1 ? this.next.clone(s1,s2,m) : this.next;
       clone.student= s2 ? this.student.clone(s2,m) : this.student;
       return clone;
     }
}
```

## 4.2 Code generation for `clone`

In Fig. 5 we show the generation of the basic cloning method for a class `C`. It calls the parametric cloning method on itself, where all the boolean parameters are set to `false`. This indicates that except for the cloning domain of the currnt object, no other cloning domains are "active".

In Fig. 6 we show the generation of the parametric cloning method for `C`.

As a first step, in the first four lines of the method, we need to determine whether the object is to be cloned. If the object is the outcome of a previous cloning action, *i.e.* is in the lookup table `m`, then nothing happens. Otherwise, a new object of that class is created, and entered into the table.

```
C clone(){
     return this.clone($false_1$, ...$false_q$, newIdentitityHashMap());
}
```

where $q$ is the number of clone parameters of class `C`

**Fig. 5.** The function `clone()` for class `C`

```
C clone(Boolean $s_1$...Boolean $s_q$, Map m){
     Object o = m.get(this)
     if o $\neq$ null then
          return (C)o;
     else{
          C o' = new C();
          m.put(this, o');
          o'.$f_1$ = $s_{1,1}$ ? this.$f_1$.clone($s_{1,1}$...$s_{1,k1}$, m) : this.$f_1$;
          ... = ...
          o'.$f_n$ = $s_{n,1}$ ? this.$f_n$.clone($s_{n,1}$...$s_{n,kn}$, m) : this.$f_n$;
          return o';
     }
}
```

where

{$\mathtt{f}_1$, ...$\mathtt{f}_n$} are the fields defined in class `C`

and where, for all $i \in 1..n$ :

($fType$(C⟨$s_1$...$s_n$⟩, $f_i$))[true/this]=$C_i$⟨$s_{i,1}$...$s_{i,ki}$⟩

for some classes $C_1$,.. $C_n$.

**Fig. 6.** The function clone($s_1$ ... $s_n$, m) for class `C`

As a second step, we need to initialize the fields of the newly crated object, and determine which further objects have to be cloned. We assume that {$\mathtt{f}_1$, ...$\mathtt{f}_n$} are all the fields defined in class `C`. Consider any field $\mathtt{f}_i$. We calculate the type of $\mathtt{f}_i$, and because anything that belongs to the cloning domain of the current object also belongs to the cloning domain of the owner

of the current object, we replace any occurrence of `this` by `true`. This gives that $\mathtt{C_i}\langle \mathtt{s_{i,1}}...\mathtt{s_{i,ki}}\rangle = \mathit{fType}(\mathtt{C}\langle \mathtt{s}_1...\mathtt{s}_n\rangle, \mathtt{f}_i)[\mathtt{true}/\mathtt{this}]$ for some $\mathtt{C}_i$. We find the cloning parameter corresponding to $\mathtt{s}_{i,1}$: If $\mathtt{s}_{i,1}$ is `true`, then the object at $\mathtt{f}_i$ has to be cloned and the result has to be assigned to the field $\mathtt{f}_i$. If $\mathtt{s}_{i,1}$ is `false`, then the object at $\mathtt{f_i}$ should not be cloned, but the field $\mathtt{f_i}$ of the new object needs to alias its value. This is why we emit the the conditional expression:
$\mathtt{o}'.\mathtt{f}_i = \mathtt{s}_{i,1}\ ?\ \mathtt{this}.\mathtt{f}_i.\mathtt{clone}(\mathtt{s}_{i,1}...\mathtt{s}_{i,ki}, \mathtt{m})\ :\ \mathtt{this}.\mathtt{f_i}$.

## 5 Guarantees of cloning

We now discuss the properties of our cloning operation, namely

- termination,
- soundness, *i.e.,* the objects which are cloned are those which are reachable through paths in the cloning domain,
- completeness, *i.e.,* if the system also guarantees a consistent view of the heap then all objects from the cloning domain will be cloned.

We do not present an operational semantics, because our system works for any imperative object oriented language with the standard meaning for field read and write, conditional expressions and method calls. Because the system works on the basis of statically known paths and their types, the runtime representation of objects does not need to be enhanced with cloning domain information - we do not even need any ghost information for the purposes of the formal argument.

Nevertheless, we expect an operational semantics of form $H, \phi, \mathtt{expr} \rightsquigarrow H', v$, where $H$ and $H'$ are heaps, $\phi$ is a stack frame, `expr` is an expression, $v$ stands for values, that values include `null` the booleans `true` and `false`, and addresses; and that addresses are represented as $\iota$, $\iota'$ etc. Furthermore, we expect that $H(\iota, f)$ returns the value of field $f$ of the object at $\iota$, that $H$ stores the class for each object, and that the stack frames $\phi$ map identifiers to values. Finally, we expect that expressions include field read, filed write, conditional expressions, and that these have the standard meaning.

The guarantees we describe in this section are applicable to richer languages, such that *e.g.,* support exceptions, or re-entrant method calls, or in fact, any further sequential control features. This is so, because our work makes guarantees about the effect of the code produced by our approach. This code only uses the language features listed above, and does not call any user-defined functions. The guarantees about the effects of the code remain valid regardless about the features used in the context that may be calling the generated `clone` methods, provided that this context is in the sequential setting.

In the following, we distinguish between the original call of the `clone()` method — without any arguments — and the subsequent recursive calls of the `clone(...)` methods which have at least one argument.

### 5.1 Termination

**Lemma 1.** *For all heaps $H$ and variables $x \in dom(\phi)$ there exists an address $\iota$ and a heap $H'$, such that $H, \phi, \mathtt{x.clone}() \rightsquigarrow H', \iota$.*

**Proof Sketch** We can show that during execution of `clone()` the values of fields in the objects in the original heap $H$ do not change, and that all receivers of any of the recursive calls of the `clone(..)` methods were accessible from the originator object, $\iota$, through a path $p$ in the original heap $H$.

Therefore, the transitive calls of the `clone` method do not clone objects from the original heap $H$ more than once, and do not clone any of the newly created objects. This gives a finite upper bound to the possible number of calls to the recursively called `clone(...)` methods. Since these methods do not contain any iteration, the fact that the number of possible recursive calls is finite, guarantees termination.

## 5.2 Soundness

With respect to the new heap, the cloning operation $H, \phi, \mathtt{x.clone}() \rightsquigarrow H', \iota$, makes the following four guarantees:

1. It creates a new object for the object `x`,
2. $H'$, the new heap, is homomorphic to $H$, the old heap,
3. In $H'$ there is a self-contained part that corresponds to the old heap,
4. In $H'$ there is a part that corresponds to the part in the old heap, which had to be copied according to the *ClnDom* function.

In order to express the last point formally, we define the set of objects that should be copied, as follows:

$\mathit{ToCopy} \quad : \quad \mathit{Addr} \times \mathit{Heap} \rightarrow \mathit{Power}(\mathit{Addr})$

$\mathit{ToCopy}(\iota)_H = \{\ H(\iota.\overline{\mathtt{f}}) = \iota' \mid \mathtt{this}.\overline{\mathtt{f}} \in \mathit{ClnDom}(\mathtt{C}), \text{and } \mathtt{C} \text{ the class of } \iota \text{ in } H\ \}$

We can now express soundness of the `clone` operation as follows:

**Lemma 2.** *If $H, \phi, \mathtt{x.clone}() \rightsquigarrow H', \iota$, then there exists a mapping $\alpha : \mathit{dom}(H') \rightarrow \mathit{dom}(H)$ such that*

1. *$\iota$ is new in $H$, and $\alpha(\iota) = \phi(\mathtt{x})$,*
2. *$\alpha(H'(\iota', \mathtt{f})) = H(\alpha(\iota'), \mathtt{f}))$ for all $\iota' \in \mathit{dom}(H')$ and fields $\mathtt{f}$.*
3. *$\alpha|_{\mathit{dom}(H)}$ is the identity function, and $\alpha|_{\mathit{dom}(H') \setminus \mathit{dom}(H)}$ is injective.* [3]
4. *$\alpha(\mathit{dom}(H') \setminus \mathit{dom}(H)) \subseteq \mathit{ToCopy}(\phi(x))_H$.* [4]

**Proof Sketch:** We can show that during execution of `clone` for the newly created `Map` object `m`, that the range of `m` is the original heap $H$, and that any object being mapped comes from the new heap $H'$, *i.e.,* for all objects $\iota'$, $\iota''$, if $\mathtt{m.get}(\iota'') = \iota'$ then $\iota'' \notin \mathit{dom}(H)$, and $\iota' \in \mathit{dom}(H)$. Furthermore, we can show that the lookup $\mathtt{m.get}(...)$ is injective, *i.e.,* for all objects $\iota'$, $\iota''$, if $\mathtt{m.get}(\iota') = \mathtt{m.get}(\iota'')$ then $\iota' = \iota''$.

[3] **Notation:** For an $f$ which is a mapping from $A$ to $B$, and for $A' \subseteq A$, we use the term $f|_{A'})$ to the function $f$ as restricted to the domain $A'$.

[4] **Notation:** For an $f$ which is a mapping from $A$ to $B$, and for $A' \subseteq A$, we use the term $f(A')$ to describe the image of $f$ from $A'$.

Because of injectivity, we can construct the mapping $\alpha(\iota')$ as the inverse of `m` in the cases where $\iota'$ is the image of an original object in $H$ (ie $\texttt{m.get}(\iota') \neq \texttt{null}$), and the identity otherwise.

Parts 1 and 3 follow from the construction of $\alpha$ and from the properties of the `Map` object mentioned above.

Part 2 follows from the body of the methods `clone(...)`.

For Part 4, we show that for any recursive application of any of the `clone(...)` methods, all the objects cloned so far are reachable from `x` through a path which lies completely within $\mathit{ClnDom}(\texttt{C})$, where `C` is the class of the object at `x`.

The latter follows from the following two facts

- All receivers of one of the recursive calls of the `clone(...)` methods are reachable from $\phi(\texttt{x})$ through a path that lies within $\mathit{ClnDom}(\texttt{C})$. This fact can be shown by induction on the recursive calls as follows:
  - The base case is easy.
  - For the inductive step, consider an object $\iota''$ which executes a method `clone(...)` as part of the n-th recursive call, and which calls a method `clone(...)` on an object $\iota'''$.
    Then, by construction of the `clone(...)` method, we know that the former call has the shape $\texttt{clone}(\texttt{true}, \texttt{b}_2, ...\texttt{b}_\texttt{q})$ and $q \geq 2$, that the latter call has the shape $\texttt{clone}(\texttt{b}'_1, ...\texttt{b}'_\texttt{r})$ and $r \geq 2$, and that $\texttt{b}'_1 = \texttt{true}$ such that $H(\iota'', f){=}\iota'''$, and where $\mathit{fType}(\texttt{C}' < \texttt{true}, \texttt{b}_2, ...\texttt{b}_\texttt{q} >, f) = \texttt{D} < \texttt{b}''_1, .., \texttt{b}''_\texttt{r} >$ for some $\texttt{b}''_1, .., \texttt{b}''_\texttt{r}$, s.t. $\texttt{b}''_1, .., \texttt{b}''_\texttt{r}[\texttt{true}/\texttt{this}] = \texttt{b}'_1, .., \texttt{b}'_\texttt{r}$, and where $\texttt{C}'$ is the class of the object $\iota''$.
    From the inductive hypothesis, we obtain that there exists a path from $\phi(x)$ to $\iota''$, which lies within $\mathit{ClnDom}(\texttt{C})$, *i.e.,* that $\texttt{C} \vdash p : \texttt{C}' < \texttt{p}', ... >$ for some path $p'$. Furthermore, the fact that $\texttt{b}'_1 = \texttt{true} = \texttt{b}''\texttt{1}[\texttt{true}/\texttt{this}]$ gives that $\mathit{fType}(\texttt{C}' < \texttt{c}_1, \texttt{c}_2, ...\texttt{c}_\texttt{q} >, f) = \texttt{D} < \texttt{ca}_1, .., \texttt{ca}_\texttt{r} >$ such that $\texttt{ca}_1 = \texttt{this}$, or that there exists an $m$ such that $\texttt{ca}_1 = \texttt{c}_\texttt{m}$ and $\texttt{b}_\texttt{m} = \texttt{true}$. In the first case, by definition of the judgment, we obtain that $\texttt{C} \vdash p.f : \texttt{D} < \texttt{p}', ... >$, while in the second case we obtain $\texttt{C} \vdash p.f : \texttt{D} < \texttt{p}, ... >$. In either case, the path $p.f$ lies within $\mathit{ClnDom}(\texttt{C})$.
- Only the objects that received one of the recursive calls of `clone(...)` may be have been cloned.

## 5.3 Completeness

Note, that Lemma 2 only guarantees that all cloned objects come from $ToCopy(\iota)_H$, and does not guarantee that *all* objects from $ToCopy(\iota)_H$ have been cloned. However, we can obtain this stronger property, if the system guarantees consistent views of the heap.

We first define consistent views of the heap to say that different paths leading to the same object must have the same path type.

**Definition 1.** *We say that a type system gives* consistent views of the heap*, if for all classes* C*, paths* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, and for all heaps* $H$ *and stack frames* $\phi$ *which may arise through execution of well-typed expressions, if* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *are aliases in* $H$ *and* $\phi$*, and* $\phi(\mathtt{this})$ *is an object of class* C*, then*
$\Gamma \vdash \mathtt{p}_1 : T_1$ *and* $\Gamma \vdash \mathtt{p}_2 : T_2$ *imply* $T_1 = T_2$

For example, if we had a system with consistent views of the heap, and if in our Fig. 1 the object of class `StudentList` contained a field called `last` which pointed to the same object as `this.head.next.next`, then, for typing environments $\Gamma_1$ and $\Gamma_2$, such that $\Gamma_1(\mathtt{this}) = \mathtt{StudentList}$ and $\Gamma_2(\mathtt{this}) = \mathtt{Union}$, we would also have that $\Gamma_1 \vdash \mathtt{this.last} : \mathtt{Node} < \mathtt{this}, \mathtt{c} >$ — this follows because of $\Gamma_1 \vdash \mathtt{this.head.next.next} : \mathtt{Node} < \mathtt{this}, \mathtt{c} >$. Furthermore, we would also have that $\Gamma_2 \vdash \mathtt{this.students.last.student} : \mathtt{Student} < \mathtt{this} >$ — this follows because of $\Gamma_2 \vdash \mathtt{this.students.head.next.next.student} : \mathtt{Student} < \mathtt{this} >$.

The requirement of consistent views guarantees that the underlying type system restricts field assignment so that paths pointing to separate cloning domains cannot be aliases. Such requirements are satisfied by any sound ownership type systems, *e.g.,* as early as the system in [8].

Consistency in the views of the heap implies consistency in the path types:

**Lemma 3.** *If a type system gives* consistent views of the heap*, then, for all classes* C*, paths* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, and for all heaps* $H$ *and stack frames* $\phi$ *which may arise through execution of well-typed expressions, if* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *are aliases in* $H$ *and* $\phi$*, and* $\phi(\mathtt{this})$ *is an object of class* C*, then*
$\mathtt{C} \vdash \mathtt{p}_1 : \mathtt{PT}_1$ *and* $\mathtt{C} \vdash \mathtt{p}_2 : \mathtt{PT}_2$ *imply* $\mathtt{PT}_1 = \mathtt{PT}_2$

Continuing our example from above, cosistent views of the heap would give us that $\mathtt{Union} \vdash \mathtt{this.students.last} : \mathtt{Node} < \mathtt{this.students}, \mathtt{this} >$, because $\mathtt{Union} \vdash \mathtt{this.students.head.next.next} : \mathtt{Node} < \mathtt{this.students}, \mathtt{this} >$.

**Lemma 4.** *Assume that the type system gives consistent views of the heap. If* $H, \phi, \mathtt{x.clone}() \rightsquigarrow H', \iota$*, then the mapping* $\alpha : dom(H') \rightarrow dom(H)$ *described in lemma 2 has the property that* $\alpha(dom(H') \backslash dom(H)) = ToCopy(\phi(x))_H$

**Proof Sketch:** We show by induction on $n$, that when considering the breadth first view of recursive applications of the `clone(...)` methods at depth $n$, then all objects which are reachable from `x` through a path of length at most $n$ which lies within $ClnDom(\mathtt{C}')$, have been cloned – with $\mathtt{C}'$ the class of the object at `x`.

In order to prove the above, we strengthen the assertion to say that for all $n$:

1. For all paths `p` and $\mathtt{p}'$, where $\mathtt{p} \in ClmDom(\mathtt{C}')$, and `p` has length at most $n$ such that $\mathtt{C}' \vdash \mathtt{p} : \mathtt{D} < \mathtt{p}_1, \mathtt{cap}_2...\mathtt{cap}_\mathtt{m} >$, if $H(\phi(\mathtt{x}), \mathtt{p}) = \iota$, then within the breadth first view of recursive applications of the `clone(...)` methods at depth $n$ there is a call of $\mathtt{clone}(\mathtt{true}, \mathtt{b}_2, ..\mathtt{b}_\mathtt{m})$ on $\iota$, where $\mathtt{b}_\mathtt{i}$=`true` if $\mathtt{cap}_\mathtt{i}$ has the form $\mathtt{this}.\overline{f}$, and `false` otherwise.

2. For any calls of $\mathtt{clone}(\mathtt{true}, \mathtt{b}_2, ..\mathtt{b}_\mathtt{m})$ on an object $\iota$ within the breadth first view of recursive applications of the `clone(...)` methods at depth $n$, there is a path $\mathtt{p}$ from $\phi(\mathtt{x})$ to $\iota$, such that $\mathtt{C}' \vdash \mathtt{p} : \mathtt{D} < \mathtt{p}_1, \mathtt{cap}_2 ... \mathtt{cap}_\mathtt{m} >$, and $\mathtt{b}_\mathtt{i}$=`true` if $\mathtt{cap}_\mathtt{i}$ has the form $\mathtt{this}.\overline{f}$, and `false` otherwise.

We now sketch the proof of the proposition from above:

- The base case is trivial.
- For the inductive step, we look at the two assertions separately:
  1. Consider any object $\iota'$ which is reachable from $\phi(\mathtt{x})$ through a path $\mathtt{p.f}$ of length $n+1$ such that $\mathtt{p.f} \in \mathit{ClmDom}(\mathtt{C}')$, and $\mathtt{C}' \vdash \mathtt{p.f} : \mathtt{D} < \mathtt{p}', \mathtt{cap}_2 ... \mathtt{cap}_\mathtt{m} >$ for some further path $\mathtt{p}'$. Therefore, there exists a further object $\iota''$, reachable from $\phi(\mathtt{x})$ through $\mathtt{p}$, such that $\mathtt{C}' \vdash \mathtt{p} : \mathtt{E} < \mathtt{cap}'_1, \mathtt{cap}'_2 ... \mathtt{cap}'_\mathtt{r} >$, and that $H(\iota'', f) = \iota'$, and $\mathtt{cap}'_1$ has the form of a path.
     By application of the inductive hypothesis part 1., we know that when considering the breadth first view of the recursive applications of the `clone(...)` methods at depth $n$, the object $\iota''$ will have received the recursive method call $\mathtt{clone}(\mathtt{b}_1, \mathtt{b}_2, ...\mathtt{b}_\mathtt{r})$ such that $\mathtt{b}_\mathtt{i}$ is `true`, if $\mathtt{cap}'_\mathtt{i}$ has itself the form of a path, and otherwise $\mathtt{b}_\mathtt{i}$ is `false`.
     We now inspect the first lines of the code of the `clone(...)` method. At the point of execution of the method call $\mathtt{clone}(\mathtt{b}_1, \mathtt{b}_2, ...\mathtt{b}_\mathtt{r})$ on $\iota'$ it is possible that the object $\iota'$ already had been entered in the map table `m`. By inspection of the `clone(...)` methods, this implies that $\iota'$ had already received the method call $\mathtt{clone}(\mathtt{b}'_1, \mathtt{b}'_2, ...\mathtt{b}'_\mathtt{r})$ for some $\mathtt{b}'_1, \mathtt{b}'_2, ...\mathtt{b}'_\mathtt{r}$.
     By application of the inductive hypothesis part 2, we know that there exists a path `p''` from $\phi(\mathtt{x})$ to $\iota''$, such that $\mathtt{C}' \vdash \mathtt{p}'' : \mathtt{E} < \mathtt{p}''_1, \mathtt{cap}''_2 ... \mathtt{cap}''_\mathtt{m} >$, and $\mathtt{b}'_\mathtt{i}$=`true` if $\mathtt{cap}''_\mathtt{i}$ has the form $\mathtt{this}.\overline{f}$, and `false` otherwise.
     By application of the consistent view property and lemma 3, we obtain that $\mathtt{b}'_\mathtt{i} = \mathtt{b}_\mathtt{i}$. We now inspect the remaining lines of the code of the `clone(...)` method. These will give that the object $\iota'$ will receive the method call $\mathtt{clone}(\mathtt{true}, \mathtt{b}''_2, ...\mathtt{b}''_\mathtt{m})$ where $\mathtt{b}''_\mathtt{i}$ is `true` if if $\mathtt{cap}_\mathtt{i}$ has itself the form of a path, and otherwise $\mathtt{b}''_\mathtt{i}$ is `false`.
  2. This assertion is proven by induction on the depth of method calls of `clone(...)`, and analysis of the body of the method.

## 6 Related Work and Conclusions

In this paper we have sketched a way of *specifying* cloning policies by annotating the fields of objects, and then *deriving* code for cloning which satisfies the cloning policy. Our lemma 2 guarantees that the derived cloning code produces a faithful copy of the cloned object, and that cloned objects are within the domain expressed by the types, while lemma 4 guarantees that if the types give consistent views of the heap, then *all* objects within the domain expressed by the types will be cloned. We have not yet dealt with subclasses, but we do not expect this to be challenging, because the cloning method will be bound dynamically to the class of the receiver. We plan to develop a detailed formal system and its proofs in further work.

Our work was inspired by Jensen et al.'s secure cloning policies [14]. Unlike the secure cloning policies, our work *generates* cloning methods, rather than simply checking them, and is based on ownership types, rather than a series of ad-hoc annotations. Compared with most ownership systems [5, 10, 9] our system is *descriptive* [3] in that it captures object relationships, rather than *prescriptively* ensuring invariants over the heap. We do not require or enforce heap properties (*e.g.,* clone/owners as dominators or owners as modifiers), nor even that the paths support a consistent view of the heap. In fact, owners as dominators could be too strong a requirement for cloning purposes, as it would unnecessarily forbid many programs, e.g. cloning a list with iterators. A consistent view of the heap may be enforced by the type system in the usual way [5]. We want to investigate in how far such a consistent view is the usual idiom, or whether there are situations where consistency is too restrictive from the practical perspective [15].

The relationship between ownership and cloning was first identified in a *dynamic* ownership setting, where every object knows its owner at runtime [17]. These ownership-based clones are called "sheep clones", as they are intermediate between shallow (single object) clones and deep (fully recursive) clones. Cloning, and programming models and languages based on cloning, have of course been studied more generally [4, 18, 20, 23]. Ownership type techniques have also been applied to other problems closely related to cloning, in particular object initialisation [11, 22], as cloning creates and initialises a new object — and ownership transfer [7], as transfer can be modelled by cloning an object then deleting the original. Cloning objects is also similar to comparing objects [12, 13] and generating hash codes [21].

As well as completing the formal model, in further work we plan to investigate whether it is possible to derive more efficient versions of the `clone` function, so as to avoid the use of the `Map` object when possible. We also want to support generics, and existential (unknown) clone parameters.

## Acknowledgments

We thank the FTfJP'11 reviewers for their constructive criticism on an earlier version of this paper, the FoVEOOS'11 reviewers for their feedback, and Paley Li for general discussions.

## References

1. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-Time Java. In European Conference on Object-Oriented Programming (ECOOP), 2006.
2. N. Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types.* PhD thesis, Imperial College London, 2009.

3. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM, 2007.
4. K. Campbell, J. McWhir, W. Ritchie, and I. Wilmut. Sheep cloned by nuclear transfer from a cultured cell line. *Nature*, 380:64–66, Mar. 1996.
5. D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.
6. D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *OOPSLA*, pages 292–310, Seattle, Washington, USA, November 2002. ACM Press.
7. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.
8. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM Press, 1998.
9. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, volume 4609 of *LNCS*, pages 28–53. Springer, 2007.
10. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8):5–32, October 2005.
11. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM Press, 2007.
12. P. Grogono and P. Chalin. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Data bases and Software Engineering (COODBSE'94)*, Montreal, Quebec, May 1994.
13. P. Grogono and M. Sakkinen. Copying and comparing: Problems and solutions. In *ECOOP*, pages 226–250, 2000.
14. T. Jensen, F. Kirchner, and D. Pichardie. Secure the Clones – Static Enforcement of Policies for Secure Object Copying. In *ESOP*, 2011.
15. P. Li. Sheep cloning for ownership. In *OOPSLA Doctoral Consortium*, 2011. To Appear.
16. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
17. J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific*, 1999.
18. J. Noble and B. Foote. Attack of the clones. In *Proceedings of the 2002 Australasian Conference on Pattern Languages of Programs*, volume 13 of *CRPIT*, pages 99–144, 2002.
19. J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *ECOOP*, Brussels, Belgium, July 1998.
20. J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Conecepts, Languages, Applications*. Springer-Verlag, 1997.
21. D. Rayside, Z. Benjamin, R. Singh, J. P. Near, A. Milicevic, and D. Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In *ICSE*, pages 342–352, 2009.
22. A. J. Summers and P. Müller. Freedom before commitment : A lightweight type system for object initialisation. In *OOPSLA*, 2011. To Appear.
23. D. Ungar and R. B. Smith. SELF: the Power of Simplicity. *LISP and Symbolic Computation*, 4(3), June 1991.
24. J. Vitek and J. Bokowski. Confined types for java. *Software Partice and Experience*, 2001.