Types for Deep/Shallow Cloning

Ka Wai Cheng
kwc108@ic.ac.uk

Supervisor: Prof. Sophia Drossopoulou
scd@doc.ic.ac.uk

Department of Computing
Imperial College London

June 19, 2012

Abstract

In Java, exact copies of objects are achieved by calling clone methods. By default, these clone methods create a shallow clone; if programmers desire sheep cloning (in between shallow and deep cloning) then they must override and maintain custom implementations of clone methods. For the programmer, this is tedious and error prone.

The cloning approach proposed in a recent paper, *Trust the Clones* [15], addresses this problem by using object ownership as the copy/clone policy and deriving the implementation of clone methods from them. In this report, we extend this cloning approach to deal with cloning when there are arrays, subclassing and generics. We explore the problems specific to each feature and propose solutions that allow clone methods to be derived from ownership types.

As a proof of concept, I implemented the basic cloning approach as a Java language extension using Polyglot (an extensible compiler framework). This compiles classes written in the extended Java language into standard Java with generated clone method implementation.

I also discovered situations where this cloning approach has problems – type errors sometimes occur when cloning without owners-as-dominators. We describe and formulate solutions to statically prevent these problematic cases from occurring.

Acknowledgements

Special thanks to my supervisor, *Prof. Sophia Drossopoulou*, for her support, patience and inspirational insights. Our lengthy discussions have been most useful in this project.

I would also like to thank *Raoul-Gabriel Urma* for his much appreciated advice and assistance regarding Polyglot.

Contents

Chapter 1

Introduction

When developing software in Java, care needs to be taken when passing/receiving internal objects to/from untrusted code. The security concern is that untrusted code can change the state of internal mutable objects. It is therefore recommended for methods to return copies of mutable objects and to use copies of mutable objects received from untrusted sources, unless sharing is intended [20, 25].

Creating a new object that is an exact copy of an existing object is known as cloning. Different methods of cloning are:

- *Shallow cloning* – where a copy of the top level object is created; values and references (not the actual objects) are copied from the original object. As a result, an object will share the same referenced objects as its shallow clone. For example, given the heap in Figure 1.1, the result of shallow cloning *Department* is shown in Figure 1.2a, where $oDepartment'$ is a shallow clone of *oDepartment*.
- *Deep cloning* – where cloning an object involves cloning all referenced objects recursively (to produce the same structure). For example, deep cloning *oDepartment* in Figure 1.1 results in Figure 1.2b, where $oDepartment'$ is a deep clone of *oDepartment*.

Both shallow and deep cloning have problems. With shallow cloning, it is possible to clone too little – we probably do not want *oDepartment*'s clone, $oDepartment'$, to reference the same *oStudentList*, since adding/removing any student from either department object will affect the other. On the other hand, it is also possible to clone too much when deep cloning – we may not want to have copies of student objects when cloning *oDepartment*.

In Java, cloning objects is often achieved through the `clone()` method in the `Object` class, which by default performs shallow cloning; and deep cloning can be achieved by serialising objects. On the other hand, to produce object clones that are in between shallow and deep clones, the programmer can override the implementation of `clone()` for each class of objects that are permitted to be cloned. In this way, the programmer can specify precisely which referenced objects to clone and how deep to clone.

The disadvantages of manually overriding `clone()` are that it is tedious and error prone because it is the programmer's responsibility to ensure the clone implementation is correct. This is not always a good idea, for example if fields are added/changed, the programmer may forget to modify the clone implementation. Proposals to address this include: forming copy policies and perform code analysis to check theses policies are adhered to [20]; and sheep cloning [15, 21, 26], which is in between shallow and deep cloning, achieved using a type system that incorporates object ownership to guide cloning (Section 2.3).

In particular, the cloning approach proposed in *Trust the Clones* [15] uses object ownership as the copy/clone policy and derives the implementation of clone methods from them (rather than being manually implemented by the programmer). Further details of this approach are included in Section 2.2. The proposed approach has not yet been implemented and has not yet dealt with subclasses, arrays and generics classes.

This project expands on this cloning approach, making the following contributions:

- A description of situations in which this cloning approach is problematic, and solutions to this (Section 3.1.1).

This problem arises based on the ownership structure of objects and field references between them; in some situations, clone objects may have type errors. Solutions proposed detect and prevent these problematic situations from occurring.

- An extension for generating clone methods for arrays (Section 3.3).

  An array is an object that references a number of other objects (the array elements), which we may or may not want to clone when the array object is cloned. One problem is that the array class is predefined in Java and we would prefer not to make change to the predefined class. I propose a solution that generates an array clone method for each class and dimension array combination.

- An extension for generating cloning methods for subclasses (Section 3.4).

  This involved solving the problem that objects could be assigned to fields that are declared as a superclass of the object and may expect different arguments to be passed when the field is cloned. The idea of the solution is to only allow assignments of subclasses if the correct clone implementation can be derived from the clone method arguments that would be passed to the superclass.

- An extension for generating cloning methods for generic classes (Section 3.5).

  The problem with this is that the actual type parameters are not known until an instance of the generic class is created, so there is not any infomation to derive clone implementation from. The solution is to store dynamic information that enables ownership information to be derived.

- An implementation of the basic generation of clone methods using ownership types (Chapter 4).

  Thisis achieved by building a pre-processor that will transform programs written in the extended Java language with parameterised ownership types into the standard Java language with generated clone methods.

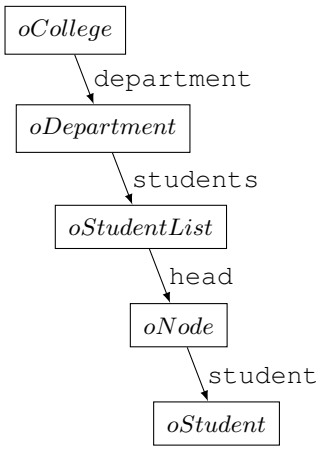
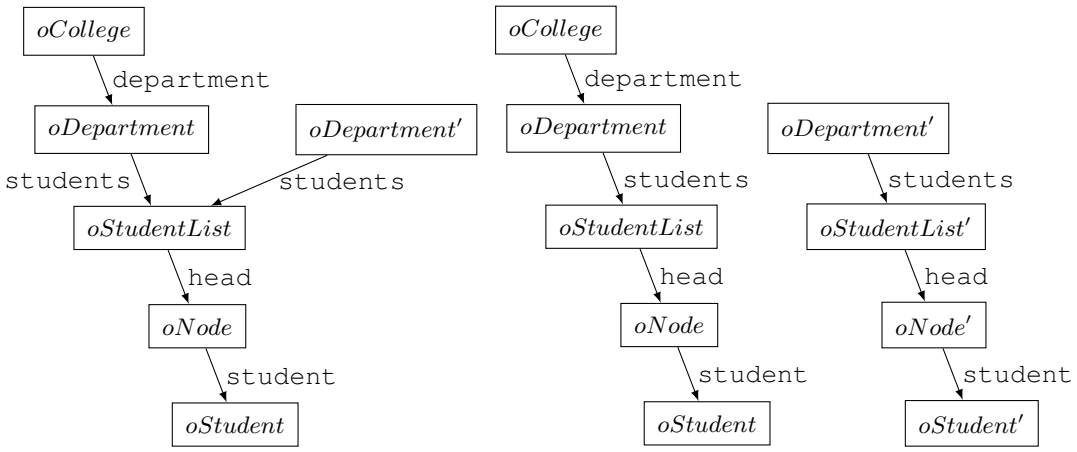


Figure 1.1: Example structure of objects



(a) Result of shallow cloning *oDepartment*

(b) Result of deep cloning *oDepartment*

Figure 1.2: Examples of cloning

Chapter 2

Background

In order to understand the details of the project, the necessary background information is included in this chapter, including existing solutions and frameworks that this project uses, and formal notation used in this report.

2.1 Notation

This section presents the formal notation used in throughout this report.

Figure 2.1 contains the identifier conventions used in the illustrative examples. Object identifiers are the object's class name prefixed with o and identifiers for object clones will be the original object's identifier appended with prime ($'$). Traditionally, field types are the defined $ClassId$'s, however, types are extended by the cloning approach we are adopting (explained in Section 2.3.2).

The basic syntax for the heap and type environment are shown in Figure 2.2. Objects on the heap are tuples consisting of the object type and a mapping of field identifiers to values. This representation of objects will be extended to include ownership information in Section 2.2, where we explain the notion of Object ownership. Values can be object identifiers, `null` or boolean values: `true` and `false`. The heap maps object identifiers to objects and type environment maps object identifiers to types.

$o \in ObjectId$	Object identifiers
$C \in ClassId$	Class identifiers
$f \in FieldId$	Field identifiers
$t \in FieldType$	Field types

Figure 2.1: Identifier conventions

$\chi \in Heap =$	$ObjectId \rightarrow Object$	Heap
$\Gamma \in TypeEnv =$	$x \rightarrow t$	Type environment
$Object =$	$t \times (FieldId \rightarrow v)$	Object
$v \in Values =$	$ObjectId \cup \{\texttt{null}, \texttt{true}, \texttt{false}\}$	Values

Figure 2.2: Syntax for heaps and type environment

Given the syntax, $\chi(o)$ is the object that o is an identifier for, and $\chi(o) \downarrow_2 (f)$ is value in field f of the object, $\chi(o)$. As shorthand, I define the following for field access: $object(f) = object \downarrow_2 (f)$. For

example, the heap for Figure 1.1 and some example field accesses are:

$$\begin{aligned}
\chi(oCollege) &= (\texttt{College}, (\texttt{department} \mapsto oDepartment)) \\
\chi(oDepartment) &= (\texttt{Department}, (\texttt{students} \mapsto oStudentList)) \\
\chi(oStudentList) &= (\texttt{StudentList}, (\texttt{head} \mapsto oNode)) \\
\chi(oNode) &= (\texttt{Node}, (\texttt{student} \mapsto oStudent, \texttt{next} \mapsto \texttt{null})) \\
\chi(oStudent) &= (\texttt{Student}, \emptyset)
\end{aligned}$$

$$\begin{aligned}
\chi(oCollege)(\texttt{department}) &= oDepartment \\
\chi(oStudentList)(\texttt{head}) &= oNode
\end{aligned}$$

$$p ::= \texttt{this} \mid p.f$$

Figure 2.3: Syntax for field paths

In our discussions in the main content of this report, we talk about paths between objects via fields accesses; the syntax for paths is shown in Figure 2.3, where `this` refers to the object that we are starting from. The composition of paths is defined as:

$$p_1 p_2 = \begin{cases} p_1 & p_2 = \texttt{this} \\ (p_1 p').f & p_2 = p'.f \end{cases}$$

Following a path from an object on the heap is written as $\chi(o, p)$ and is defined as:

$$\chi(o, p) = \begin{cases} o & p = \texttt{this} \\ \chi(o, p')(f) & p = p'.f \end{cases}$$

Using the heap, χ, from the previous example, the following are some examples of path compositions and results of following paths from objects in the heap:

$$\begin{aligned}
(\texttt{this.department.students})(\texttt{this.head}) &= \texttt{this.department.students.head} \\
(\texttt{this})(\texttt{this.student}) &= \texttt{this.student}
\end{aligned}$$

$$\begin{aligned}
\chi(oCollege, \texttt{this.department}) &= oDepartment \\
\chi(oStudentList, \texttt{this.head.student}) &= oStudent
\end{aligned}$$

2.2 Ownership Types

The notion of object ownership is that every object has an owner and can own other objects. An owner can be an object or `world`, which owns objects that are not owned by other objects. The representation of an object, o, refers to the objects that have o as their owner; there is no restriction that objects referenced by o through its fields must be in o's representation. Ownership is acyclic, so forms a tree of objects with `world` at the root.

The hierarchical structure of objects in the heap can be represented diagrammatically with boxes such as in Figure 2.4. In these diagrams, objects are rectangles with identifiers starting with o and arrows are field references. An object's representation is shown through rounded boxes, where the owner is the object on the edge of the box and objects inside the box are its representation. For example, *oCollege* is the owner of *oDepartment* and *oStudent*. Boxes are omitted in cases where the representation is empty such as for *oNode* and *oStudent*. The outermost objects (that are not strictly in a box), i.e. *oCollege*, are owned by `world`.

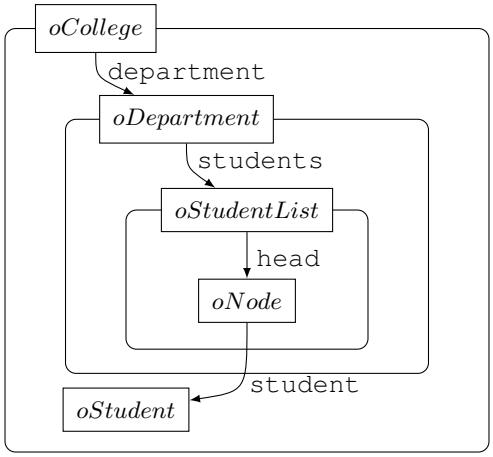


Figure 2.4: Example of object ownership diagram

$$\begin{aligned} Object &= FieldType \times (f \mapsto v) \times Owner \\ Owner &= \{\texttt{world}\} \cup ObjectId \end{aligned}$$

Figure 2.5: Extended syntax for objects on the heap, with owners

The syntax for objects on the heap from Figure 2.2 is extended with ownership information by adding a third element in the tuple as shown in Figure 2.5. This third element is either `world` (not owned by any other object) or an object with the given identifier. For example, the heap for Figure 2.4 is:

$$\begin{aligned} \chi(oCollege) &= (\texttt{College}, (\texttt{department} \mapsto oDepartment), \texttt{world}) \\ \chi(oDepartment) &= (\texttt{Department}, (\texttt{students} \mapsto oStudentList), oCollege) \\ \chi(oStudentList) &= (\texttt{StudentList}, (\texttt{head} \mapsto oNode), oDepartment) \\ \chi(oNode) &= (\texttt{Node}, (\texttt{student} \mapsto oStudent, \texttt{next} \mapsto \texttt{null}), oStudentList) \\ \chi(oStudent) &= (\texttt{Student}, \emptyset, oCollege) \end{aligned}$$

Ownership types introduce ownership information to types and the type system will ensure the ownership relation is not violated statically. For example, we can declare a variable, `v`, that has static type `Department` with owner *oCollege*, often written with parameterised types as `v:Department<`*oCollege*`>`. Hence, `v` can reference any object instance of class `Department`, as long as its owner is the object *oCollege*.

We also say an object, $o1$, is *inside* an object, $o2$, iff $o1$ is in $o2$'s transitive representation or $o1$ is $o2$ itself.This inside relation is written formally as $o1 \prec^* o2$ and satisfies the reflexivity and transitivity properties. This relation is clear in the diagrams as all objects contained in the box (including the owner of the box) are inside the owner of the box. For example, the following holds for Figure 2.4:

$$\begin{aligned} oDepartment &\prec^* oDepartment \\ oDepartment &\prec^* oCollege \\ oStudent &\prec^* oCollege \\ oStudentList &\prec^* oDepartment \\ oStudentList &\prec^* oCollege \end{aligned}$$

The opposite idea, $o1$ is *outside* $o2$ iff $o1 \not\prec^* o2$ (i.e. $o1$ is not inside $o2$), this does not imply $o2$ is inside $o1$, although the converse is true ($o2$ inside $o1$ implies $o1$ is outside $o2$). For example, the following

holds for Figure 2.4:

$$oCollege \not<^* oDepartment$$
$$oStudent \not<^* oDepartment$$
$$oDepartment \not<^* oStudent$$
$$oStudent \not<^* oStudentList$$
$$oStudentList \not<^* oStudent$$

2.2.1 Owners-as-Dominators

Objects in the heap can be viewed as nodes in a graph where field references are directed edges (arrows) between object nodes. A node, d, dominates a node, n, *iff* every path from the start node to n goes through d. Similarly, an object, od, dominates an object, o, *iff* all paths from the root of the system to o goes through od.

Owners-as-dominators is a property that may be enforced in object ownership systems [30, 14]. This property states all owners dominate the objects in their representation. Hence, owners dominate all objects that they contain (i.e. itself and its transitive representation). This means that there are no arrows/field references that enter boxes (cross over box boundaries). Figure 2.6 shows examples of valid references and invalid references (striked arrows), where the references to $o5$ from $o7$ and $o9$ are not valid (reference should be through $o3$) and $o1$ should not directly reference $o2$.

The owners-as-dominators guarantee can be useful such as for memory management where deallocating memory for an object, o, means we can safely deallocate memory for all objects inside o. However, this property restricts the number of programs that can be written such as those that use iterators to traverse a lists of objects. The iterator object and list object will not be able to reference the node/link objects of the list at the same time.

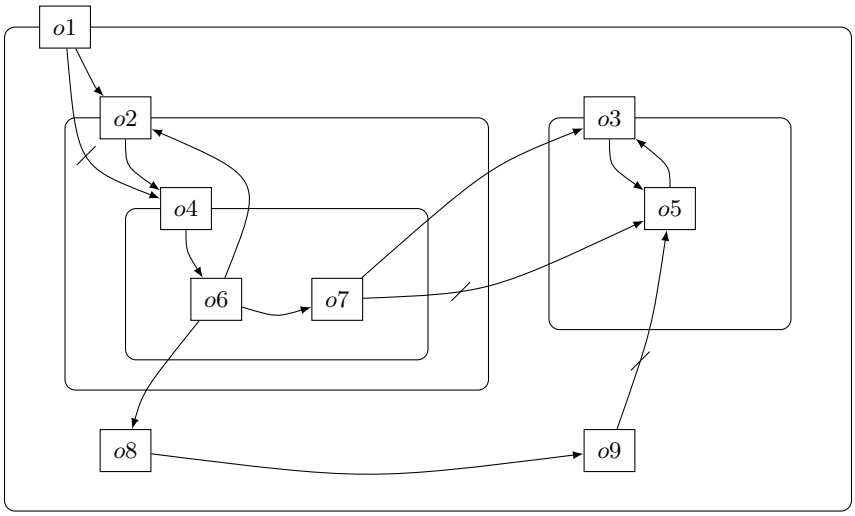


Figure 2.6: Example of valid references (plain arrows) and invalid references (crossed arrows) when owners-as-dominators is enforced

2.2.2 Related Works

Ownership types are first introduced to restrict aliasing in object oriented programming, and works included *Flexible Alias Protection* [13, 28]. When an object, o, has aliases (many objects, os, reference o) then if an object in os modifies o's state then all the other objects in os are affected.

The use of ownership types for sheep cloning is described in Section 2.3.1. Related to cloning, ownership types have also been made possible to transfer of ownership and minimal cloning in dealing with concurrency [12], where a cloning approach is similar to sheep cloning is used (owner parameters determine whether an object's fields should be cloned). Here, cloning returns a unique reference and ownership can be transferred.

A few other applications of ownership types include:

- Memory management [7], where owners-as-dominators can be used so that when deallocating memory of an object, o, we can safely deallocate memory of all objects inside o. This is possible because having the owners-as-dominators property means that o is the single point of access for all objects inside o, so deleting o will leave all objects that were inside o to be unreachable.
- Preventing programs from having *data races*, which is where multiple threads access the same data and one or more threads changes the value, without having to lock every object [6]. In this system, acquiring a lock to an object means that all objects transitively owned are locked.
- Enforcing the architecture of an application using ownership [4] to prevent lower layers in the application from accessing the upper layers, but allow objects to access objects in layers below it; this uses different owners to define different layers.

2.3 Cloning Approach

This project is based upon and extends the cloning approach proposed in *Trust the Clones* [15], which is explained in this subsection. In this ownership system, owners of objects do not change over time, all owners are known and owners-as-dominators is not enforced. However, after some investigation, I have found that this cloning approach will have type errors when cloning in some situations where enforcing owners-as-dominators is not enforced; the details are in Section 3.1.1.

Main features of this approach:

1. Uses ownership types (Section 2.2) to determine the set of objects to clone when an object is cloned, called its cloning domain, (Section 2.3.1).
2. Extends a subset of the Java language with ownership types by adding annotations (Section 2.3.2).
3. Generates the clone methods in standard Java, by deriving them from the annotations (Section 2.3.3).

Related Works. Issues with cloning objects have been studied in general including depth of cloning, equality between objects, cyclic object structures [17]. Cloning should produce a new object, which is considered as equal to the original; in the cloning approach we adopted we require structural equivalence between objects and their clones. Other notions of equality highlighted in the paper [17] include a deeper comparison of objects.

2.3.1 Ownership Types and Cloning

For cloning, the use of ownership types was proposed in *Object Ownership for Dynamic Alias Protection* [26], where ownership can be used to define the cloning domain of an object, o, (objects that require cloning when o is cloned). The cloning domain of o includes o itself and all objects that are reachable from o via some path that is inside o at every step of the path, i.e. does not go outside of o's box. Otherwise, all referenced objects that are outside of o should be shared with the clone (object references are copied). Diagrammatically, this means that all reachable objects inside the box should be cloned when the owner of the box is cloned.

For example, cloning $oDepartment$ involves cloning $oStudentList$ and $oNode$ is cloned as part of the process of cloning $oStudentList$, resulting in Figure 2.7 where $oNode'$ references the original $oStudent$. On the other hand, cloning $oCollege$ will result in Figure 2.8, where $oStudent$ is also cloned as part of cloning $oNode$ because $oStudent$ is inside $oCollege$. These two examples highlight that the cloning behaviour of an object (such as $oNode$) depends on the originator of the cloning process (the outermost owner being cloned).

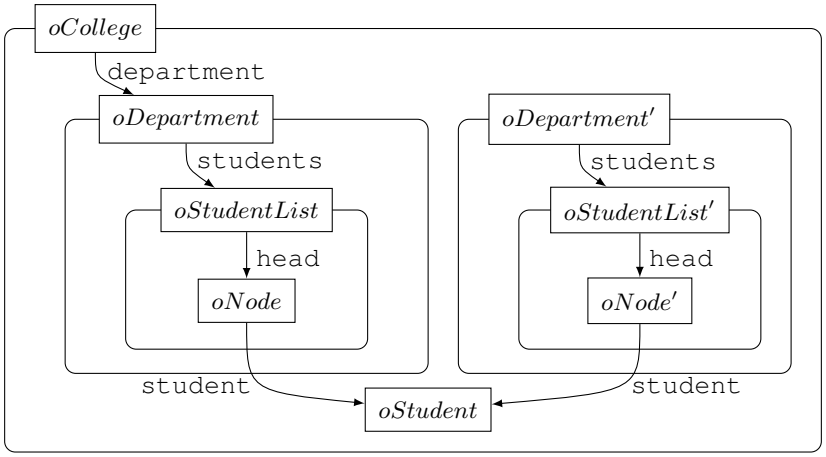


Figure 2.7: Example of sheep cloning *oDepartment*

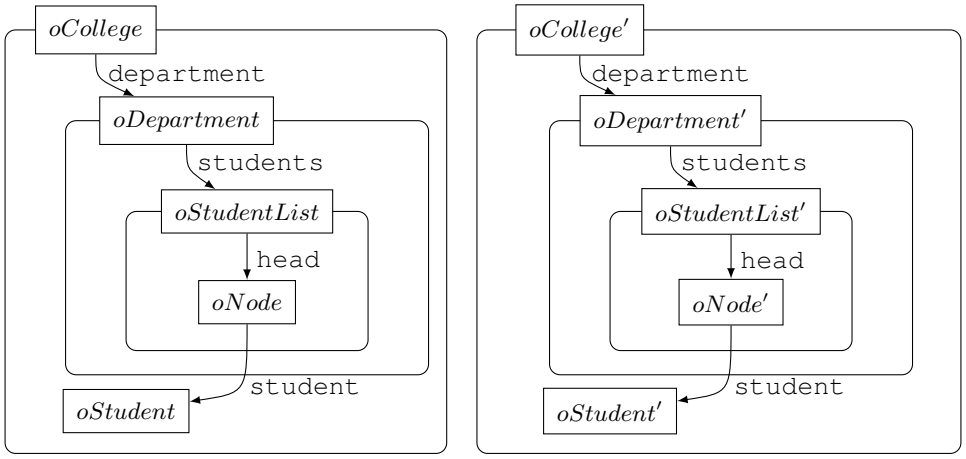


Figure 2.8: Example of sheep cloning *oCollege*

2.3.2 Clone Annotations

Extending Java with ownership types is achieved by adding annotations to types, similar to type parameters for generic types, the proposed extended syntax is given in Figure 2.9. Each class has one or more formal owner/cloning parameters, $\overline{c}$, which are identifiers representing objects. The first parameter is the owner of the class; hence $\overline{c}$ is always a non-empty list. The other cloning parameters can be used in the cloning parameters of fields declared in the class. Listing 2.1 gives an example of class declarations corresponding to the objects in Figure 2.4 using this extended syntax.

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

Figure 2.9: Extended Java syntax

Given the extended class declarations, object types are of the form `C<ca1,...,can>`, where `C` is the class and `ca1,...,can` are the actual cloning parameters, $\overline{ca}$. Actual cloning parameters instantiate

the formal cloning parameters of the object – these can be formal cloning parameters in scope, or `this`. To illustrate with the current example, the types of the objects are:

$$\begin{aligned} oCollege&:\texttt{College}\langle\texttt{world}\rangle \\ oDepartment&:\texttt{Department}\langle oCollege\rangle \\ oStudentList&:\texttt{StudentList}\langle oDepartment,oCollege\rangle \\ oNode&:\texttt{Node}\langle oStudentList,oCollege\rangle \\ oStudent&:\texttt{Student}\langle oCollege\rangle \end{aligned}$$

```
1  class College<c> {
2    Department<this> department;
3  }
4
5  class Department<c> {
6    StudentList<this,c> students;
7  }
8
9  class StudentList<c1,c2> {
10   Node<this, c2> head;
11 }
12
13 class Node<c1,c2> {
14   Student<c2> student;
15   Node<c1,c2> next;
16 }
17
18 class Student<c> {
19 }
```

Listing 2.1: Example class declarations

2.3.3 Cloning Methods

Cloning an object involves recursively cloning referenced objects that are *inside* the object that initiated the cloning process. Therefore, when cloning an object, o, we need to know whether the objects represented by o's formal cloning parameters are also to be cloned. This was illustrated in Figure 2.7 and Figure 2.8 where the cloning $oNode$:`Node<`$oStudentList$`,`$oCollege$`>` involves cloning $oStudent$ whenever $oCollege$ is being cloned.

To indicate which relevant objects are being cloned, clone is overloaded so that each class, `C<`$\texttt{c}_1$`,...,`$\texttt{c}_n$`>`, has a parametric clone method: `clone(Boolean` $\texttt{s}_1$`,..., Boolean` $\texttt{s}_n$`, Map m)`. The purpose of this method is to clone class instances as part of the cloning process of some object and so will only be called from within other clone methods. The value of $\texttt{s}_i$ (ith Boolean parameter) indicates whether objects inside $\texttt{c}_i$ (ith formal cloning parameter) should be cloned. Hence, the fields that should be cloned are those with the first actual cloning parameter, ca, s.t. $ca \in \{$ `this` $\} \cup \{c_i : 1 \leq i \leq n$ and $s_i =$ `true` $\}$.

Starting the actual clone process is by invoking the original `clone()` method, which calls the parametric clone method of the same class. The generated implementation of this method for class `C<c1,...cn>` is given in Listing 2.2 (lines 1 - 3). `false` is passed for all Boolean parameters, which means only fields with owner, `this`, will be cloned. A new Map (mapping `Objects` to `Objects`) is passed as the last parameter to keep track of the objects already cloned. This prevents objects from being cloned multiple times in the same cloning process.

```
1  C clone() {
2    return this.clone(false1,...,falsen, new IdentitityHashMap());
3  }
4
5  C clone(Boolean s1,...,Boolean sn, Map m){
6    Object o = m.get(this);
7    if o != null then
8      return (C)o;
9    else {
10     C oClone = new C();
11     m.put(this, oClone);
12     oClone.f1 = s1,1 ? this.f1.clone(s1,1,...,s1,k1, m) : this.f1;
13     ... = ...
14     oClone.fn = sq,1 ? this.fn.clone(sq,1,...,sq,kq, m) : this.fn;
15     return oClone;
16   }
```

```
17 }
18
19 where
20   {f_1, ..., f_q} are the fields defined in class C
21 and where, for all i ∈ {1, ..., n} :
22   (fType(C⟨s_1, ..., s_n⟩, f_i))[true = this] = C_i⟨s_{i,1}, ..., s_{i,ki}⟩
23 for some classes C_1, ...C_n
```

Listing 2.2: General implementation of clone methods.

The implementation of the parametric clone method is given in Listing 2.2 (lines 5 - 17), which performs the main steps:

1. Return the existing clone of this object if it exists (lines 6 - 8).
2. Create and record a new clone, o', of this object (lines 9 - 11).
3. For every field, $\texttt{f}_i$, in this class (lines 12 - 14):
   (a) Calculate the type of field, $\texttt{f}_i$, in an object of type $\texttt{C<s}_1\texttt{,...,s}_n\texttt{>}$ (this transforms the list `Boolean` parameters in the same way `C`'s list of formal cloning parameters is transformed to give $\texttt{f}_i$'s actual cloning parameters) and replace all occurrences of `this` with `true`. Hence, we have the `Boolean` values indicating the cloning parameters of $\texttt{f}_i$ that are being cloned. This refers to line 22.
   (b) If $\texttt{s}_{i,1}$ is true, this means that the field should be cloned so assign to $\texttt{oClone.f}_i$ the result of cloning $\texttt{f}_i$, passing the Boolean values $\texttt{s}_{i,1}\texttt{...s}_{i,ki}$. Otherwise, assign to $\texttt{oClone.f}_i$ the object referenced by $\texttt{f}_i$ and do not clone.

An example of the clone methods of class Node is shown in Listing 2.3. In particular, the Boolean parameters, `s1` and `s2`, are passed in lines 19 - 22 in the order of the field's actual cloning parameters wrt class `Node`'s formal cloning parameters.

```
1  Node clone( ){
2    this.clone(false, false, new IdentityHashMap());
3  }
4
5  Node clone(Boolean s1, Boolean s2, Map m){
6    Object n = m.get(this);
7    if (n != null) {
8      return (Node)n;
9    } else {
10     Node clone = new Node();
11     m.put(this,clone);
12     clone.next= s1 ? this.next.clone(s1,s2,m) : this.next;
13     clone.student= s2 ? this.student.clone(s2,m) : this.student;
14     return clone;
15   }
16 }
```

Listing 2.3: Clone methods for class Node.

2.4 Extensible Compiler Frameworks

Language extensions can be implemented as a preprocessor, which is a compiler that accepts source code in the extended language and outputs an equivalent program in the target language. To make the implementation easier, extensible compiler frameworks can be used such as *Polyglot* [29], *JaCo* [1] and *JastAddJ* [16] for Java. These are Java compilers by themselves and are extended to create compilers for extended/modified Java languages. The programmer only needs to write code for the changes to the Java language – less demanding for the programmer than the alternative of building the preprocessor from scratch. In addition, the created compiler is also extensible for any future extensions or modifications to the language.

Out of these extensible compiler frameworks, I chose Polyglot for this project. Polyglot has a relatively large number of projects with source code available online, of which some involve extending Java with parameterised types such as *PolyJ* [3], *Jif* [24] and *Coffer* [29]. Together with more documentation and support, this helps greatly with understanding how Polyglot works internally and how to extend Polyglot for this project especially for similarities such as extending with parameterised types. Unlike Jaco and JastAddJ, Polyglot is written in Java so using tools available in IDE's such as Eclipse helps

with debugging and code investigation. On the other hand, Jaco is written in a Java-like language and uses extensible algebraic datatypes with defaults, and JastAddJ is implemented using JastAdd [2] so the source code uses JastAdd syntax.

Polyglot

Polyglot is a framework written in Java and by itself, it is a semantic checker for Java 1.4. Extensions/-modifications to Java, including changes to syntax and semantics, are implemented by extending Polyglot (or other Polyglot extensions) and changing parts of the compilation process. The extended compiler translates programs written in the modified language into standard Java source code. Optionally, the translated program can be compiled into Java bytecode.

Currently, there are many language extensions implemented using Polyglot, ranging from small to large scale extensions. Examples of the more complex extensions are *Jif* [24], which adds support for information flow control and access control, and *PolyJ* [3, 23], which adds support for parametric polymorphism. Smaller scale extensions include *Coffer* [29], which extends Java with resource management, and J_0 [5], which is aimed at novice programmers and allows methods and statements without defining classes. There is also an extension, *Polyglot5* [22], supporting Java 1.5 and can be used instead of the base Polyglot as the starting point for language extensions.

Polyglot uses Polyglot Parser Generator (PPG) [8], an extensible LALR parser generator based on CUP [18]. PPG allows programmers to modify the syntax of Java for the extension by adding, removing or changing productions (rewrite rules) and symbols from the grammar (the syntax rules) of the base language being extended.

Extending Polyglot

This section briefly describes the main steps for implementing a new Polyglot extension. A more detailed tutorial has been written by Raoul-Gabriel Urma and is available from the Polyglot homepage [32].

To start, the Polyglot source code includes a shell script to create the skeleton files for new extensions. For extensions that change/modify the Java grammar, the programmer needs to specify the changes in the ppg file generated. The PPG syntax and semantics is available online[8] and is quite straightforward.

The compilation process of an extended Polyglot compiler is shown in Figure 2.10. First the source code written in the extended language is parsed by the extended parser and produces an extended Abstract Syntax Tree (AST) that represents the program. The extended AST is extended from the AST of the base compiler, which is a tree of nodes implementing the `Node` interface and are constructed by node factories. The extended AST may have new classes of nodes to represent the new syntax or record new information. These extra classes are defined in the extension's AST package, and all AST nodes must be created through the node factory class (one of the extension skeleton files).

Once the extended AST is obtained, the *pass scheduler* manages the compilation passes run over the AST of each source file, ensuring that dependencies between files are maintained. A compilation pass rewrites the AST by performing semantic analysis or translation to Java. If the compilation passes are successful, the result should be a Java AST and Polyglot's code generator will output the Java source code from the AST.

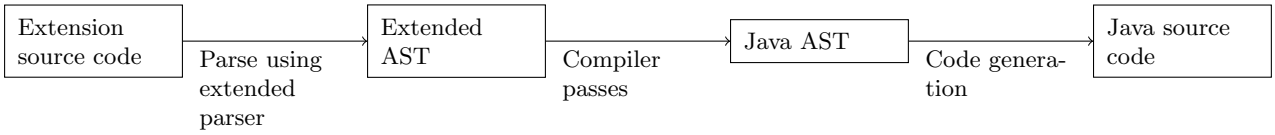


Figure 2.10: Compilation process of an extended Polyglot compiler

Chapter 3

Extending Ownership Types for Cloning

There are problems/issues related to cloning using ownership types that remain to be solved for generating cloning methods. The possible solutions to these problems are discussed and evaluated here, including: extending this cloning approach, by generating cloning methods, to arrays, subclasses and generics. First, desirable properties of cloning are described, which form the basis for evaluating alternative solutions for each extension. We will consider each feature separately as an extension to the approach as proposed in *Trust the Clones*, before bringing these features together in a single extension in Section 3.6. This allows us to focus on the problems specific to each feature: arrays (Section 3.3), subclasses (Section 3.4), and generics (Section 3.5).

The proposed solutions to these features make use of an alternative `clone` methods (Section 3.2) to the original one described in Section 2.3.3. In the originally proposed method, Boolean values are passed as individual arguments to the method, whereas the alternative `clone` method have a `List<Boolean>` as one argument.

3.1 Desired Cloning Properties

Several desired properties of cloning any object, o:

1. All objects that are reachable inside o are also cloned (in accordance to sheep cloning).
2. All objects outside of o must not be cloned (in accordance to sheep cloning).
3. The heap after cloning o is homomorphic to the heap before cloning o (i.e. the clone has the same shape as o).
4. Minimise the amount of dynamic information that is needed to be stored (cost for this method of cloning).

Properties 1 and 2 must hold for producing sheep clones, however, examples of cloning in *Trust the Clones* only had situations where owners dominate objects inside them (this property, owners as dominators, is detailed in Section 2.2.1). This led to some investigation in cloning objects where owners as dominators do not hold. I discovered that a subset of these situations result in type errors when cloning according to properties 1 and 2. These problematic situations are where for objects $o1$, $o2$, $o3$, there is a path (via fields) from $o1$ to $o3$; there is a path from $o3$ to $o2$; $o3$ is outside of $o1$; and $o2$ is inside of $o1$. This problem is discussed in detail in Section 3.1.1, where we also explore possible solutions.

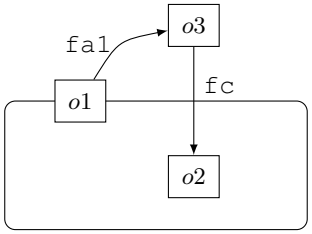
3.1.1 Cloning without Owners-as-Dominators

When cloning an object, $o1$, other objects that are cloned as part of the process include all objects that are in the cloning domain of $o1$. This raises the question of whether it is desirable to clone an object, $o2$, whose owner is $o1$, but some object $o3$ outside of $o1$, references $o2$ (i.e. owners as dominators do not hold).

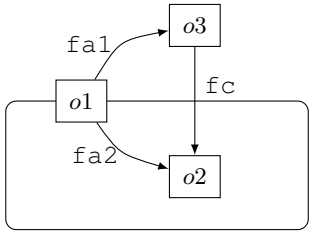
In this section, we discuss the possible results of cloning without owners-as-dominators, their problems (or lack of problem) and possible solutions. We divide situations that break owners as dominators into 2 cases based on object references, for any objects $o1$, $o2$ inside $o1$, and $o3$ outside of $o1$:

1. *Problematic case* – there is a path from $o1$ to $o3$, and there is a path from $o3$ to an $o2$. We call such a path from $o1$ to $o2$ via $o3$ as a *re-entering domain path* (RDP). Figure 3.1 illustrates 2 examples of this. In the following, we consider the possible cloning results, where I show the desired result of cloning $o1$ will have type error.

2. *Non-problematic case* – there is no re-entering ownership path but $o3$ has a field reference to $o2$. We call such a reference as an *inward cross-boundary reference* (ICBR), corresponding to how such references appear in our diagrams. Figure 3.2 shows an example of this, where there is no re-entering domain path from $o1$, but $o3$ has an inward cross-boundary reference to $o2$.



(a) Path from $o1$ to $o2$ is not in $o1$'s cloning domain.



(b) There is a path from $o1$ to $o2$ in $o1$'s cloning domain and another path that is not in $o1$'s cloning domain.

Figure 3.1: Examples where there is a re-entering ownership path from $o1$ to $o2$ via $o3$, which is outside of $o1$.

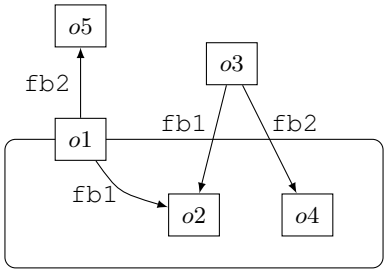


Figure 3.2: Example where there is no re-entering ownership path and there is an inward cross-boundary reference from $o3$ to $o2$ (and from $o3$ to $o4$).

For this method of generating clone code to be applicable to a larger range of programs, it is desirable to allow owners-as-dominators to be broken if there are only ICBR's and no RDP's. Such references and paths occur dynamically – some instances of classes may have ICBR's or RDP's depending on their actual owner parameters. To illustrate, Figure 3.3a shows objects where owners as dominators hold; whilst the property does not in Figure 3.3b, where the objects are instances of the same classes with different actual owner parameters in Listing 2.1. In Figure 3.3a, *oStudentList* has type `StudentList<world,world>` whereas *oStudentList* has type `StudentList<world,oCollege` in Figure 3.3b.

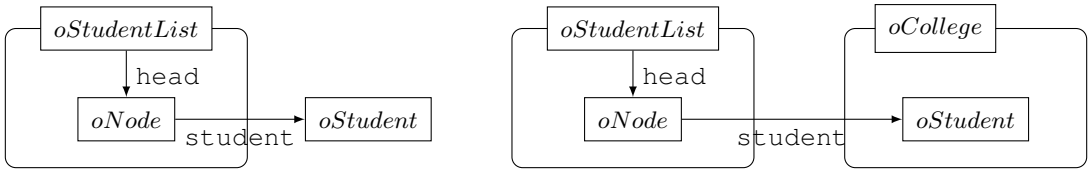
However, by property 4, we would like to remove ownership information from the generated Java code. Thus, the challenge is to find a solution that prevents RDP's through static checks. After showing the possible cloning results, the alternative solutions considered are discussed:

Solution 1 Prevent all such situations from occurring by enforcing owners as dominators (Section 3.1.2).

Solution 2 Allow the non-problematic cases but prevent the problematic case by checking the relative owners of objects along every possible path from each class (Section 3.1.3).

In the process of finding this solution, I made other attempts of detecting RDP's, which were insufficient. These are detailed in Appendix B for readers who are interested – otherwise the section may be skipped.

Under the first solution, even non-problematic cases cannot occur, we would prefer to allow owners-as-dominators to be broken so that the cloning approach is more applicable to real programs, such as those using iterators. Hence, solution 2 is more desirable since it prevents the possibility of problems occuring, while making this cloning approach applicable to more programs.



(a) Example of objects with owners as dominators.

(b) Example of objects where owners as dominators is broken.

Figure 3.3: Example of instances of classes from Listing 2.1, when owners as dominators hold and when it does not.

Cloning the Problematic Case

Under this case where there is an RDP from objects $o1$ to $o2$, we consider whether $o2$ should be cloned when $o1$ is cloned. The possible answer to this differs for the further subcases of where there is a RDP from $o1$ to $o2$, illustrated in Figure 3.1:

(a) *Only RDP's exist* – all paths from $o1$ to $o2$ are RDP's.

(b) *RDP's and internal paths from* $o1$ *to* $o2$ *exist* – there is a path, p, from $o1$ to $o2$ where all objects referenced in p are in the cloning domain (we call p an *internal path*) as well as another path that is a RDP.

However, in both situations, we show type errors occur for the desired cloning results regardless of whether internal paths exist because the problem is caused by the type of objects in the RDP. For readers who are interested, I also include how the two situations can be characterised formally in Appendix C – otherwise, readers may choose to skip this section.

Possible results of cloning $o1$ for our example in Figure 3.1a, when only RDP's exist:

- $o2$ *is cloned* – the result would be Figure 3.5, where $o2'$ (the clone of $o2$) is unreferenced and inaccessible. This is undesirable.

  $o2'$ is not referenced although $o1$ owns $o2$ because $o1$ can only reach $o2$ by paths that go outside of its ownership and objects that are not owned by $o1$ should not be cloned. This implies that all references by $o3$ will remain unchanged, (i.e. $o3$ continues to reference $o2$) and $o1'$ will reference $o3$.

- $o2$ *is not cloned* – the result would be Figure 3.4, where $o2$ is not cloned despite being inside the cloning domain of $o1$, and $o1'$ is the clone of $o1$. Similar to the previous subcase, $o3$ remains unchanged and $o1'$ references $o3$. This is the ideal outcome.

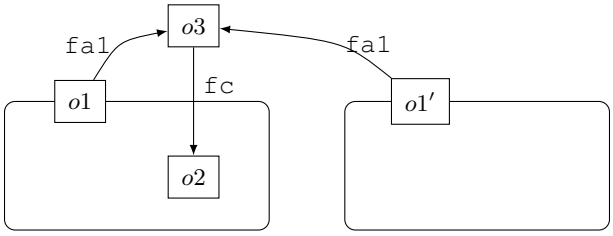


Figure 3.4: Possible result of cloning object $o1$ from Figure 3.1a.

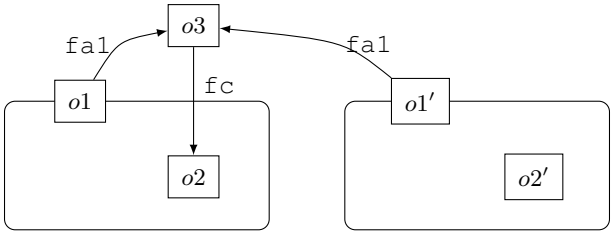


Figure 3.5: Undesirable result of cloning object $o1$ from Figure 3.1a.

Possible results of cloning $o1$ for our example in Figure 3.1b, when RDP's and internal paths exist:

- $o2$ *is cloned* – the result is Figure 3.6, where $o1'$ is the clone of $o1$, and references $o2'$ (clone of $o2$) and $o3$, which is not cloned. Since $o3$ is not cloned, it is left unchanged and continues to reference the original $o2$ object and has no knowledge of the clones. This is the ideal outcome in contrast to the situation with only RDP's because $o2$ is owned and referenced by $o1$ and so should be cloned by property 1.
- $o2$ *is not cloned* – the result is Figure 3.7, where $o2$ is not cloned and $o1'$ will have to reference the original $o2$ object. This is undesirable because we find a type error when we compare the type of field `fa2` with the type of $o2$ as seen in the diagram.

  Using the example class declarations in Listing 3.1 and Figure 3.7:

  $$o1 : \texttt{A}\langle \texttt{world} \rangle \qquad \text{(in Figure 3.7)}$$
  $$o1' : \texttt{A}\langle \texttt{world} \rangle \qquad \text{(in Figure 3.7)}$$

  Therefore, based on the class declaration, we expect:

  $$o1'.\texttt{fa2} : \texttt{B}\langle o1' \rangle \qquad \text{(by class declaration)}$$

  However, the type of $o1'.\texttt{fa2}$ as seen in the diagram does not match the expected type:

  $$o1'.\texttt{fa2} = o2 \qquad \text{(in Figure 3.7)}$$
  $$o2 : \texttt{B}\langle o1 \rangle \qquad \text{(in Figure 3.7)}$$
  $$\texttt{B}\langle o1' \rangle \neq \texttt{B}\langle o1 \rangle$$

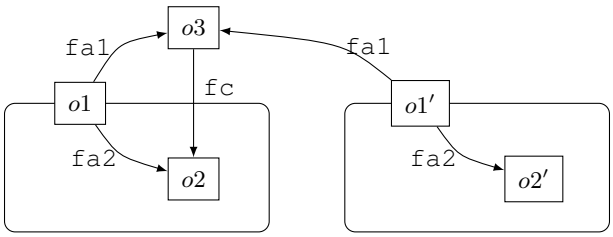


Figure 3.6: Possible result of cloning object $o1$ from Figure 3.1b.

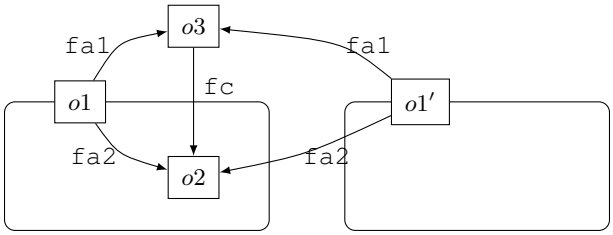


Figure 3.7: Undesirable result of cloning object $o1$ from Figure 3.1b.

```
1  class A<c> {
2      C<c,this> fa1;
3
4      B<this> fa2;
5  }
6
7  class B<c> {
8  }
9
10 class C<c1,c2> {
11     B<c2> fc;
12 }
```

Listing 3.1: Example class declarations for examples based on Figure 3.1.

Although, we would like to clone objects in the 2 subcases differently, there is a problem with achieving the desired results in both. We show there is a type error when we compare the type of field `fa1` with the type of $o3$ as seen in Figures 3.4 and 3.7. Using the example class declarations in Listing 3.1 for the ideal outcomes:

$$o1 : \texttt{A}\langle \texttt{world} \rangle \qquad \text{(in Figures 3.4 and 3.7)}$$
$$o1' : \texttt{A}\langle \texttt{world} \rangle \qquad \text{(in Figures 3.4 and 3.7)}$$

Therefore, based on the class declaration, we expect:

$$o1'.\texttt{fa1} : \texttt{C}\langle \texttt{world}, o1' \rangle \qquad \text{(by class declaration)}$$

However, the type of $o1'.\texttt{fa1}$ as seen in the diagram does not match:

$$o1'.\texttt{fa1} = o3 \qquad \text{(in Figures 3.4 and 3.7)}$$
$$o3 : \texttt{C}\langle \texttt{world}, o1 \rangle \qquad \text{(in Figures 3.4 and 3.7)}$$

This type error occurs in regardless of whether there is an internal path when there is a RDP from $o1$ to $o2$. Hence, we cannot clone clone situations where there are RDP's and need to prevented. Proposed solutions are to enforce owners-as-dominators (Section 3.1.2) and to statically prevent this problematic case only (Section 3.1.3).

Cloning the Non-problematic Case

For cases where there is no re-entering ownership path, when calling `clone()` on an object, $o1$: all objects that are outside the cloning domain of $o1$ remain unchanged by property 2; and all objects inside of and reachable from $o1$ are cloned by property 1. Hence, for the example in Figure 3.2, the result of cloning $o1$ is Figure 3.8, where $o3$, that has an inward cross-boundary reference to $o2$, will still reference $o2$, $o5$ is not cloned and $o2'$ is the clone of $o2$.

On the other hand, objects unreachable from $o1$, even if they are inside of $o1$, such as $o4$, will not be cloned. If $o4$ was cloned, the result of cloning $o1$ is shown in Figure 3.9, where $o4'$ is the clone of $o4$. However, $o4'$ is unreferenced and will be garbage collected by the Java Garbage Collector – the outcome is the same as when $o4$ was not cloned.

Compared to the problematic case, no type errors occur because for any object that is outside of and reachable from $o1$, `this` (in context of $o1$) or any object inside of $o1$ is not given as an owner parameter. Therefore, the type of fields that reference objects outside of the cloning domain of a class will be the same for the original object and the clone. This is illustrated explicitly by the corresponding class declarations in Listing 3.2 for the current example. $o1$ and $o1'$ are both of type `A<world>`, and $o5$ is of type `B<world>`. This matches with the expected type of the field `fa1` for $o1$ and $o1'$.

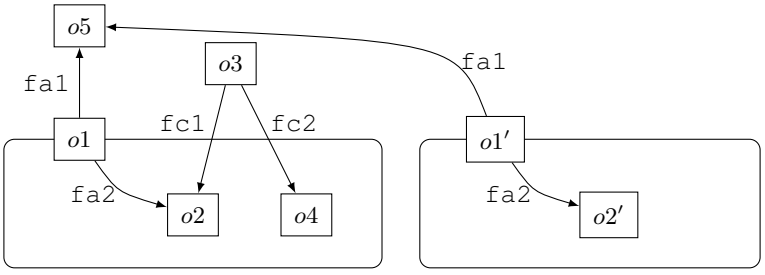


Figure 3.8: Possible result of cloning $o1$ from Figure 3.2.

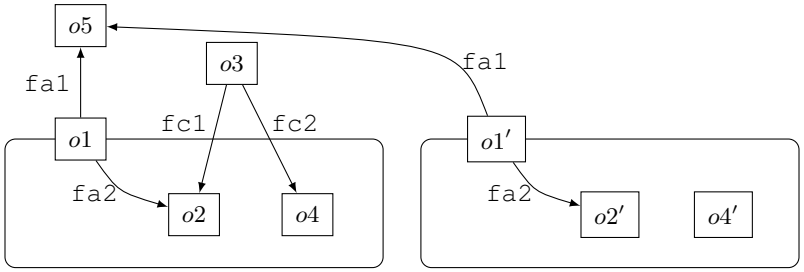


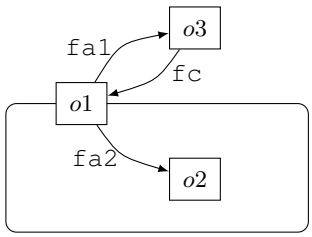
Figure 3.9: Undesired result of cloning $o1$ from Figure 3.2.

```
1  class A<c> {
2      B<c> fa1;
3
4      B<this> fa2;
5  }
6
7  class B<c> {
8  }
9
10 class C<c> {
11     B<c> fc1;
12     B<c> fc2;
13 }
```

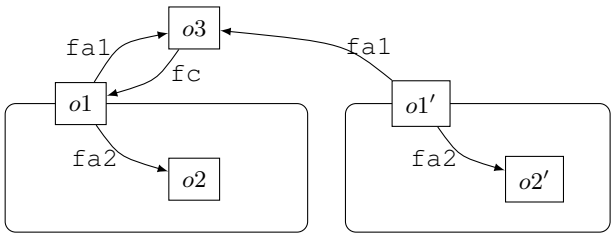
Listing 3.2: Example class declarations corresponding to Figures 3.2, 3.8 and 3.9.

3.1.2 Solution 1: Enforce Owners-as-Dominators

Situations presented in Figure 3.1 will not occur if we have owners as dominators because $o2$ should be dominated by its owner, $o1$, as described in Section 2.2.1. Therefore, $o3$ will not be able to have a direct reference to $o2$. To access $o2$, the object reference from $o3$ and $o2$ in Figure 3.1 will have to be replaced by a reference to $o1$ and add a reference from $o1$ to $o2$ (if it does not already exist), as shown in Figure 3.10a. The result of cloning $o1$ is shown in Figure 3.10b, where $o1'$ references $o3$.



(a) The examples in Figure 3.1 with owners as dominators enforced.



(b) Result of cloning $o1$ in Figure 3.10a.

Figure 3.10: Cloning example of Figure 3.1 with owners as dominators.

To enforce owners-as-dominators, it is sufficient to check the actual clone parameters at class instantiations, i.e. `C<`ca_1`,...,`ca_n`> o = new C<`ca_1`,...,`ca_n`>()`. From the syntax in Figure 2.9, the clone parameters of fields in a class `C`, can only be `this`, `world` or is one of `C`'s formal clone parameters. Therefore, the owner of each of the actual clone parameters `ca`$_2$`,...,ca`$_n$ must be a transitive owner of ca_1. Otherwise, the o may reference an object that is not dominated by its owner.

To illustrate, Figure 3.11 shows all the objects that can be referenced by o are the colour-filled rectangles. We notice that these colour-filled objects are all owned by o or a transitive owner of o, which are $\{\texttt{world}, o2, o1\}$ and are underlined in the diagram.

Operationally, we can statically check constructor calls that the owner of each actual clone parameter is in the set of transitive owners of the first actual clone parameter by checking the types. For example,

we can find the set of transitive owners of o by recursively checking the types of its owner:

$$\begin{aligned} & \text{start with empty set of transitive owners} = \{\} \\ o : <o2, ...> \quad & \text{transitive owners} = \{o2\} \cup \{\} \\ o2 : <o1, ...> \quad & \text{transitive owners} = \{o1\} \cup \{o2\} \\ o1 : <\texttt{world}, ...> \quad & \text{transitive owners} = \{\texttt{world}\} \cup \{o1, o2\} \end{aligned}$$

If owners-as-dominators is enforced, then the problematic situations cannot occur, but this also means that even the non-problematic subcases where owners-as-dominators are broken cannot occur either. If possible, we would like to allow owner-as-dominators to be broken if it does not cause problems so that objects like iterators (that break this property) can be used. This leds us on to Solution 2, which allows owners-as-dominators to be broken if the RDP's cannot possibly occur based on the class declarations.

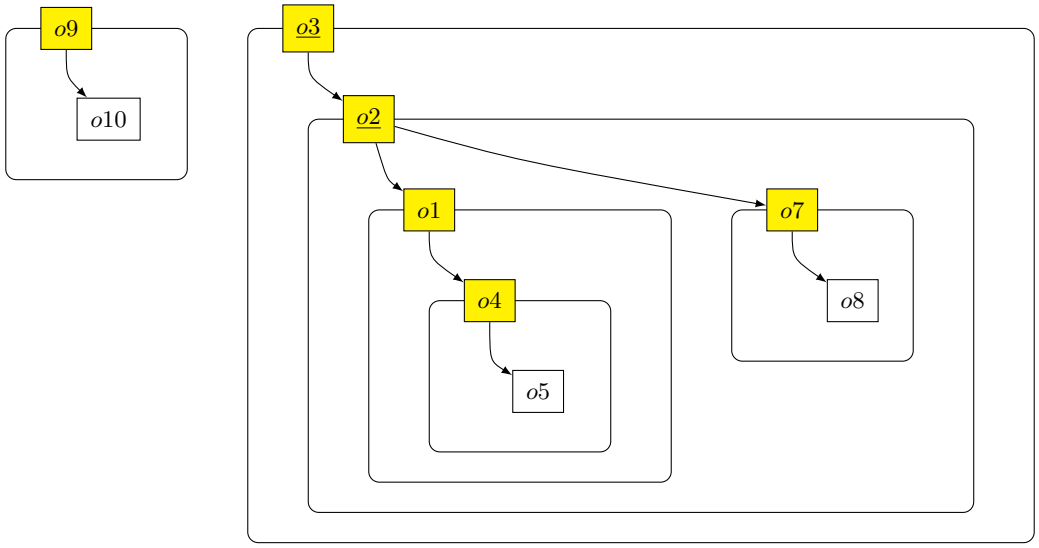


Figure 3.11: Example of objects that maybe referenced by o with owners-as-dominators.

3.1.3 Solution 2: Statically Prevent Problematic Case

This solution is to statically detect the *possibility* of RDP's occurring by checking class declarations and the relative owner positions of objects along all possible field paths. This involves traversing over fields to find all possible paths and for each path, p, determining whether each object along p is inside or outside (i.e. not inside) of the current class by substituting formal owner parameters with owner parameters relative to the current class.

All possible paths from a class, C, and the relative position of reachable objects can be represented by directed graphs with (extended) types as nodes, field references as edges and the root is the type of an object instance of C. Types are extended to allow us to use paths (relative to the root object) to express owners that are objects owned by the current object or by objects represented by formal owner parameters. These objects would otherwise be unrepresentable.

As an example, Figure 3.12 is the graph for the `College<c>` class declared in Listing 2.1. The root of the graph is `College<c>` and we treat the formal owner parameters, c, of this class as actual owner parameters. Paths to each node start from the root node (`this`) and append the label of edges as we traverse the graph to reach the node.

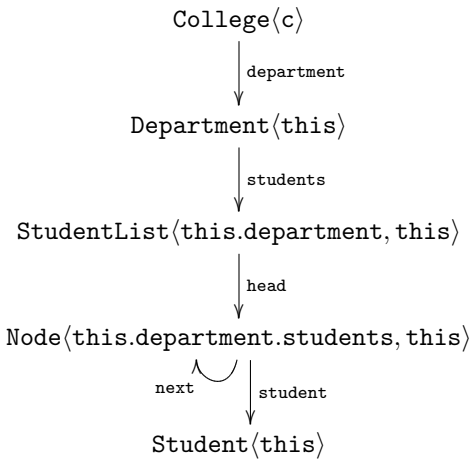


Figure 3.12: Graph showing all paths and reachable types from class `College<c>` where no RDP is possible.

The following observations enable us to identify whether a referenced object is inside or outside of the current class, given a class declaration `class C<`$\overline{c}$`>{...}`:

- Formal owner parameters, $\overline{c}$, are guaranteed to represent objects outside of `this` (the current instance of the class). This is because when calling the constructor, `new C<`$\overline{ca}$`>()`, the actual owner parameters, $\overline{ca}$, represent objects that already exist in the heap – existing objects cannot be owned or transitively owned by the new object that has not been created till the constructor call.

  However, this does not imply the current object is inside the objects represented by formal owner parameters, they may be any object that is not inside the current object.

- By the extended Java syntax, owner positions of fields must be from $\{$`this`, `world`, $\overline{c}\}$. Where `this` means the referenced object is inside the current class and `world` means the referenced object is outside the current class.

Hence, if the owner of a referenced object, o is:

- `this` then o is inside the current object. For example, the object referenced by `this.department` in Figure 3.12 is inside the current object.
- `world` then o is outside the current object.
- A formal owner parameter then o is outside the current object.
- A path, p, then o is inside the current object *iff* the object referenced at p is inside the current object. For example, the object referenced by `this.department.students` is owned by `this.department` and hence is inside the current object since the owner of `this.department` is `this`.

By inspecting the owner of each node as we traverse the edges, we build a sequence of owners. We shall call such a sequence as a *relative position path* (RPP), which gives the relative position of each field referenced object in a path, p. For any object, o, at the nth referenced field of p, the owner of o is the nth element in corresponding RPP. For example, the field path, $p =$ `this.department.students.head.student`, has RPP, `this.(this.department).(this.department.students).this`, which comes from the first owner of each field referenced object.

Based on the observations mentioned, we know that a RPP corresponds to a re-entering domain path *iff* 1 of the following holds for a RPP:

- `this` follows a formal owner parameter (from $\overline{c}$) of the current class. The formal owner parameter means that the field path has reached an object, o, outside of the cloning domain of the current

class. If the next relative owner position in the RPP is `this`, then o may reference an object that is inside the current class. For the previous example of RPP, `this.this.c2`, we can see that this does not correspond to a RDP.

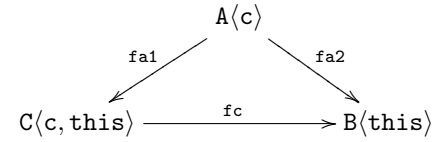
- `this` follows a path, p, where $p \neq$ `this` and $p \not\prec^*$ `this` (i.e. the object referenced by p is outside of `this`).
- A path, p_2 follows a path, p_1, where $p_1 \not\prec^*$ `this` and $p_1 \prec^*$ `this`

From the graph in Figure 3.12, we know that all referenced objects are inside the current object by transitivity of the inside relation ($\prec^*$):

$$\texttt{this.department} \prec^* \texttt{this}$$
$$\texttt{this.department.students} \prec^* \texttt{this.department}$$
$$\texttt{this.department.students.head} \prec^* \texttt{this.department.students}$$
$$\texttt{this.department.students.head.next} \prec^* \texttt{this.department.students}$$
$$\texttt{this.department.students.head.student} \prec^* \texttt{this}$$

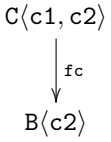
Hence, there are no RDP's from a `College` object because at each step of all possible paths, the reference object is inside of the current object. If there is an RDP, p, then there is are referenced objects, $o1$ and $o2$, where $o2$ follows $o1$ in p, $o1$ is outside of the current object and $o2$ is inside the current object.

Example where RDP exists. For the class declarations in Listing 3.1, which we have previously used to show that RDP's may occur, the graph for each class is given in Figure 3.13. The path, `this.fa1.fc`, from an `A<c>` object is a RDP with RPP, `c.this`. This is because the first field referenced object is owned by `c` (i.e.`this.fa1` $\not\prec^*$ `this`) and it references an object owned by `this` (i.e. `this.fa1.fc` $\prec^*$ `this`). However, RDP's cannot occur for the other classes because class `B` has no fields.



(a) Graph for class `A<c>`, where there is a RDP.

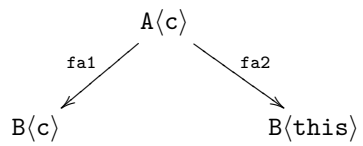
(b) Graph for class `B<c>`, where there is no RDP.



(c) Graph for class `C<c1,c2>`, where there is no RDP.

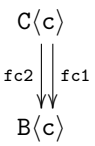
Figure 3.13: Graph showing all paths and reachable types from classes declared in Listing 3.1, where RDP's are possible from `A` objects.

Example where ICBR exists but there is no RDP. For the class declarations in Listing 3.2, previously used to show that RDP's cannot occur but OAD property is broken by the possibility of ICBR's, the graph for each class is given in Figure 3.14. The graph shows that no RDP's are possible since class `B` has no fields, from nodes with an owner



(a) Graph for class `A<c>`, where there is no RDP.

(b) Graph for class `B<c>`, where there is no RDP.



(c) Graph for class `C<c>`, where there is no RDP.

Figure 3.14: Graphs showing all paths and reachable types from classes based on Listing 3.2, where there is no possibility of RDP's.

Finding the type of referenced objects systematically. I show how the type of objects in these graphs are calculated systematically through a simple example based on the class declarations in Listing

2.1. The graph for class StudentList<c1,c2> is Figure 3.16 and we show the steps in calculating the types of objects along the field path, p = this.head.next.student, in Figure 3.15.

First, we know that the formal parameters to the StudentList class is <c1,c2> and are also treated as the actual owner parameters. At each step, we find the class, c, and the actual owner parameters, $\overline{ca}$, of the object referenced by the *Field Path* column. To access the next field in the path, we will go into the class declaration of c, so the formal parameters of c need to be substituted with $\overline{ca}$ and in the class declaration of c, references to this as an owner parameters need to be substituted with the path in *Field Path*. This propagates the actual owner parameters to referenced objects so that types of fields use this substitution to find their owner position relative to this object. The following gives a brief explanation of each step to further illustrate:

Step 0 We know that the formal parameters to the StudentList class is <c1,c2>, which we treat as actual owner parameters and c1 is the owner. Hence, the substitution to use when calculating the type of a field in this class is [this/this, c1/c1, c2/c2].

Step 1 Field head in StudentList is declared as Node<this,c2> head;. After applying the substitution from Step 0, the type of the object at this.head is Node<this,c2>, where the owner is this. To calculate the actual parameters of the next field reference, the formal owner parameters of class Node are substituted by the actual parameters. Also, fields in class Node that use this as a owner parameter actually refer to the object at this.head from StudentList. This substitution is given in the last column.

Step 2 The substitution from Step 1 is applied to the field declaration of field next in class Node. Hence, the actual owner parameters of the referenced object is <this,c2> and substitutes this and the formal parameters of the class of next.

Step 3 The substitution from Step 2 is applied to the field declaration of field student in class Node. Hence, the actual owner parameters is <c2>. We can further calculated the substitution to be applied when we go into the class declaration of Student.

Step	Field Path	Type	Owner Parameter Substitution for Class of Field
0	this	StudentList<c1,c2>	[this/this, c1/c1, c2/c2]
1	this.head	Node<this,c2>	[this.head/this, this/c1, c2/c2]
2	this.head.next	Node<this,c2>	[this.head.next/this, this/c1, c2/c2]
3	this.head.next.student	Student<c2>	[this.head.next.student/this, c2/c]

Figure 3.15: Steps to finding the types of referenced objects in the path this.head.next.student from StudentList<c1,c2>.

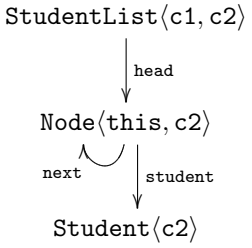


Figure 3.16: Type graph showing reachable types from StudentList<c1,c2>.

Formalisation

Having described the solution, the formalisation is given in here, including syntax of extended types, definition of path graphs, and origin graphs . This formalisation is based on the graphs showing reachable types, however in the process of finding this formalisation, another formalisation was made and this is included in Appendix A for readers who are interested.

Programs can be represented as a mapping from class names to a tuple, consisting of: the formal owner parameters, a mapping from field names to field types, and a mapping from method identifiers to the method declaration. This allows us to defined functions: F, Fs and O, to find the type of a field in a class; all fields of a class, and formal owner parameters of a class declaration, respectively. The definition of these are given in Figure 3.17.

$$Program, P = ClassId \to (\overline{c} \times (FieldId \to type) \times (MethId \to meth))$$

$$\begin{aligned} F(P, C, f) &= P(C) \downarrow_2 (f) && \text{Gets the type of the field } f \text{ in class } C \\ Fs(P, C) &= \{f | F(P, C, f) \text{ is defined}\} && \text{Gets all fields of class } C \\ O(P, C) &= P(C) \downarrow_1 && \text{Gets the formal owner parameters} \end{aligned}$$

Figure 3.17: Representation of programs and auxiliary functions.

Figure 3.18 gives the syntax of extended types as used in our graphs, where owner parameters are actual owner parameters or field paths. The information given by the graphs (relative to a class, C) shown so far is formalised as a pair, (PG, OG), where PG is a path graph and OG is an origin graph. PG holds the information about edges, $PG(t)(f)$ gives the extended type of the object referenced by field f from object of type t. OG(t) returns the field path from C to the node representing type t. The type of the partial function and map are given in Figure 3.19, where an auxiliary function, *inside* is also defined. Given an origin graph and a parameter (po) of an extended type, *inside* returns true if po represents an object inside of the root object in the represented graph.

$$\begin{aligned} EType &::= ClassId\langle\overline{por}\rangle && \text{Extended types} \\ PathOrOwner, po &::= p | ca && \text{Field path or actual owner parameter} \end{aligned}$$

Figure 3.18: Syntax for extended types.

$$\begin{aligned} PG &: EType \to (FieldId \to EType) && \text{Path Graph} \\ OG &: EType \to (Path) && \text{Original Graph} \end{aligned}$$

$$inside : \ OriginGraph \times PathOrOwner \to Boolean$$

$$inside(OG, po) = \begin{cases} true & po = \texttt{this} \\ false & po = \texttt{world} \\ false & po = ca \\ inside(OG, po_1) & \text{otherwise} \\ \ \text{where } OG(C\langle po_1, ..., po_n\rangle) = po & \end{cases}$$

Figure 3.19: Types for path graphs and origin paths; and definition of an auxiliary function to determine whether the owner parameter of an extended type is inside the root object.

The pair (PG, OG) is *complete* for a class C in program P *iff* the following conditions hold:

- The current class is in the graph, i.e.
  $C\langle c_1, ..., c_n\rangle \in dom(PG)$ where $C \in dom(P)$ and $O(P, C) = c_1, ..., c_n$
- The current class has edges for each field declared in its class declaration, i.e.
  $Fs(P, C) = dom(PG(C\langle c_1, ..., c_n\rangle))$
- The current class is the root of the graph, i.e.
  $OG(C\langle c_1, ..., c_n\rangle) = \texttt{this}$
- All reachable objects from the fields of the current class are also in the graph, i.e.
  For any $D \in dom(P)$ and any field, f:

$$\begin{aligned} &D\langle d_1, ...d_n\rangle \in \bigcup\{range(fToET) \mid fToET \in range(PG)\} \text{ and } F(P, D, f) = t \\ &\implies \\ &PG(D\langle d_1, ..., d_n\rangle) = t' \text{ and } OG(t') = OG(D\langle d_1, ..., d_n\rangle).f \\ &\text{where } t' = t[d_1, ..., dn/O(P, D), OG(D\langle d_1, ..., d_n\rangle)/\texttt{this}] \end{aligned}$$

If (PG, OG) is *complete* for class C in program P, then a C object may have RDP's *iff*:

$$\begin{aligned} \exists D\langle d_1, ..., d_n\rangle, E\langle e_1, ...e_n\rangle, f: \; & PG(D\langle d_1, ..., d_n\rangle)(f) = E\langle e_1, ...e_n\rangle \\ & \text{and} \\ & D\langle d_1, ...d_n\rangle \neq C\langle O(P, C)\rangle \\ & \text{and} \\ & \neg inside(OG, d_1) \text{ and} inside(OG, e_1) \end{aligned}$$

This will detect whether there is an edge, f, from $D\langle d_1, ..., d_n\rangle$ to $E\langle e_1, ...e_n\rangle$ in the graph for C, where the D object is not the current object and is not inside the current object, and the E object is inside the current object (creating a RDP).

To demonstrate, I show that the RDP is detected for the class declarations in Listing 3.1, where we have shown there may be a RDP from `A` (Figure 3.1 contains example objects where there is an RDP and the extended type graph of `A` is Figure 3.13a). The pair (PG, OG) is complete for `A`, where:

$$\begin{aligned} PG = [&\texttt{A}\langle\texttt{c}\rangle \mapsto [\texttt{fa1} \mapsto \texttt{C}\langle\texttt{c}, \texttt{this}\rangle, \texttt{fa2} \mapsto \texttt{B}\langle\texttt{this}\rangle], \\ &\texttt{B}\langle\texttt{this}\rangle \mapsto [], \\ &\texttt{C}\langle\texttt{c}, \texttt{this}\rangle \mapsto [\texttt{fc} \mapsto \texttt{B}\langle\texttt{this}\rangle]] \\ OG = [&\texttt{A}\langle\texttt{c}\rangle \mapsto \texttt{this}, \\ &\texttt{B}\langle\texttt{this}\rangle \mapsto \texttt{this.fa2}, \\ &\texttt{C}\langle\texttt{c}, \texttt{this}\rangle \mapsto \texttt{this.fa1}] \end{aligned}$$

Then the possibility of an RDP is detected because the above condition holds:

$$\begin{aligned} PG(\texttt{C}\langle\texttt{c}, \texttt{this}\rangle)(\texttt{fc}) &= \texttt{B}\langle\texttt{this}\rangle \\ \texttt{C}\langle\texttt{c}, \texttt{this}\rangle &\neq A\langle\texttt{c}\rangle \\ \neg inside(OG,&\texttt{c}) \\ inside(OG,&\texttt{this}) \end{aligned}$$

3.2 Alternative Cloning Methods

In this section, variations of the proposed cloning methods (Section 2.3.3) are presented here as alternatives and the advantage of these alternatives is the parametric clone method can be called for objects whose dynamic type (and number of cloning parameters) may vary during execution. These alternative cloning methods are adopted in my proposed solutions to extending ownership directed cloning for arrays, subclassing and generics.

3.2.1 Use a Single List Argument for Boolean Values in Clone Methods

The proposed cloning method described in Section 2.3.3 expects Boolean values and a `Map` as arguments. An alternative is to use a single `List<Boolean>` object as an argument rather than individual Boolean arguments, where the order of values in the list is the same as the order of the original sequence of Boolean arguments. This alternative is used in the proposed cloning methods for arrays, subclassing and generics, where variable's dynamic type (and number of cloning parameters) is variant.

Listing 2.2 gives the general implementation of the alternative clone methods, where the differences with the original are:

Lines 2 - 6, 9: the parametric `clone` method takes a `List<Boolean>` object as the first argument.

Lines 17 - 29: a list of Boolean values to is constructed by getting the values from `bs` (line 38) and passed to the clone methods of the field objects.

As a concrete example, the cloning methods of class `Node` is shown in Listing 3.4.

```
1  C clone() {
2    List<Boolean> bs = new List<Boolean>();
3    bs.add(false_1);
4    ...
5    bs.add(false_n);
6    return this.clone(bs, newIdentitityHashMap());
7  }
8
9  C clone(List<Boolean> bs, Map m){
10   Object o = m.get(this);
11   if o != null then
12     return (C)o;
13   else {
14     C oClone = new C();
15     m.put(this, oClone);
16
17     List<Boolean> bs1 = new List<Boolean>();
18     bs1.add(s_{1,11});
19     ...
20     bs1.add(s_{1,1k});
21     oClone.f_1 = s_{1,1} ? this.f_1.clone(bs1, m) : this.f_1;
22
23     ...
24
25     bs1 = new List<Boolean>():
26     bs1.add(s_{q,q1});
27     ...
28     bs1.add(s_{q,qk});
29     oClone.f_n = bs.get(s_{q,q1}) ? this.f_n.clone(bs1, m) : this.f_n;
30     return oClone;
31   }
32 }
33
34 where
35   {f_1, ..., f_q} are the fields defined in class C
36 and where, for all i ∈ {1, ..., q} :
37   (fType(C⟨bs.get(0), ..., bs.get(n − 1)⟩, f_i))[true = this] = C_i⟨s_{i,1}, ..., s_{i,ik}⟩ (*@for some classes C_1, ...C_n
```

Listing 3.3: Alternative implementation of clone methods using `List<Boolean>`.

```
1  Node clone( ){
2    List<Boolean> bs = new List<Boolean>();
3    bs.add(false);
4    bas.add(false);
5    this.clone(bs, new IdentityHashMap());
6  }
7
8  Node clone(List<Boolean> bs, Map m){
9    Object n = m.get(this);
10   if (n != null) {
11     return (Node)n;
12   } else {
13     Node clone = new Node();
14     m.put(this,clone);
15
16     List<Boolean> bs1 = new Liost<Boolean>();
17     bs1.add(bs.get(0));
18     bs1.add(bs.get(1));
19     clone.next= bs.get(0) ? this.next.clone(bs1,m) : this.next;
20     bs1 = new List<Boolean>();
```

```
21      bs1.add(bs.get(1));
22      clone.student= bs.get(1) ? this.student.clone(bs1,m) : this.student;
23
24      return clone;
25    }
26  }
```

Listing 3.4: Alternative clone methods for class Node.

3.2.2 Generate Permutation-like Order of Cloning Parameters for Fields

This approach to generating cloning methods extends the cloning alternative described in the previous section by generating a list of indices for each field, in standard Java. We shall call these lists as *permutation-like order lists* as they are lists of indices (indexing from 0) into the formal cloning parameters of the class declaration that gives the sequence of actual parameters. Actual cloning parameters `this` and `world` (which are not formal cloning parameters) are represented in permutation-like order lists by -1 and -2, respectively. For example, if a field's permutation-like list is [1,2,0,-1], this means that the field has 4 actual cloning parameters, where:

- The first actual cloning parameter is the second (1) formal cloning parameter of the class declaration that the field is declared in.
- The second actual cloning parameter is the third (2) formal cloning parameter.
- The third actual cloning parameter is the first (0) formal cloning parameter.
- The fourth actual cloning parameter is `this` (-1).

Hence, the permutation-like order also gives the indices of the Boolean values in the arguments of the class' clone method that should be passed to the field's clone method. This alternative is used in the proposed cloning methods for subclassing and generics, where fields' dynamic type (and number of cloning parameters) may vary during execution.

This permutation-like order is useful when the actual cloning parameters of a field's dynamic type is not known statically. Hence, this is used in proposed solutions to extending cloning for subclassing, where subclasses may have different actual cloning parameters, and for generic classes, where there are generic type parameters with unknown actual cloning parameters.

Cloning using Permutation-like Order Lists

Given a permutation-like order list, *perm*, and a list of Boolean values (from the parametric clone method's arguments) it is possible to obtain the list of boolean values to pass to the field that *perm* is associated with. A helper method, `reorder`, in Listing 3.5 returns a list of Boolean values to pass to the clone method of the field that is associated with the given `perm` argument. The function visits each index in `perm` in order (lines 9 - 12) and determines what the Boolean value the index refers to on line 10. If the index is:

- -1 then the actual cloning parameter is `this`, so `true` must be the Boolean value (objects in the cloning domain of `this` should be cloned).
- -2 then the actual cloning parameter is `world`, so `false` must be the Boolean value (objects owned by `world` are not in the cloning domain of any object).
- Otherwise the index refers to a Boolean value in `bs`.

Constructing and Assigning Permutation-like Order Lists

Whenever a reference type variable is declared (`A<c> a;`), an associated permutation-like order list is also declared (`A a; List<Integer> aPerm;`) in the standard Java code generated. For every assignment statement to local variables and fields of the current class (`x = y;`) written in the extended language, an assignment is also made to the permutation-like order list (`x = y; xPerm = yPerm;`) in the code generated.

However, care needs to be taken when assigning to fields in objects that are not `this` and when assigning values from fields of objects that are not `this`. This is because indices in permutation-like order lists are in terms of the object that they are stored in, when there are assignments to fields of other objects (e.g.

`this.f1.f1 = x;`) the permutation-like order (`xPerm`) needs to be calculated relative to the cloning parameters of `this.f1`. Helper methods, `getRelativePerm` and `getReverseRelativePerm`, have been defined to do this (Listing 3.5). The purpose of `getRelativePerm` is to calculate the permutation-like order list to assign to another object's field. The purpose of `getReverseRelativePerm` is to calculate the permutation-like order list relative to some other object and find the the permutation-like order list relative to the current object.

getRelativePerm. Given two permutation-like order lists representing cloning parameter indices for objects, $o1$ and $o2$, that are in terms of cloning parameters of the same object, $o3$, `getRelativePerm` returns a new permutation-like order list that represents the cloning parameter indices for $o2$ in terms of $o1$. This is used when assigning $o2$ to a field, f, of $o1$ so that $o1$ can calculate the Boolean values to pass to the f's clone method. This method visits each index in `fieldPerm` (lines 26 - 32) and finds where it appears in `newBasePerm`) because the same index represents the same object since bother arguments are relative to the same object. If the index is:

- -2 then the actual cloning parameter is `world`, which is the same for all objects, so -2 must be the index in `relativePerm`.
- Otherwise the index must occur in `newBasePerm` and we can find the index at which it occurs.

For example, if $o1$ has permutation-like order list, $p1 = [1, 0]$, and $o2$ has permutation-like order list, $p2 = [0]$, relative to the current object that has formal owner parameters $\langle c1, c2 \rangle$, then 0 refers to $c1$ and 1 refers to $c2$. If $o2$ was to be assigned to a field, f, of $o1$, then $p2$ has to be in terms of $o1$'s cloning parameters. Since $o2$ only has 1 cloning parameter (referring to $c1$), the permutation-like order list for f in $o1$ must also refer to $c2$. Hence, the permutation-like order list stored in $o1$ for f is [1] (the second cloning parameter of $o1$ refers to $c2$), this is the index of 0 in $p1$.

getReverseRelativePerm. The two method arguments are permutation-like order lists representing cloning parameter indices for objects, $o1$ and $o2$, where `fieldPerm` is the list indexing into the cloning parameters of $o1$ and `objPerm` is the list indexing into the cloning parameters of some other object $o3$. `getReverseRelativePerm` returns a new permutation-like order list that represents the cloning parameter indices for $o2$ in terms of $o3$. This is used when in the context of $o3$, there are uses of objects ($o2$) referenced by another object ($o1$). This method visits each index in `fieldPerm` (lines 49 - 55) and finds the value it indexes in `objPerm`). If the index is:

- -2 then the actual cloning parameter is `world`, which is the same for all objects, so -2 must be the index in `relativePerm`.
- Otherwise the index must be 0 or greater because $o3$ cannot access $o2$ unless it can name all of $o2$'s actual cloning parameters (i.e. they must refer to either `world`, $o3$ or one of $o3$'s formal cloning parameters). Hence, we can find the value in `objPerm` at the given index.

For example, if $o1$ has permutation-like order list, $p1 = [1, 0]$,relative to $o3$; $o2$ has permutation-like order list, $p2 = [0]$, relative to $o1$; and $o3$ that has formal owner parameters $\langle c1, c2 \rangle$; then $o2$ has only 1 cloning parameter that refers to the first cloning parameter of $o1$ ($c2$). If $o2$ was to be assigned to a field, f, of $o3$, then $p2$ has to be in terms of $o3$'s cloning parameters. Since $o2$ only has 1 cloning parameter (referring to $c2$), the permutation-like order list for f in $o3$ must also refer to $c2$. Hence, the permutation-like order list stored in $o1$ for f is [1] (the second cloning parameter of $o1$ refers to $c2$), this is the value at the 0th index of $p1$.

Full example. To illustrate using a concrete example, the class in Listing 3.6, which has a method with some assignment statements to variables and fields, will have the generated Java code is in Listing 3.7. Each reference type variable and field has its own permutation-like order list; the actual permutation-like order list is actually created when a constructor is called (Listing 3.7 lines 11 - 14 are generated for Listing 3.6 line 8). The generated clone method will call the helper methods before deciding whether to clone each field to get the list of Boolean values corresponding to the field (for example line 49). The field containing the permutation-like order list is also given to the clone on line 49 so that `oClone` knows how to clone its fields.

For assignments to fields of objects that are not `this` (Listing 3.6 lines 14 and 16), `getRelativePerm` to assign the permutation-like order list associated with the field (Listing 3.7 lines 26 and 29). When

field of other objects are used, such as in Listing 3.6 line 16, `getReverseRelativePerm` is also used to first get the permutation-like order list of `y.f1` in terms of cloning parameters of class `C`.

```
1  class Perm {
2    /*
3     * Gets the a new list that is bs reordered according to perm.
4     * Used when calling clone on a field that has a permutation-like order list, perm.
5     */
6    static List<Boolean> reorder(List<Boolean> bs, List<Integer> perm) {
7      List<Boolean> reordered = new List<Boolean>();
8
9      for(Integer index : perm) {
10       Boolean b = index == -1 ? true : (index == -2 ? false : bs.get(index));
11       reordered.add(b);
12     }
13
14     return reordered;
15   }
16
17   /*
18    * Returns a permutation-like order list representing fieldPerm relative to newBasePerm.
19    * Used when assigning to field of objects that has a permutation-like
20    * order list, newBasePerm.
21    * newBasePerm and fieldPerm are in terms of the current object, this.
22    */
23   static List<Integer> getRelativePerm(List<Integer> newBasePerm, List<Integer> fieldPerm)
         {
24     List<Integer> relativePerm = new List<Integer>();
25
26     for(Integer index : fieldPerm) {
27       if (index== -2) {
28         relativePerm.add(-2);
29       } else {
30         relativePerm.add(newBasePerm.indexOf(index));
31       }
32     }
33
34     return relativePerm;
35   }
36
37   /*
38    * Returns a permutation-like order list representing fieldPerm relative to the
39    * same object that objPerm is relative to.
40    * Used when assigning to local variables/fields with
41    * with a field of some other object that has a permutation-like order likst, objPerm.
42    * objPerm is relative to the current object.
43    * fieldPerm the permutation-likfe order list of the object relative to the
44    * object with objPerm.
45    */
46   static List<Integer> getReverseRelativePerm(List<Integer> objPerm, List<Integer>
         fieldPerm) {
47     List<Integer> relativePerm = new List<Integer>();
48
49     for(Integer index : fieldPerm) {
50       if (index== -2) {
51         relativePerm.add(-2);
52       } else {
53         relativePerm.add(objPerm.get(index));
54       }
55     }
56
57     return relativePerm;
58   }
59 }
```

Listing 3.5: Helper methods for getting a list of Boolean values reordered according to the permutation-like order list and calculating the permutation-like order list to assign to fields of objects other than `this`.

```
1  class C<c1,c2,c3> {
2    A<this,c2> f1;
3
4    void m1() {
5      A<this,c2> x;
6      A<this,c2> y;
7
8      x = new A<this,c2>();
9
10     y = x;
11
12     this.f1 = y;
13
14     this.f1.f1 = x;
15
```

```
16      x.f1 = y.f1;
17    }
18  }
```

Listing 3.6: Example class declaration with assignments.

```
1  class C {
2    A f1;
3    List<Integer> f1Perm;
4
5    void m1() {
6      A x;
7      List<Integer> xPerm;
8      A y;
9      List<Integer>yPerm;
10
11     x = new A();
12     xPerm = new List<Integer>();
13     xPerm.add(-1);
14     xPerm.add(1);
15
16     y = x;
17     yPerm = xPerm;
18
19     this.f1 = y;
20     this.f1Perm = yPerm;
21
22     this.f1.f1 = x;
23     this.f1.f1Perm = Perm.getRelativePerm(this.f1Perm, xPerm);
24
25     y.f1 = x;
26     y.f1Perm = Perm.getRelativePerm(yPerm, xPerm);
27
28     x.f1 = y.f1;
29     x.f1Perm = Pern.geRelativePerm(xPerm, Perm.getReverseRelativePerm(yPerm, y.f1Perm));
30   }
31
32   C clone() {
33     List<Boolean> bs = new List<Boolean>();
34     bs.add(false);
35     bs.add(false);
36     bs.add(false);
37     return this.clone()
38   }
39
40   C clone(List<Boolean> bs,Map m) {
41     Object o = m.get(this);
42     if (o != null) {
43       return (C)o;
44     } else {
45       C oClone = new C();
46       m.put(this,oClone);
47
48       if (this.oClone != null) {
49         List<Boolean> fBs = Perm.reorder(bs,this.f1Perm);
50         oClone.f1 = fBs.get(0) ? this.f1.clone(fBs, m) : this.f1;
51         oClone.f1Perm = this.f1Perm;
52       }
53
54       return oClone;
55     }
56   }
57 }
```

Listing 3.7: Generated class that uses permutation-like order lists, in standard Java, for the class in Listing 3.6.

3.3 Arrays

Arrays are commonly used in programs so we would like to extend cloning using ownership types to deal with arrays of objects. An array is an object and elements of an array are also objects of a class, which will have their own cloning parameters in their class declaration. The implication is that elements of an array can have different cloning parameters to the array object. Figure 3.20 shows an example of such an array, oAs, whose owner is o and elements of the array ($oA1$, ..., oAn) are not owned by o, i.e. they have different cloning parameters.

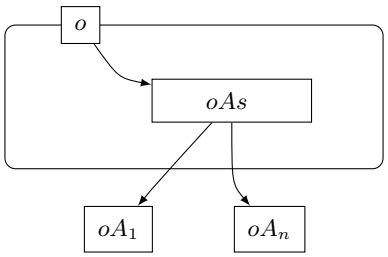


Figure 3.20: Example of an array object with ownership types.

There are several considerations when extending the cloning approach to arrays mainly because the array class is predefined in Java and may be any dimension (i.e. arrays of arrays). We would prefer not to make changes to the predefined array class, this implies we cannot call the `clone` method for array objects. Therefore, there should be a different approach to implementing array clone methods and clone methods of arrays need to know whether the array elements are array objects since array objects will be cloned differently.

Extending the system will involve:

- Extending the syntax to have arrays with cloning parameters (Section 3.3.1).
- Generating clone methods for arrays (Section 3.3.2), this involves:
  - Solving the problem of how objects know what information is required to be passed to the cloning method of the array object, in order for the array object to call the array elements' clone methods and pass the correct cloning Boolean values.
  - Finding a solution to how/where the cloning method for arrays should be defined – we would prefer not to have to change the predefined array class.

3.3.1 Syntax

Array types are of the form `C<c1,...,cn>[]<o>`, corresponding to array types commonly used: `C[]`, which is an array of `C` objects. The type of the array elements are `C<c1,...,cn>` and the owner of the array is `o`. The syntax from Figure 2.9 is extended as shown in Figure 3.21, where the actual clone parameter after `[]` represents the owner of the array object and the array elements are of the type t, which appears before the rightmost `[]`.

Listing 3.8 shows some example class declarations in the extended syntax and Figure 3.22 shows an example `A` object and objects referenced by its fields. $o3$ is a multi-dimensional array (an array of arrays of `C<world,world>`), and $o4$ and $o5$ are arrays of `C<world,world>`.

We will call array elements that are not arrays as *array leaf elements*, so $o3$, $o4$ and $o5$ have the same leaf element type `C<world,world>`.

$$FieldType, t ::= C\langle\overline{ca}\rangle \mid t[]\langle ca\rangle$$

Figure 3.21: Extended syntax for arrays.

```
1  class A<c1,c2> {
2      B<c2>[]<c1> fbs;
3
4      C<c1,c2>[]<c1>[]<this> fcs;
5  }
6
7  class B<c> {
8  }
9
10 class C<c1,c2> {
11 }
```

Listing 3.8: Example class with arrays.

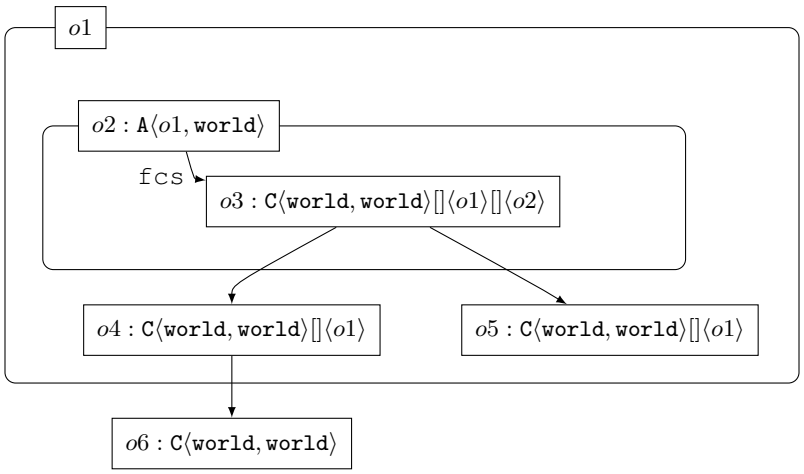


Figure 3.22: Example of an array object with ownership types.

3.3.2 Cloning Methods

Like any other object, cloning an array object, $o3$, involves determining whether referenced objects should also be cloned. For example, $o3$ in Figure 3.22 has type `C<world,world>[]<o1>[]<o2>`, so if $o3$ is cloned then its elements ($o4$ and $o5$), which are arrays of type `C<world,world>[]<o1>` and should be also be cloned if the domain of $o1$ is being cloned. In turn, elements of $o4$ and $o5$ ($o6$), which are of type `C<world,world>` should also be cloned if the domain of `world` is being cloned.

As highlighted from the example, cloning a n-dimensional array, o, of $C\langle\overline{ca}\rangle$ objects take into account the cloning parameter of each k-dimensional array (for $1 < k < n$) and the cloning parameters of the array leaf elements (i.e. $\overline{ca}$). Hence, the clone method for arrays need to know the dimension of the array; and the Boolean values indicating whether the cloning parameters each dimension array and array leaf element are in the cloning domain.

The proposed solutions makes use of the alternative style of generated clone method for classes (Section 3.2.1), where a lists of Boolean values is passed as a single argument rather than as individual Boolean arguments. The proposed alternatives for array clone methods are:

Solution 1 Generate a new class with a static clone method for arrays of any class objects and any dimension (Section 3.3.3).

Solution 2 Generate an array clone method for each class and any dimension (Section 3.3.4).

Solution 3 Generate an array clone method for each class and each dimension (Section 3.3.5).

For solution 2 and 3, the generated clone methods can be in a new class or added to the class that the array clone method is associated with.

The evaluation of the proposed solutions against the desired cloning properties are summarised in the following table. All solutions satisfy the desired cloning properties, however, there is more use of Java reflection in solution 1 than in solution 2,, whilst solution 3 does not require reflection at all. We would like to avoid using reflection if possible, so solution 3 is adopted.

Solution	Property 1	Property 2	Property 3	Property 4
1	No	Yes	Yes	No additional dynamic information required.
2	Yes	Yes	Yes	No additional dynamic information required.
3	Yes	Yes	Yes	No additional dynamic information required.

Figure 3.23: Table of solutions to arrays against desired cloning properties

3.3.3 Solution 1: Static Clone Method for All Arrays

This solution is to generate a static clone method in a new class, `Arrays`, for cloning any array (Listing 3.9), using Java reflection. This includes arrays of any class of objects and of any dimension.

To clone an array using this method, the array object itself needs to be passed as the `array` argument; the actual cloning parameters for the array elements' type as `bs`; and the `Map` of the cloned objects so far. The array object to be cloned must be passed to the clone method so that it can be cloned and a map is passed to the method so that we only clone objects once. The Boolean actual cloning parameters are passed as a `List` rather than as separate Boolean parameters because arrays of different class of objects may have different number of cloning parameters.

For multi-dimensional arrays, each array object has an owner and their Boolean actual cloning parameters must be included in the list of Boolean values. The order of the Boolean values should be according to depth of the array element. For example, `C<o3,o4>[]<o2>[]<o1> cs` is a multi-dimensional array owned by `o3` and the array elements are arrays (owned by `o2`) of `C<o1>` objects. When `cs` is cloned, the order of the list of Boolean values is `[o2,o3,o4]`. Boolean for `o1` is not included because it is the owner `cs` and the clone method is called only when the object requires cloning.

The main steps of the clone method are:

1. Check whether `array` has already been cloned. Return the clone if it already exists (lines 3 - 6).
2. Otherwise (`array` has not been cloned yet), create an array object as the clone (and add to the `m`) using Java reflection (lines 8 - 13).
3. If `array` is a multi-dimensional array:
   If the first element of `bs` is `true` then clone the elements of the array by recursion; otherwise reference the same object (lines 16 - 21). The first element of `bs` is removed because it is the owner of the elements of `array` and since each array element, a_e, is also an array. The array clone method does not expect the cloning parameter of a_e to also be passed because the clone method is called only when arrays requires cloning.
4. Otherwise, if `array` is an array with elements that are of primitive types, then the values of `array` are copied into the clone, `arrayClone` (lines 22 - 23).
5. Otherwise, we know `array` is a one-dimensional array of non-primitive types. If the first element of `bs` is `true`, we call the clone method defined in the array elements' class (lines 25 - 30). The first element of `bs` is not removed because the generated parametric clone methods in classes expect all cloning parameters of the class to be passed.

```
1  class Arrays {
2    static Object clone(Object array, List<Boolean> bs, Map m) {
3      Object n = m.get(array);
4      if (n != null) {
5        return n;
6      }
7
8      Class arrayClass = array.getClass();
9      int length = Array.getLength(array);
10     Class elementType = arrayClass.getComponentType();
11
12     Object arrayClone = Array.newInstance(elementType, length);
13     m.put(array, arrayClone);
14
15     if (elementType.isArray()) {
16       Boolean owner = bs.remove(0);
17       for(int i = 0; i < l; i++) {
18         Cloneable element = (Cloneable)Array.get(array, i);
19         Object elementClone = owner && element != null ? clone(element, bs, m) : element;
20         Array.set(arrayClone, i, elementClone);
21       }
22     } else if (elementType.isPrimitive()) {
23       System.arraycopy(array, 0, arrayClone, 0, length);
24     } else {
25       Boolean owner = bs.get(0);
26       for(int i = 0; i < l; i++) {
27         Cloneable element = (Cloneable)Array.get(array, i);
28         Object elementClone = owner && element != null ? element.clone(bs, m) : element;
29         Array.set(arrayClone, i, elementClone);
30       }
31     }
32
33     return arrayClone;
34   }
35 }
```

Listing 3.9: General array clone method.

This array clone method uses reflection to create an array clone and does not know the exact class of the array elements, therefore, it does not know the number of parameters that it will pass. Therefore, we change the arguments of clone methods of classes to `clone(List<Boolean> bs, Map m)`, so that we can pass `List<Boolean>` instead of the Boolean values as individual parameters.

However, the array clone method assigns elements of the array clone to an object, whose type is unknown (lines 26 - 28) and hence does not guarantee that a clone method exists for this object. To solve this, we can create an interface, Listing 3.10, that all classes implement and the clone methods of classes index into the list to get the Boolean values instead of using individual Boolean parameters directly.

```
1 interface Cloneable {
2   Cloneable clone();
3
4   Cloneable clone(List<Boolean> bs, Map m);
5 }
```

Listing 3.10: Interface for all classes.

An alternative to using `List<Boolean>` is to use variable arguments list for the Boolean argument. The class declaration will be in the form, `Cloneable clone(Map m, boolean ... bs) {...}`, where `bs` is treated as an array. This requires the clone method call on line 27 in Listing 3.9 to be changed to `element.clone(m,bs.toArray())` instead of `element.clone(bs,m)`.

As an example, Listing 3.11 gives the clone methods for `A` from Listing 3.8, where the array fields are cloned in lines 21 - 29 by the array clone method in Listing 3.9.

```
1 A clone() {
2   List<Boolean> bs = new List<Boolean>();
3   bs.add(false);
4   bs.add(false);
5   return this.clone(bs, newIdentitityHashMap());
6 }
7
8 A clone(List<Boolean> bs, Map m) {
9   Object o = m.get(this);
10  if (o != null) {
11    return(A)o;
12  } else {
13    A oClone = new A();
14    m.put(this, oClone);
15
16    List<Boolean> arrayBs = new List<Boolean>();
17    arrayBs.add(bs.get(2));
18    oClone.fbs = bs.get(0) && this.fbs != null ? Arrays.clone(this.fbs, arrayBs, m) : this
          .fbs;
19
20    arrayBs = new List<Boolean>();
21    arrayBs.add(bs.get(0));
22    arrayBs.add(bs.get(0));
23    arrayBs.add(bs.get(1));
24    oClone.fcs = this.fcs != null ? Arrays.clone(this.fcs, arrayBs, m) : this.fcs;
25
26    return oClone;
27  }
28 }
```

Listing 3.11: Clone methods for `A` from Listing 3.8.

3.3.4 Solution 2: Array Clone Method for Each Class and Any Dimension

This solution is to generate static array clone methods in every class (for any dimension) declared. The clone method is in the form `clone(Object array, List<Boolean> bs, Map m, int d)`, where `d` is the dimension of the array. This method is similar to the clone method in Solution 1 except the type of the array leaf elements of multi-dimensional arrays are known. Listing 3.12 is an example of this for cloning `Student` arrays of any dimension, where:

- Java reflection is not needed to check whether the object is a multi-dimensional array, since the dimension is given by `d`.
- Cloning the 1-dimensional array is lines 7 - 19, where reflection is not used since the type of array leaf elements is known.

- For multi-dimensional arrays of the current class, reflection is used to get the dimension of the array to create a clone (lines 21 - 25) and recursively calls this clone method for the elements of the array if the Boolean representing their owner is true (lines 28 - 33).

The array clone method is needed for every class although the second method will be very similar in each class. The only lines that will differ are lines 8 and 10 where the actual leaf element class is known.

However, multi-dimensional arrays for primitive types still need their own clone method, in a new class (Listing 3.13), which just needs to copy the values of array leaf elements if the 1-dimensional array is to be cloned. This is very similar to the clone method for classes except reflection is used to find the type of array leaf elements and for elements of 1-dimensional arrays, the values are copied (not cloned) since they are of primitive type.

```
1  static Object clone(Object array, List<Boolean> bs, Map m, int d) {
2    Object n = m.get(array);
3    if (n != null) {
4      return n;
5    }
6
7    if (d == 1) {
8      Student[] baseArray = (Student[]) array;
9      int length = baseArray.length;
10     Student[] arrayClone = new Student[length];
11     m.put(array, arrayClone);
12
13     Boolean owner = bs.get(0);
14     for(int i = 0; i < length; i++) {
15       arrayClone[i] = owner && baseArray[i] != null ? baseArray[i].clone(bs, m) :
             baseArray[i];
16     }
17
18     return arrayClone;
19   }
20
21   Class arrayClass = array.getClass();
22   int length = Array.getLength(array);
23   Class elementType = arrayClass.getComponentType();
24
25   Object arrayClone = Array.newInstance(elementType, length);
26   m.put(array, arrayClone);
27
28   Boolean owner = bs.remove(0);
29   for(int i = 0; i < length; i++) {
30     Cloneable element = (Cloneable)Array.get(array, i);
31     Object elementClone = owner && element != null ? clone(element, bs, m, d--) : element;
32     Array.set(arrayClone, i, elementClone);
33   }
34
35   return arrayClone;
36 }
```

Listing 3.12: Array clone method where the class of the leaf element is known.

```
1  class PrimitiveArrays {
2    static Object clone(Object array, List<Boolean> bs, Map m, int d) {
3      Object n = m.get(array);
4      if (n != null) {
5        return n;
6      }
7
8      Class arrayClass = array.getClass();
9      int length = Array.getLength(array);
10     Class elementType = arrayClass.getComponentType();
11
12     Object arrayClone = Array.newInstance(elementType, length);
13     m.put(array, arrayClone);
14
15     if (d == 1) {
16       System.arraycopy(array, 0, arrayClone, 0, length);
17
18       return arrayClone;
19     }
20
21     Boolean owner = bs.remove(0);
22     for(int i = 0; i < length; i++) {
23       Cloneable element = (Cloneable)Array.get(array, i);
24       Object elementClone = owner && element != null ? clone(element, bs, m, d--) :
             element;
25       Array.set(arrayClone, i, elementClone);
26     }
```

```
27
28      return arrayClone;
29    }
30  }
```

Listing 3.13: Array clone method where array leaf elements are of primitive types only.

3.3.5 Solution 3: Array Clone Method for Each Class and Each Array Dimension

This solution is to generate a static array clone method for each class and each array dimension that *is used/occurs* in the program. The method will be of the form `cloneC2(Object array, List<Boolean> bs, Map m)`, where `C` is the class of the leaf array elements and `2` is the dimension of the array. This method is similar to the clone method in Solution 2 except the exact dimensions and array leaf element types (primitive or reference types) are also known. Unlike Solution 1 and 2, this solution does not use Java reflection to find the dimension of the array or the array leaf element types.

Listing 3.14 shows examples of clone methods for single and 2-dimensional arrays of `Student` objects, and Listing 3.15 show examples of clone methods for single and 2-dimensional arrays of `int`'s. These closely resemble the clone methods for arrays of classes except reflection is not used to create the multi-dimensional array clones in Listing 3.14 line 24, and Listing 3.15 lines 7 and 20.

```
1  static Student[] cloneStudent1(Student[] students, List<Boolean> bs, Map m){
2    Object n = m.get(students);
3    if (n != null) {
4      return (Student[]) n;
5    }
6
7    Student[] clone = new Student[students.length];
8    m.put(students, clone);
9
10   Boolean owner = bs.remove(0);
11   for(int i = 0; i < students.length; i++) {
12     clone[i] = owner && students[i] != null ? students[i].clone(bs, m) : students[i];
13   }
14
15   return clone;
16 }
17
18 static Student[][] cloneStudent2(Student[][] students, List<Boolean> bs, Map m){
19   Object n = m.get(students);
20   if (n != null) {
21     return (Student[][]) n;
22   }
23
24   Student[][] clone = new Student[students.length][];
25   m.put(students, clone);
26
27   Boolean owner = bs.remove(0);
28   for(int i = 0; i < students.length; i++) {
29     clone[i] = owner && students[i] != null ? cloneStudent1(students[i], bs, m) : students
         [i];
30   }
31
32   return clone;
33 }
```

Listing 3.14: Array clone methods for `Student` arrays with dimensions known.

```
1  static int[] cloneint1(int[] array, List<Boolean> bs, Map m){
2    Object n = m.get(array);
3    if (n != null) {
4      return (int[]) n;
5    }
6
7    int[] arrayClone = new int[array.length];
8    m.put(array, clone);
9    System.arraycopy(array, 0, arrayClone, 0, array.length);
10
11   return clone;
12 }
13
14 static int[][] cloneint2(int[][] array, List<Boolean> bs, Map m){
15   Object n = m.get(array);
16   if (n != null) {
17     return (int[][]) n;
18   }
19
```

```
20    int[][] arrayClone = new int[array.length][];
21    m.put(array, arrayClone);
22
23    Boolean owner = bs.remove(0);
24    for(int i = 0; i < array.length; i++) {
25      arrayClone[i] = owner && array[i] != null ? cloneint1(array[i], bs, m) : array[i];
26    }
27
28    return clone;
29  }
```

Listing 3.15: Array clone methods for `int` arrays with dimensions known.

3.4 Subclasses

To extend the system for subclassing, considerations will have to be made concerning the cloning parameters and cloning methods of subtypes. The syntax for class declaration is extended as follows:

$$\begin{aligned} ClassDecl ::= \; & class\ C\langle\overline{c}\rangle\{\overline{FieldDecl}\ \overline{MethDecl}\} \mid \\ & class\ C_1\langle\overline{c}\rangle\ extends\ C_2\langle\overline{c'}\rangle\{\overline{FieldDecl}\ \overline{MethDecl}\} \end{aligned}$$

All objects require cloning parameters with length of at least 1. In general, a subclass may have a different number of cloning parameters than its superclass, which we will show may cause problems for cloning. The first cloning parameter of a class indicates the owner, therefore, the first cloning parameter of a subclass must be the same as the first cloning parameter of the superclass. For example, `B<o,o>` is a subtype of `A<o,o>` and `C<o1,o2,o3>` is a subtype of `A<o1,o2>`, given the class declarations in Listings 3.16.

```
1  class A<c1,c2> {
2    D<c1> fd;
3  }
4
5  class B<c1,c2> extends A<c1,c2> {
6  }
7
8  class C<c1,c2,c3> extends A<c1,c2> {
9    C<c1,c2,c3> fc;
10 }
11
12 class D<c> extends A<c,c> {
13   C<c,c,c> fc;
14 }
15
16 class E<c2,c3,c1> extends C<c1,c2,c3> {
17   E<this,c3,c1> fe;
18 }
19
20 class F<c> {
21   A<c,c> fa;
22   C<c,c,c> fc;
23 }
```

Listing 3.16: Examples of subclassing

Under the cloning methods proposed by [15], all classes have cloning methods defined. Therefore, subclasses must also provide the overloaded clone methods: `clone()`, to clone the object, and `clone(Boolean b`$_1$`,...,Boolean b`$_n$`, Map m)`, where `n` is the number of cloning parameters of the subclass. In addition, if a class, *subC* has a different number of cloning parameters to its superclass, *supC*, then the parametric clone method defined in *supC* is also visible from *subC*.

When an object is cloned, its clone method will call the clone method for the field referenced objects that are in the cloning domain. If subclassing is allowed, the dynamic type of a field may differ from its static type

The implications of the cloning approach described in Section 2.3.3 are explored in the following subsections for the different situations a field referenced object can be in:

1. Object's static and dynamic types are the same. No extension to the current cloning approach need to be made for this case.

2. Object's static and dynamic types are different:

(a) With the same number of actual cloning parameters. A potential problem is that if the cloning parameters are in a different order, then Boolean values may be passed to the clone method in the wrong order.

(b) With different actual cloning parameters, where the dynamic type has:

i. Less actual cloning parameters. More cloning parameters will be passed to the clone method than the dynamic class of the object expects. However, we show that the actual cloning parameters of the dynamic type can be derived from the static type.

ii. More actual cloning parameters. We show that this is most problematic case because the actual cloning parameters of the dynamic type cannot be derived from the static type.

Different approaches to tackling the problematic cases include:

Solution 1 Clone referenced objects according to their static type, disregarding their dynamic type (Section 3.4.1).

This requires each class to provide a parametric clone methods to clone an object as each superclass (e.g. class `F` must provide 3 parametric clone methods: `cloneF`, `cloneC` and `cloneA` to clone an instance of the class as an `F`, `C` and `A` object, respectively). This eliminates the problems in case 2.

Solution 2 Clone referenced objects according to their dynamic type (Section 3.4.2).

For each field, the class stores a permutation-like order for actual clone parameters as described in Section 3.2.2. This is used to get the correct order and number of Boolean values when calling the referenced object's parametric clone method.

Solution 3 Restrict assignments of objects to variables/fields so that only cases where the dynamic actual cloning parameters can be derived from the static type (Section 3.4.3).

This solution involves for:

Case 2(a) if the cloning parameters are in a different order to its superclass, they are renamed consistently in the body of the subclass's class declaration so that the formal cloning parameters are in the same order.

Case 2(b)i subclass also overrides the parametric clone method from the superclass.

Case 2(b)ii this case is eliminated – any assignments that causes this case to occur are invalid.

The evaluation of the proposed solutions against the desired cloning properties are summarised in the following table. Since solution 1 does not satisfy property 3 and solution 2 requires additional dynamic information to be stored, we adopt solution 3.

Solution	Property 1	Property 2	Property 3	Property 4
1	No	Yes	No	No additional dynamic information required.
2	Yes	Yes	Yes	An array/list of indices of clone parameters required for at least each field that is of a class that has a subclass.
3	Yes	Yes	Yes	No additional dynamic information required.

Figure 3.24: Table of solutions to subclassing against desired cloning properties

Case 1: Referenced object's static and dynamic types are the same

An example of this case is shown in Listing 3.17 (using class declarations in Listing 3.16), where an instance of class `F`, references a `C<this,this,this>` object through field, `fc`.

Here, cloning the object in field `fc` as part of cloning `e` through the clone method `clone(boolean b, Map m)` in class `F`, and poses no problems. This is regardless of the number of cloning parameters that the subtype has because the class, `F`, knows exactly the class and the clone parameters of the object referenced, `f.fc`. Hence, the clone method in `E` will call the clone method as defined in class `C`, passing the required values to clone the dynamic type of `f.fc`.

```
1 F<this> f = new F<this>();
2 f.fc = new C<this,this,this>();
3 f.clone();
```

Listing 3.17: Example of an object, `f.fc`, whose static and dynamic types are the same, based on class declarations in Listing 3.16.

Case 2a: Referenced object's static and dynamic types are different, but have the same number of cloning parameters.

An example of this case is shown in Listing 3.18, where `f.fa` has static type `A<this,this>` and dynamic type `B<this,this>`.

Because the number of cloning parameters of the dynamic type and the static type are the same, the expected parameters of the cloning methods of the static and dynamic class will be the same. Cloning `e` will result in calling `clone(boolean b, boolean b, Map m)` generated for class `F`, which clones the object at `f.fa`. The `clone(boolean b, boolean b, Map m)` method generated for `B` will be executed due to dynamic dispatch. Hence, the object will be cloned in accordance to its dynamic type and the current cloning methods do not need to be altered.

A potential problem is if a subclass has the same number of cloning parameters as the class it extends, but have the cloning parameters in a different order e.g. class `E` in Listing 3.16. Then when the cloning method is called, the wrong order of Boolean values may be passed if the caller of the method expected the object to be an instance of the superclass.

```
1 F<this> f = new F<this>();
2 f.fa = new B<this,this>();
3 f.clone();
```

Listing 3.18: Example of an object, `f.fa`, with different static and dynamic types and the same cloning parameters, based on class declarations in Listing 3.16.

Case 2b(i): Referenced object's dynamic type has less cloning parameters.

An example of this case is shown in Listing 3.19, where `f.fa` has static type `A<world,world>` and dynamic type `D<world>`.

Class `A` will have a generated parametric clone method that expects 2 boolean arguments, whilst class `B`'s parametric clone method only expects 1 boolean argument (corresponding to the number of formal cloning parameters each class has). As a consequence, when `f.clone()` is executed on line 3, the method call to clone the object at `f.fa` will only be passed 1 Boolean value since class `F` only knows the field has static type `A<world>`. This means that the parametric clone method implemented in `A` is executed and `fClone.fa` has a different dynamic type to `f.fa`.

```
1 F<world> f = new F<world>();
2 f.fa = new D<world>();
3 F<world> fClone = f.clone();
```

Listing 3.19: Example of an object with different static and dynamic types that have different cloning parameters

Case 2b(ii): Referenced object's dynamic type has more cloning parameters.

An example is shown in Listing 3.20, where `f.fa` has static type `A<world,world>` and dynamic type `C<world,world,this>`.

The clone method in `F` will call the clone method in `A` to clone field `fa`. Class `F` does not know the dynamic type of `fa`, which may change during execution. Hence, when the field `f.fa` is cloned, only 2 Boolean values will be passed to the clone method.

```
1 F<world> f = new F<world>();
2 f.fa = new C<world,world,this>();
3 f.clone();
```

Listing 3.20: Example of an object with different static and dynamic types that have different cloning parameters

3.4.1 Solution 1: Clone Referenced Objects According to their Static Type

A solution is to use static binding when calling clone methods, which does not require the proposed clone methods to be changed. This means that class `F` will always call the clone method defined in the `A` class. This guarantees that the cloned object is a copy of the original object with respect to the static type and have a dynamic type equal to the static type.

However, for subclasses (B) that have the same number of cloning parameters as their superclass (A), if an field has dynamic type B<o1,o2> and static type A<o1,o2> then when the field is cloned, the method implemented in class B will be executed. The object should instead be cloned as an A object.

To solve this problem, we can ensure objects are cloned according to their static type by renaming the original parametric clone method for a class, C, as clone$C$ to indicate that it clones itself as type C. So clone$C$ should only be called when the object is a field where the declared type is C. If the object was stored as a superclass A, then clone$A$ (inherited) will be called.

For example, the generated code for the classes in Listing 3.16 are shown in Listing 3.21. Class E implements cloneE) and inherits methods (cloneC and (cloneA. When fields are cloned in parametric clone methods, the clone method matching the static type is called (lines 15, 36, 37, 57 - 59).

```
1  class A {
2    D fd;
3
4    A clone() {
5      return this.clone(false, false, new IdentityHashMap());
6    }
7
8    A cloneA(Boolean b1, Boolean b2, Map m) {
9      Object o = m.get(this);
10     if (o != null) {
11       return (A)o;
12     } else {
13       A oClone = new A();
14       m.put(this,oClone);
15       oClone.fd = b1 && this.fd != null ? this.fd.cloneD(b1,m) : this.fd;
16       return oClone;
17     }
18   }
19 }
20
21 class C extends A {
22   C fc;
23
24   C clone() {
25     return this.cloneC(false, false, new IdentityHashMap());
26   }
27
28
29   C cloneC(Boolean b1, Boolean b2, Boolean b3, Map m) {
30     Object o = m.get(this);
31     if (o != null) {
32       return (C)o;
33     } else {
34       C oClone = new C();
35       m.put(this,oClone);
36       oClone.fd = b1 && this.fd != null ? this.fd.cloneD(b1,m) : this.fd;
37       oClone.fc = b1 && this.fc != null ? this.fc.cloneC(b1,b2,b3,m) : this.fc;
38       return oClone;
39     }
40   }
41 }
42
43 class E extends C {
44   E fe;
45
46   E clone() {
47     return this.clone(false, false, false, new IdentityHashMap());
48   }
49
50   E cloneE(Boolean b1, Boolean b2, Boolean b3, Map m) {
51     Object o = m.get(this);
52     if (o != null) {
53       return (E)o;
54     } else {
55       E oClone = new E();
56       m.put(this,oClone);
57       oClone.fd = b3 && this.fd != null ? this.fd.cloneD(b3,m) : this.fd;
58       oClone.fc = b3 && this.fc != null ? this.fc.cloneC(b3,b1,b2,m) : this.fc;
59       oClone.fe = true && this.fe != null ? this.fe.cloneE(true,b2,b3,m) : this.fe;
60       return oClone;
61     }
62   }
63 }
```

Listing 3.21: Generated code for classes A, C and E in Listing 3.16 under current cloning method.

Cloning Properties

Here we evaluate this solution against the desired cloning properties described in Section 3.1. Properties 1 and 3 do not hold in a subset of situations that fall under the classification of case 2b.

Property 1: All objects that are reachable inside o are also cloned. This does not always hold if class `C` has additional fields (not declared in the superclass `A`) that has `this` as the owner. Because objects are cloned as their static types, only fields that are declared in the static class will be considered when cloning.

Property 2: All objects outside of o must not be cloned. This will hold because static binding means that only fields known in the static class (i.e. the superclass) will be known and considered for cloning. Hence all other referenced objects in the subclass will not be cloned regardless of whether they are inside or outside the object.

Property 3: The heap after cloning o is homomorphic to the heap before cloning o. This cannot be guaranteed for the same reason why property 1 is not guaranteed. The structure of the clone of an object, which has a dynamic type different to its static type, may not be the same as the original object's structure since some fields may not be referenced.

Property 4: Minimise the amount of dynamic information that is needed to be stored. No dynamic information is needed to be maintained.

Other Potential Problems

Other potential problems with the result of cloning according to the static type are: the clone may have a different behaviour; additional difficulty with cloning in certain situations; and abstract classes.

It is possible that the cloned object will not have the exact behaviour of the original object if subclasses can override methods of the superclass. However, we would expect the returned result of calling a method on an object, o, to be the equal to the result of calling a method on o's clone.

There is a potential problem with cloning when an object is referenced by fields with different static type. An example is illustrated in Figure 3.25, where an object, oF, has two fields references to the same object, oAC (whose dynamic type is $\texttt{C}\langle oG, oG, oG\rangle$):

- Via field, `fa`, with static type $\texttt{A}\langle oG, oG\rangle$.
- Via field, `fc`, with static type $\texttt{C}\langle oG, oG, oG\rangle$.

Listing 3.22 shows how these objects and references can be constructed. A possible result of cloning oG is Figure 3.26, where oAC is cloned as an `A` object when field `fa` is cloned before field `fc`. Then when calling `clone(...)` on field `fc`, the clone, oAC', already exists and will try to assign it to the field. However, oAC' has dynamic type $\texttt{A}\langle oG, oG, oG\rangle$ and cannot be assigned to `fc` that has static type $\texttt{C}\langle oG, oG, oG\rangle$.

If abstract classes are supported, another potential problem is when there is a field declaration with an abstract class (i.e. references objects that extend the abstract class). In this case, cloning an object would clone the referenced object as if it was its superclass and return an instance of the abstract class.

```
1 class G<c> {
2   F<this> ff;
3
4   G() {
5     this.ff  new F<this>();
6     this.ff.fc = new C<this,this,this>();
7     this.ff.fa = this.ff.fc;
8   }
9 }
```

Listing 3.22: Example of cloning difficulty when an object is referenced by fields with different static types, based on Listing 3.16.

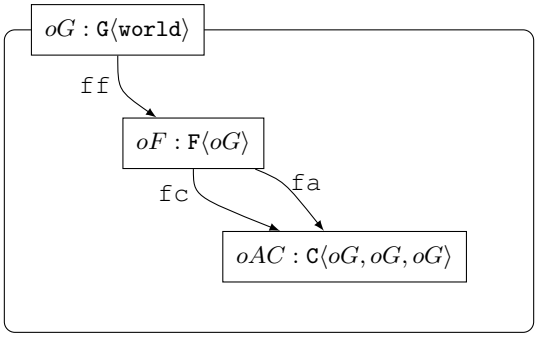


Figure 3.25: Example where an object is referenced by fields with different static types, based on class declarations in Listings 3.22 and 3.16.

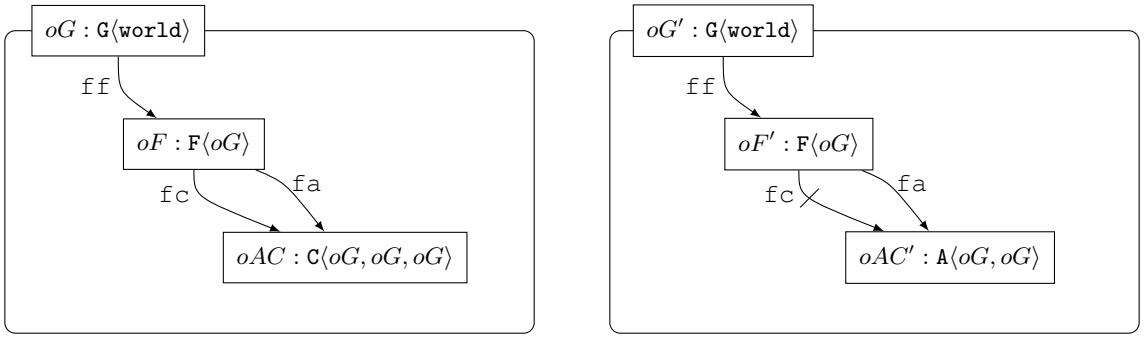


Figure 3.26: A possible result of cloning oG in Figure 3.25, where oAC has been cloned as an `A` object and field `fc` cannot reference the clone of $oACS$.

3.4.2 Solution 2: Clone Objects According to their Dynamic Type

In contrast to solution 1, this solution is to clone objects exactly as their dynamic type even in the case where the static and dynamic types has different clone parameters. This guarantees the cloned object will have the same dynamic type and have the same behaviour even with overridden methods.

To achieve this, the main challenge is the cloning method of `F` does not know statically the exact sequence of parameters that it needs to pass when cloning the field `f.fa`. This can be solved by adopting the alternative cloning method, *Generate Permutation-like Order of Cloning Parameters for Fields*, explained in more detail in Section 3.2.2. A permutation-like order list for each field is stored in the class to give the field's cloning parameters as indices into the class's formal cloning parameters. This is then used when cloning fields, to find the Boolean values to pass to the field referenced object, which may have different actual cloning parameters in its dynamic type from its declared type in the class.

For example, Listing 3.23 will be the code for class `F` from Listing 3.16. When a field, `fa`, is cloned, the boolean values given to `F`'s `clone` method is re-ordered based on the permutation-like order and passed to the `clone` method call.

Here, the parametric clone methods expect 2 arguments exactly, a list of Boolean values (rather than given as individual arguments) and the map of objects cloned. As a result, the parametric clone method executed will be the one implemented in the dynamic class of the object and since the permutation-like order list has been used to re-order `bs`, the clone method receives the correct number and order of Boolean values.

However, the problem with this solution is that the generated clone code requires dynamic information about cloning parameters to be stored and maintained as a list for each field declared in a class.

```
1 class F {
2   A fa;
3   List<Integer> faPerm;
4
5   C fc;
6   List<Integer> fcPerm;
7
```

```
8   F clone() {
9     List<Boolean> bs = new List<Boolean>();
10    bs.add(false);
11    return this.clone(bs, new IdentityHashMap());
12  }
13
14  F clone(List<Boolean> bs, Map m) {
15    Object o = m.get(this);
16    if (o != null) {
17      return (F)o;
18    }
19
20    F oClone = new F();
21    m.put(this, oClone);
22
23    if (this.fa != null) {
24      List<Boolean> bs = Perm.reorder(bs, this.faPerm);
25      oClone.fa = bs.get(0) ? this.fa.clone(bs,m) : this.fa;
26    }
27
28    if (this.fc != null) {
29      List<Boolean> bs = Perm.reorder(bs, this.fcPerm);
30      oClone.fc = bs.get(0) ? this.fc.clone(bs,m) : this.fc;
31    }
32    return oClone;
33  }
34 }
```

Listing 3.23: Generated code that uses permutation-like order lists, in standard Java, for the class `F` in Listing 3.16.

3.4.3 Solution 3: Restrict assignments

Unlike Solution 2, this solution does not use either of the alternative cloning methods described in Section 3.2. The idea of this solution is to only allow assignments of objects to variables/fields if the dynamic clone parameters can be derived from the static clone parameters. This involves:

1. *Generating cloning methods as in the original cloning approach.* This allows cloning works as normal for Case 1, where the static and dynamic types of a field are the same, and for a subset of Case 2(a), where the cloning parameters of a class are the exactly the same as the class it extends.

2. *Reordering formal cloning parameters in class declarations where the class extends another class with the same cloning parameters but in a different order.* This solves the problem with subcases of Case 2(a), where Boolean values could be passed to the cloning method in the wrong order if the caller referenced the object through a field with the superclass as its static type.

3. *Generating clone methods to override superclass' parametric clone method if the class extends a class that has more cloning parameters.* This deals with case 2(b)i, where more Boolean values may be passed to the parametric clone method and hence executing the clone method implemented in the superclass.

4. *Restricting assignments* in the extended language so that any assignments that cause Case 2(b)ii to arise are now invalid.

Reordering Formal Cloning Parameters for Case 2(a)

For classes that have the same number of cloning parameters but in a different order than the class it extends, the formal cloning parameters can be reordered in the subclass' class declaration so that the cloning parameters are expected to be in the same order as its superclass. Listing 3.24 is an example of the rewritten class declaration of the example, where only line 1 has been rewritten. The cloning parameters in field declarations remain unchanged because we are only changing the order in which cloning parameters appear in the class's sequence of formal cloning parameters (and hence in types).

Once, the cloning parameters are reordered, generating the cloning methods according to the original approach is sufficient. Calls to the parametric clone method when the object is referenced as its superclass will result in the subclass's clone method to be called with Boolean arguments passed in the correct order.

```
1 class E<c1,c2,c3> extends C<c1,c2,c3> {
2   E<this,c3,c1> f;
3 }
```

Listing 3.24: Class `E` in Listing 3.16 rewritten with cloning parameters renamed so that they are in the same order as its superclass.

Overriding Superclass' Parametric Clone Method for Case 2(b)i

For classes have less formal cloning parameters than their superclass, override the superclass' parametric `clone` method. The implementation of the method will call the parametric clone method that has the correct number of Boolean values for this class.

For example, the class declaration that would be generated from class `D`, including the overridden clone method is given in Listing 3.25. Lines 21 - 23 is overrides the clone method from `A` and calls its original parametric clone method declared on lines 8 - 19. This is possible because the cloning parameters of the subtype (`D<c>`) can be derived from the cloning parameters of the supertype (`A<c,c>`) when the subclass has less formal cloning parameters.

```
1  class D extends A {
2    C fc;
3
4    D clone() {
5      return clone(false, new IdentityHashMap());
6    }
7
8    D clone(Boolean b1, Map m) {
9      Object o = m.get(this);
10     if (o != null) {
11       return (D) o;
12     }
13
14     D oClone = new D();
15     m.put(this, oClone);
16     oClone.fd = b1 ? this.fd.clone(b1, m) : this.fd;
17     oClone.fc = b1 ? this.fc.clone(b1, b1, b1, m) : this.fc;
18     return oClone;
19   }
20
21   D clone(Boolean b1, Boolean b2, Map m) {
22     return clone(b1, m);
23   }
24 }
```

Listing 3.25: Example of class `D` that override the clone method from its superclass, based on Listing 3.16.

Restricting Assignments

This solution is to restrict assignments of subtypes to only allow assignments of objects that are subtypes to the static type of the receiver and do not have more cloning parameters. The problematic case 2b will not occur if this is enforced.

For example, `A<this,this> a = new B<this,this>()` is a valid assignment but `A<this,this> a = new C<this,this,this>()` is an invalid assignment. Hence, `C` objects can only be referenced through fields that are declared as type an instance of class `C`.

3.5 Generics

In Java, generics are used such as for the generic class, `List<T>`, where `T` is a generic type parameter, and when a List is instantiated, the actual type is passed in the parameters e.g. `new List<String>()`. Extending cloning to generics requires solving the problem that generic types, `T`, are unknown until the class is instantiated. Therefore, statically, it is not known what the cloning parameters of the generic type is and consequently, the it is not known which Boolean values should be passed to the referenced objects of type `T` when cloning the generic class.

Extending the system involves:

- Extending the syntax with parameters for generics (Section 3.5.1).
- Solving the problem of how objects know what information is required to be passed to the cloning method of fields in a generic class (Section 3.5.2).

  The proposed solution is to store permutation-like order lists for generic type parameters, a variation to the alternative cloning method described in Section 3.2.2. These indicate the cloning parameters of generic parameters by indexing into the formal cloning parameters.

The solution satisfies properties 1, 2 and 3, but it requires dynamic information to be stored in the corresponding standard Java code for each generic class, otherwise there is no way of knowing the actual cloning parameters of type parameters. This is unless a restriction is placed on generic class declarations

such as ensuring that cloning parameters of type parameters are declared in class declarations. For example `class C<c1,c2,T<c2>> {...}`, where the classes that `T` may take are those that have a single cloning parameter (which must be the second cloning parameter of the object instance of `C`). This is likely to be too restrictive in practice because the type that the generic type can take would be restricted to classes that have the exact same number of cloning parameters.

3.5.1 Syntax

The syntax for class declaration and types are extended as follows:

$$
\begin{aligned}
\mathit{ClassDecl} &::= \texttt{class}\ C\langle \overline{c}, \overline{G}\rangle\{\overline{\mathit{FieldDecl}}\ \overline{\mathit{MethDecl}}\} \\
\mathit{FieldType} &::= C\langle \overline{ca}, \overline{Ga}\rangle \mid Ga \\
G &::= \mathit{Identifier} && \text{generic type identifiers} \\
Ga &::= G \mid t && \text{actual type parameter}
\end{aligned}
$$

Classes can be declared with type parameters after the clone parameters, can be used as field types and must be instantiated with a type when an instance of the class is created. For example, Listing 3.28 shows a generic class declaration, where G parameters are the generic type parameters. Constructing a new instance of the class `A` involves passing actual types in place of the generic parameters, for example, `A<`c_1`,`c_2`,B<`c_2`>> a = new A<`c_1`,`c_2`,B<`c_2`>>()`.

The actual cloning parameters of type parameters must be one of the actual parameters of the class, i.e. $\{c_1, c_2\}$ for the current example, and actual type parameters do not change over time.

```
1 class A<c1,c2,G> {
2   G f1;
3   C<c2> f2;
4 }
```

Listing 3.26: Example of generic class A in standard Java.

3.5.2 Solution

For generic classes the problem is that the sequence of cloning parameters for type parameters are not known statically and hence the Boolean values to pass to the clone method of any generic field is unknown. This can be solved by adapting the alternative cloning method, *Generate Permutation-like Order of Cloning Parameters for Fields*, explained in more detail in Section 3.2.2. A permutation-like order list for each generic type parameter is stored in the class to give the generic type's cloning parameters as indices into the class's formal cloning parameters. This is then used when cloning fields that use generic type parameter as a field type, to find the Boolean values to pass to the field referenced object.

The permutation-like order list should be assigned when generic class is instantiated and do not change since actual type parameters of an object cannot change over time. This removes the need to maintain the permutation-like order list once it is first assigned.

Constructors

Since the order of actual cloning parameters is only known during instantiation, the constructors of generic class will accept a permutation-like order list for each generic type parameter and stores them in new fields. For example, the translated class `A` is shown in Figure 3.28, where the constructor is shown in lines 7 - 9, storing the list in a new field, `gPerm`.

For every instantiation of a generic class, the permutation-like order list is generated and passed to the constructor, e.g. `A<c1,c2,B<c2>> a = new A<c1,c2,B<c2>>()` becomes Listing 3.27.

```
1 List<Integer> gPerm = new List<Integer>();
2 gPerm.add(1);
3 A<B> a = new A<B>(gPerm);
```

Listing 3.27: Example of corresponding standard Java code for `A<c1,c2,B<c2>> a = new A<c1,c2,B<c2>>()`.

```
1 class A<c1,c2,G> {
2   G f1;
3   C f2;
4
5   List<Integer> gPerm;
```

```
6
7   A(List<Integer> gPerm) {
8     this.gPerm = gPerm;
9   }
10
11  A<G> clone() {
12    List<Boolean> bs = new List<Boolean>();
13    bs.add(false);
14    bs.add(false);
15    return this.clone(bs, new Map());
16  }
17
18  A<G> clone(List<Boolean> bs, Map m) {
19    Object n = m.get(this);
20    if (n != null) {
21      return (A<G>)n;
22    }
23
24    A<G> oClone = new A<G>(this.gPerm);
25    m.put(this, oClone);
26
27    oClone.f1 = this.f1 != null && bsG.get(this.gPerm(0)) ? this.f1.clone(Perm.reorder(bs,
          this.gPerm), m) : this.f1;
28
29    List<Boolean> bsc = new List<Boolean>();
30    bsc.add(bs.get(1));
31    oClone.f2 = this.f2 != null && bs.get(1) ? this.f2.clone(bsc, m) : this.f2;
32
33    return oClone;
34  }
35 }
```

Listing 3.28: Example of generic class A in standard Java.

Clone methods

The permutation-like order list is sufficient for deriving the corresponding Boolean values (from the parametric clone method) that will have to be passed when cloning fields that are instances the generic type parameter. In this solution, parametric clone methods accept a `List<Boolean>` argument instead of individual Boolean arguments, and use the stored ordering when checking whether to clone a field of type `G` and passing boolean values.

This general cloning method involving generics is shown Figure 3.29, where the changes are lines:

25 Changed parametric `clone` method signature by replacing Boolean parameters with a list of Booleans. This causes necessary changes to the implementation of the `clone()` method in lines 17 - 23.

33 - 35 The Boolean value corresponding to the owner of each field is checked before deciding whether to clone. If the Boolean is `true` then the field is cloned by calling `clone` on the field and passing the Boolean values that it expects by reordering the list of boolean values by using the permutation-like order list. This uses the helper method defined in Listing 3.5 to get the list of Boolean values to pass to the generic field's clone method.

```
1  class C<G1,...,Gk> {
2    G1 f1;
3    ...
4    Gn fn;
5
6    List<Integer> g1Perm;
7    ...
8    List<Integer> gkPerm;
9
10   C(List<Integer> G1,..., List<Integer> Gk) {
11     this.g1perm = g1perm;
12     ...
13     this.gkperm = gkperm;
14
15   }
16
17   C<G1,...,Gn> clone() {
18     List<Boolean> bs = new List<Boolean>();
19     bs.add(false);
20     ...
21     bs.add(false);
22     return this.clone(bs, new Map());
23   }
24
```

```
25  C<G1,...,Gn> clone(List<Boolean> bs, Map m) {
26    Object n = m.get(this);
27    if (n != null) {
28      return (C<G1,...,Gn>)n;
29    }
30
31    C<G1,...,Gn> oClone = new C<G1,...,Gn>(g1Perm,...,gnPerm);
32
33    oClone.fg1 = bs.get(g1Perm.get(0)) ? fg1.clone(Perm.reorder(bs, g1Perm), m) : fg1;
34    ...
35    oClone.fgk = bs.get(gkPerm.get(0)) ? fgk.clone(Perm.reorder(bs, gkPerm), m) : fgk;
36
37    return oClone;
38  }
```

Listing 3.29: General generic class implementation in standard Java.

3.6 Combining Extensions for Arrays, Subclasses and Generics

We have so far considered extending the cloning approach for arrays, subclasses and generics separately. How these extensions fit together is discussed in this section, including the extended syntax to include all three features. First, we consider the effects the features will have on each other (Section 3.6.1), then we present the refined extended syntax that covers the three features (Section 3.6.2) and the cloning methods (Section 3.6.4).

3.6.1 Effects of Arrays, Subclasses and Generics on Each Other

The selected solutions for each feature were:

- *Arrays* – generate an array clone method for each class and each dimension (Section 3.3.5).

  This solution uses the alternative cloning method described in Section 3.2.1, where parametric cloning methods have a `List<Boolean>` argument in place of the sequence of Boolean arguments.

- *Subclassing* – restrict assignments of objects to variables/fields so that only cases where the dynamic actual cloning parameters can be derived from the static type (Section 3.4.3).

- *Generics* – generate permutation-like order lists for deriving cloning parameters of generic type parameters (Section 3.5.2).

  This uses versions of the alternative cloning methods described in Section 3.2, where parametric cloning methods have a `List<Boolean>` argument in place of the sequence of Boolean arguments and use permutation-like order list to reorder Boolean values.

The implications of combining these solutions in order to have these features in one extension are:

- The solutions for the arrays and generics extensions require all parametric cloning methods to use `List<Boolean>` as an argument, whilst the originally proposed cloning methods pass these values as individual arguments. All parametric `clone` methods for the subclassing extension will have to be changed to use a list.

  This will not be a problem for arrays, but the solution for subclassing involve generating two parametric clone methods when subclasses have less cloning parameters than the class they extend. The difference between the signatures of the two clone methods are by the number of Boolean values, hence this can be adapted to using `List<Boolean>` by having one parametric clone method and performing a check on the length of the list to decide what actions to take.

- Arrays may have elements that are subtypes of the declared array element type. This will not pose a problem because we restrict assignments of subclass objects to have the same clone parameters as its superclass. Hence, all objects in an array can handle cloning when Boolean values expected by the superclass are passed to subclass' parametric cloning methods.

- There may be generic arrays declared as in fields of a generic class. This is where a generic type parameter is used to specify the type of array leaf elements (i.e. the class and formal cloning parameters of objects in an array). The arrays extension require a cloning method to be generated for each class and dimension that appears in the program. Therefore, a generic array cloning

method need to be generated for each generic array and dimension combination. However, since these will be generic arrays, reflection is needed to be used to create the array clone.

The generated permutation-like order lists can be used to order the Boolean values in the list and appending to the front of the list, the known Boolean values representing the owners of array. This final list can then be passed to the generic array cloning method, which operates very similarly to the usual array cloning methods.

- Generic classes may have subclasses assigned to fields whose type is given by a type parameter. This will not pose a problem because when an instance of the generic class is created, the actual type becomes known and cloning parameters of the generic types can be derived from the permutation-like order list. For subclassing, we restrict assignments of subclass objects to so that subclasses can only be assigned if they can handle cloning when the number of Boolean values expected by the superclass is passed. Hence, objects can only be assigned to generic fields if they are subtypes that have less of the same number of cloning parameters.
- Generic class declarations can specify bounds for generic types parameters, this restricts the types that the generic type parameter may be instatiated as. This corresponds to bounded type parameters in Java such as `class C<T extends String>`. Since the supertype of the type parameter is declared, the cloning parameters of the supertype are also known. As a result, the permutation-like order list stored will be indices for the supertype's cloning parameters. This can be further optimised by removing the permutation-like order list for bounded generic type parameters because the relative cloning parameters are constant for the class.

The combined solution has the desired cloning properties 1, 2, and 3, but the generics solution require some dynamic information to be stored in the derived code. However, this is necessary since actual cloning parameters of generic type parameters are unknown until the generic class is instantiated.

3.6.2 Syntax

The refined extended syntax that covers the three features is the following:

$$
\begin{aligned}
ClassDecl ::=\ & \texttt{class}\ C\langle\overline{c}, \overline{GPar}\rangle\{\overline{FieldDecl}\ \overline{MethDecl}\} \\
& \texttt{class}\ C_1\langle\overline{c}, \overline{GPar}\rangle\ \texttt{extends}\ C_2\langle\overline{c'}, \overline{ga}\rangle\{\overline{FieldDecl}\ \overline{MethDecl}\} \\
FieldType ::=\ & \texttt{C}\langle\overline{ca}, \overline{ga}\rangle \mid t[]\langle ca\rangle \mid ga \\
GPar ::=\ & G \mid G\ \texttt{extends}\ C\langle\overline{ca}\rangle \\
G ::=\ & Identifier && \text{generic type identifiers} \\
ga ::=\ & G \mid t && \text{actual type parameter}
\end{aligned}
$$

Listing 3.30 is an example of class declarations that uses all features, using the extended syntax. ClassA has a generic type parameter and class `B` is a subclass of `A`, with bounded generic type parameter and an array of elements that have the given type parameter.

```
1  class A<c1,c2,G> {
2    A<c1,c2,G> fa1;
3    G fa2;
4  }
5
6  class B<c1,c2,G extends A<c1,c2>> extends A<c1,c2,G> {
7    C<c2>[]<this> fb1;
8    G[]<tihs> fb2;
9  }
10
11 class C<c> {
12 }
```

Listing 3.30: Example of classes using arrays, subclassing and generics.

3.6.3 Restricting Assignments

The solution for the subclassing extension includes restricting assignments of subtypes to only allow assignments of objects that are subtypes to the static type of the receiver and do not have more cloning parameters. This can be extended to generics by adding the constraint that the actual type parameters must match (including the actual cloning parameters).

For example, Listing 3.31 contains examples of valid assignments and Listing 3.32 contains examples of invalid assignments.

```
1 A<this,this,C<this>> ac;
2 B<this,C<this>> bc;
3 ac = bc;
4
5 A<this,this,A<this,this,C<this>>> aac;
6 B<this,A<this,this,C<this>>> bac;
7 aac = bbc;
```

Listing 3.31: Example of valid assignments based on class declarations in Listing 3.30.

```
1 A<this,this,C<this>> acThis;
2 A<this,this,C<world>> acWorld;
3 A<this,world,C<this>> aWorldc;
4 B<this,C<world>> bcWorld;
5 B<world,C<this>> bWorldc;
6
7 // Generic type parameters do not match: C<this> and C<world>
8 // ac = acWorld;
9
10 // Cloning parameters do not match: A<this,this,G> and A<this,world,G>
11 // ac = aWorldc;
12
13 // Generic type parameters do not match: C<this> and C<world>
14 // ac = bcWorld;
15
16 // Cloning parameters do not match: A<this,this,G> and B<world,G>
17 // ac = bWorldc;
18
19 // Generic type parameters do not match: A<this,this,C<this>> and B<this,C<this>>
20 A<this,this,A<this,this,C<this>>> aac;
21 A<this,this,B<this,C<this>>> abc;
22 // aac = abc;
```

Listing 3.32: Example of invalid assignments based on class declarations in Listing 3.30.

3.6.4 Generating Cloning Methods

The steps involved in generating the Java code with cloning methods will be explained using the example in Listing 3.30:

1. To allow cloning generic arrays:
   (a) Generate an interface, `Cloneable`, that declares the cloning methods that all classes will implement (Listing 3.33 lines 1 - 4).
   (b) Generate a new class for cloning arrays that have array leaf elements of primitive types or generic types (Listing 3.33 lines 6 - 22):
      - Generate static generic array cloning method for each dimension of generic arrays that appears in the program (lines 7 - 21). This expects `bs` to be the Boolean values to be passed for cloning `G` objects.
2. For each appearance of a non-generic arrays:
   (a) In the new class for cloning arrays (from the previous point 1b), generate a static array cloning method for each dimension of arrays with primitive type leaf elements. (None exist in our example.)
   (b) Generate a static array cloning method for each dimension and class, C, combination that appears in the program, in the class declaration of class C. (Listing 3.34 lines 20 - 35, is the 1-dimensional array cloning method for class `C`, generated because `C[]` is a field of `B`)
3. For cloning generic classes, `A` and `B`:
   (a) Generate a class, `Perm`, that contains the helper method, `reorder`, that returns a reordered list of Boolean values should be passed to clone objects associated with the given permutation-like order list. (Listing 3.33 lines 24 - 33).
   (b) Add a field in generic classes to store a permutation-like order list for each type parameter (Listing 3.34 line 5, this field also inherited by class `B`).

(c) Add a constructor for generic classes that accepts a permutation-like order list for each type parameter (Listing 3.34 lines 7 - 9 and Listing 3.35 lines 5 - 7).

4. Cloning in general:

   (a) Check whether a clone of this object already exists (Listing 3.34 lines 19 - 22, Listing 3.35 lines 16 - 19, and Listing 3.36 lines 9 - 12). If it does not exist then create a new instance of the class.

   (b) If the class extends a class with more cloning parameters, check the length of the Boolean list argument. If the list has more elements than the number of cloning parameters of the current class, then derive and reconstruct the list of Boolean values for the subclass from the superclass. (Listing 3.35 lines 21 - 26.)

   (c) Before cloning fields, construct a new list of Boolean values that should be passed to clone the field (Listing 3.34 lines 27 - 30 and Listing 3.35 lines 31 -34).

   Additionally, for generic types, reorder the list of Boolean values before calling clone for fields of generic type (Listing 3.34 lines 32 - 33 and Listing 3.35 lines 36 -37).

   (d) When cloning arrays:

   - Construct the list of Booleans with Boolean values representing the owner of the outermost array listed first and the last Boolean values are those for cloning the array leaf elements. (Listing 3.35 lines 39 - 41)
     Additionally, for generic arrays, this involves reordering the , to get the correct order of Boolean values for cloning the generic leaf elements. (Listing 3.35 lines 44 - 45).
   - Remove the Boolean value representing the outermost array to check whether to clone the array object. If the elements of the outermost array should be cloned, call clone on each element while passing the updated list of Boolean values and the map. (Listing 3.35 lines 42 and 46).

```
1  interface Cloneable {
2    Cloneable clone();
3    Cloneable clone(List<Boolean> bs, Map<Object, Object> m);
4  }
5
6  class Arrays {
7    static <G> G[] cloneG1(G[] array, List<Boolean> bs, Map m) {
8      Object n = m.get(array);
9      if (n != null) {
10       return (G[])n;
11     }
12
13     G[] oClone = (G[]) Array.newInstance(array.getClass().getComponentType(), array.length
           );
14
15     Boolean owner = bs.get(0);
16     for(int i = 0; i < array.length; i++) {
17       oClone[i] = (G) (owner && array[i] != null ? ((ICloneable)array[i]).clone(bs,m) :
             array[i]);
18     }
19
20     return oClone;
21   }
22 }
23
24 class Perm {
25   static List<Boolean> reorder(List<Boolean> bs, List<Integer> perm) {
26     List<Boolean> reordered = new List<Boolean>();
27     for(Integer index : perm) {
28       Boolean b = index == -1 ? true : (index == -2 ? false : bs.get(index));
29       reordered.add(b);
30     }
31     return reordered;
32   }
33 }
```

Listing 3.33: Extra classes/interface that should be generated for Listing 3.30.

```
1  class A<G> implements Cloneable {
2    A<G> fa1;
3    G fa2;
```

```
4
5   List<Integer> gPerm;
6
7   A(List<Integer> gPerm) {
8     this.gPerm = gPerm;
9   }
10
11  A clone() {
12    List<Boolean> bs = new List<Boolean>();
13    bs.add(false);
14    bs.add(false);
15    return this.clone(bs, new IdentityHashMap());
16  }
17
18  A clone(List<Boolean> bs, Map m) {
19    Object n = m.get(this);
20    if (n != null) {
21      return (A) n;
22    }
23
24    A clone = new A(this.gPerm);
25    m.put(array, clone);
26
27    List<Boolean> fBs = new List<Boolean>();
28    fBs.add(bs.get(0));
29    fBs.add(bs.get(0));
30    clone.fa1 = this.fa1 != null && bs.get(0) ? this.fa1.clone(fBs,m) : this.fa1;
31
32    fBs = Perm.reorder(bs, this.gPerm);
33    clone.fa2 = this.fa2 != null && fBs.get(0) ? this.fa2.clone(fBs,m) : this.fa2;
34
35    return clone;
36  }
37 }
```

Listing 3.34: Class that should be generated for Class `A` from Listing 3.30.

```
1  class B<G extends C> extends A<G> implements Cloneable {
2    C[] fb1;
3    G[] fb2;
4
5    B(List<Integer> gPerm) {
6      this.gPerm = gPerm;
7    }
8
9    B clone() {
10     List<Boolean> bs = new List<Boolean>();
11     bs.add(false);
12     return this.clone(bs, new IdentityHashMap());
13   }
14
15   B clone(List<Boolean> bs, Map m) {
16     Object n = m.get(this);
17     if (n != null) {
18       return (B) n;
19     }
20
21     if (bs.size() == 2) {
22       // Called as superclass
23       List<Boolean> deriveBs = new List<Boolean>();
24       deriveBs.add(bs.get(0));
25       bs = deriveBs;
26     }
27
28     B clone = new B(this.gPerm);
29     m.put(array, clone);
30
31     List<Boolean> fBs = new List<Boolean>();
32     fBs.add(bs.get(0));
33     fBs.add(bs.get(0));
34     clone.fa1 = this.fa1 != null && bs.get(0) ? this.fa1.clone(fBs,m) : this.fa1;
35
36     fBs = Perm.reorder(bs, this.gPerm);
37     clone.fa2 = this.fa2 != null && fBs.get(0) ? this.fa2.clone(fBs,m) : this.fa2;
38
39     fBs = new List<Boolean>();
40     fBs.add(true);
41     fBs.add(bs.get(0));
42     clone.fb1 = this.fb1 != null && fBs.remove(0) ? C.clone1(this.fb1,fBs,m) : this.fb1;
43
44     fBs = Perm.reorder(bs, this.gPerm);
45     fBs.add(0,true);
```

```
46      clone.fb2 = this.fb2 != null && fBs.remove(0) ? Arrays.cloneG1(this.fb2,fBs,m) : this.
            fb2;
47
48      return clone;
49    }
50  }
```

Listing 3.35: Class that should be generated for class `B` from Listing 3.30.

```
1  class C implements Cloneable {
2    C clone() {
3      List<Boolean> bs = new List<Boolean>();
4      bs.add(false);
5      return this.clone(bs, new IdentityHashMap());
6    }
7
8    C clone(List<Boolean> bs, Map m) {
9      Object n = m.get(this);
10     if (n != null) {
11       return (C) n;
12     }
13
14     C clone = new C();
15     m.put(array, clone);
16
17     return clone;
18   }
19
20   static C[] clone1(C[] array, List<Boolean> bs, Map m) {
21     Object n = m.get(array);
22     if (n != null) {
23       return (C[]) n;
24     }
25
26     C[] clone = new C[array.length];
27     m.put(array, clone);
28
29     Boolean owner = bs.get(0);
30     for(int i = 0; i < array.length; i++) {
31       clone[i] = owner && array[i] != null ? array[i].clone(bs, m) : array[i];
32     }
33
34     return clone;
35   }
36 }
```

Listing 3.36: Class that should be generated for class `C` from Listing 3.30.

Chapter 4

Implementation

As a proof of concept, I implemented the basic cloning approach (Section 2.3) as a new Java extension language, which I named *CloneCodeGen*. This was implemented by using Polyglot to build a pre-processor/compiler that takes source code written in *CloneCodeGen* and translates it into standard Java with generated clone method as described in Section 2.3.3.

The process of compiling source code written in the extended language into standard Java is shown in Figure 4.1. The input file is parsed to produce an internal object structure that is the abstract syntax tree (AST) representing the program. By default Polyglot traverses the AST several times (called *passes*), each time performing different checks, modifying the AST. The last pass transforms the AST to a String representation of the program in standard Java and writes it to an output file. Changes made to the default set of passes included:

- Extension of the parser to recognise the extended syntax and create an AST with extended nodes where necessary.
- To generate the clone methods, a new pass, *GenerateCloneCode*, was inserted before the Type Check pass (Section 4.2). This pass adds the implementation of clone methods to the AST so that it will be translated by Polyglot's default passes. This pass is done here because if it was inserted after the Type Check pass instead, type checked information needs to be manually added to nodes created during this new pass.

The aim of the pre-processor is to show generating clone methods based on ownership types is possible, so only minimal type checking was implemented as a simplification and assume only type-safe source code will be given as input. Hence, I have only implemented checks to ensure the owner parameters used in field declarations are in scope (i.e. owner parameters may be `this`, `world` and the formal owner parameters of the class declaration). This ensures the compiler correctly generates the implementation of clone methods.

4.1 Structure

The overall structure of the compiler is shown in Figure 4.2, this is generated by the script provided with Polyglot for creating Polyglot extension. The main components that are modified for this extension are:

- *ExtensionInfo* class is modified to add the new Genereate Clone Code pass to the scheduler.
- *parse* package contains the extended *ppg*, *flex* and generated *cup* files. When the compiler is built, these files generate the classes that parse input files and create AST nodes by using the node factory in the *ast* package.
- *ast* package contains classes that represent nodes of AST and the node factory, which creates the concrete instances of the node. These classes and interfaces extend corresponding classes from Polyglot
- *visit* package contains the class that visits nodes in the AST for the Generate Clone Code pass. It adds clone methods for class declarations that appear in the AST.

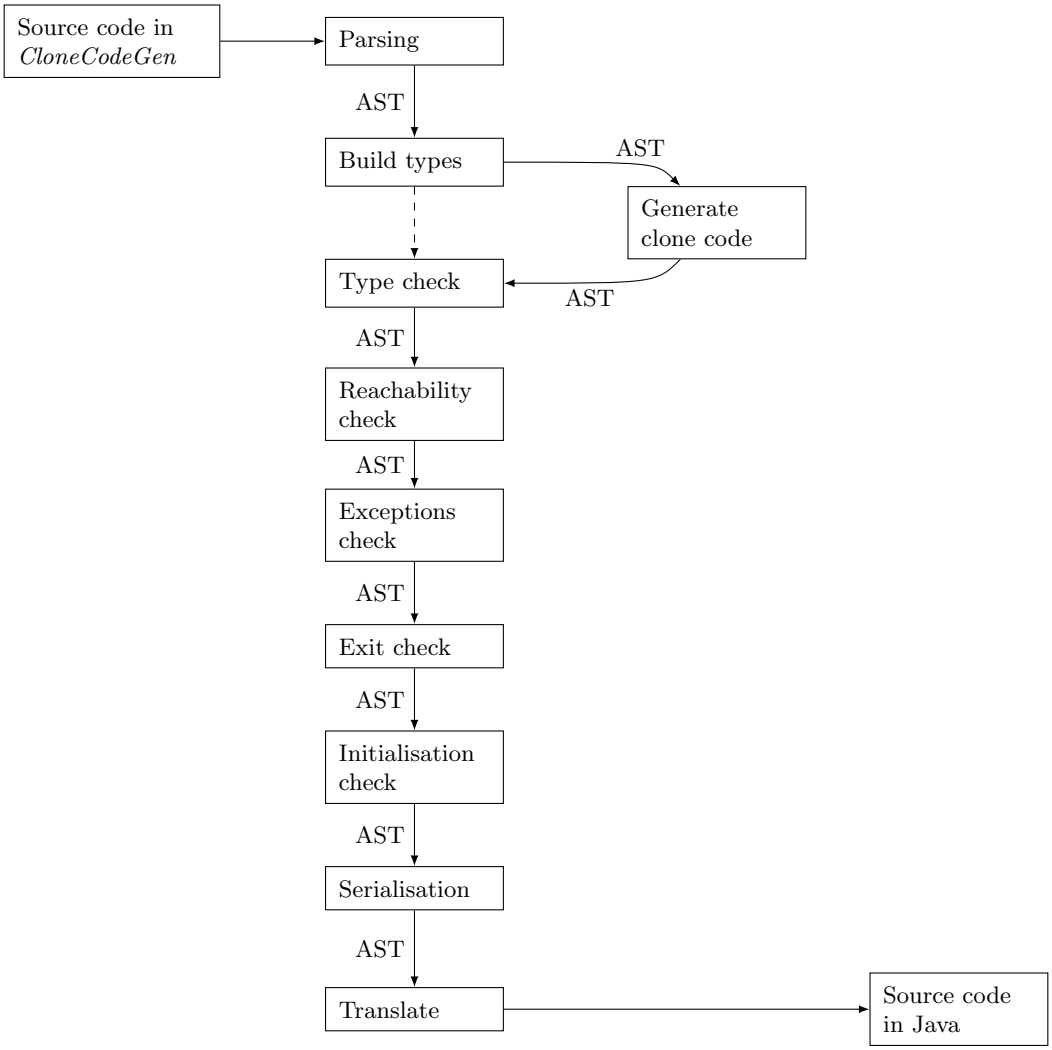


Figure 4.1: Overview of process of compiling CloneCodeGen source code into Java source code.

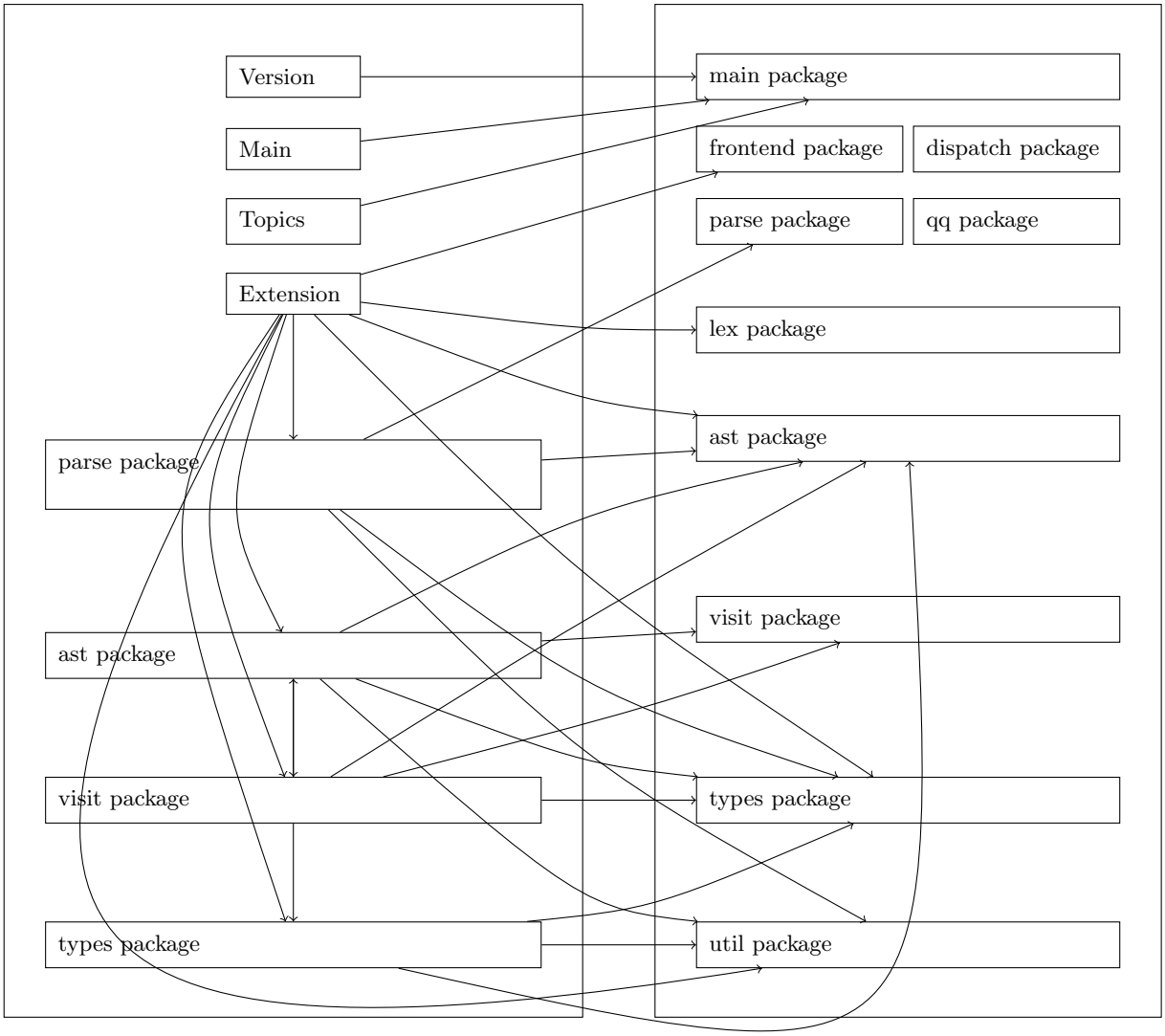


Figure 4.2: Overview of components of *CloneCodeGen* compiler and dependencies between them.

- *types* package contains classes that represent: the extended type system, new types, exceptions, and context extended with owner parameters in scope.

Figure 4.3 shows the structure of the *clonecgen.parse* package, most files are automatically generated when the compiler is built. The files created/modified are:

- New *CCGParsedName* class extending ParsedName from Polyglot with a collection of Id AST node objects that are used to represent owner parameters.
- Extended *cloneCGen.ppg* grammar file to expect owner parameters wherever reference types are expected and not allow subclassing and interfaces. This uses the new node factory to create the AST .

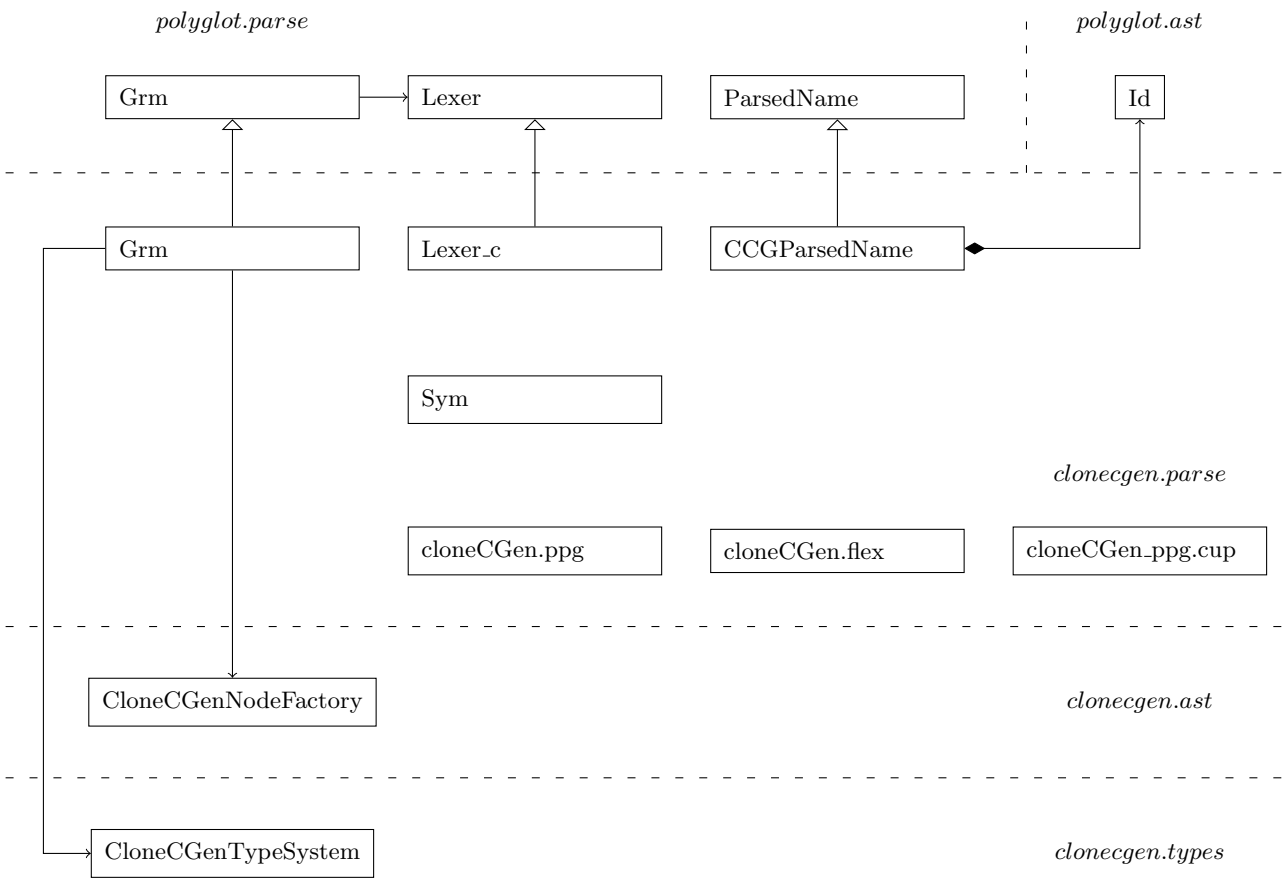


Figure 4.3: Class diagram of *CloneCodeGen*'s Parse package.

Figure 4.4 is the class diagram of the AST package, where:

- `CCGNodeFactory` interface and implementing class extends the node factory in Polyglot so that the new CCGAmbTpeNode and CCGClassDecl nodes are created instead of the classes that they extend and the creation of other nodes is not changed.
- `CCGAmbTypeNode` represents types before they are disambiguated and type checked, the actual owner parameters of the type are stored as a list of Id objects. The interface is extended with getters and setters for the owner parameters list.
- `CCGClassDecl` is the extended class declaration node, where the work on generating and adding clone methods occur when visited in the Generate Clone Code pass. This involves adding the `Cloneable` interface to the node; creating new nodes for each method, statement, and expression; and ensuring a default constructor exists and each field's owner parameters are in scope.

Other nodes and the majority of the inherited methods for the extended classes are unchanged so that the default scheduled passes have the default behaviour of checking and transforming the AST into standard Java.

Figure 4.5 is the class diagram of the Types package, where:

- `CCGTypeSystem` creates the extended context, `CCGContext_c`, and creates array type objects extended with owner parameters. Although cloning of arrays is not implemented, we allow arrays to be used in the extension language and assume it is not used as a type in field declarations.
- `CCGArrayType` extends `ArrayType` with the owner of the array object.
- `CCGContext` interface and implementing class extend `Context` with a list of String representing the owner parameters in scope and a method for checking whether an owner is in scope. "this"

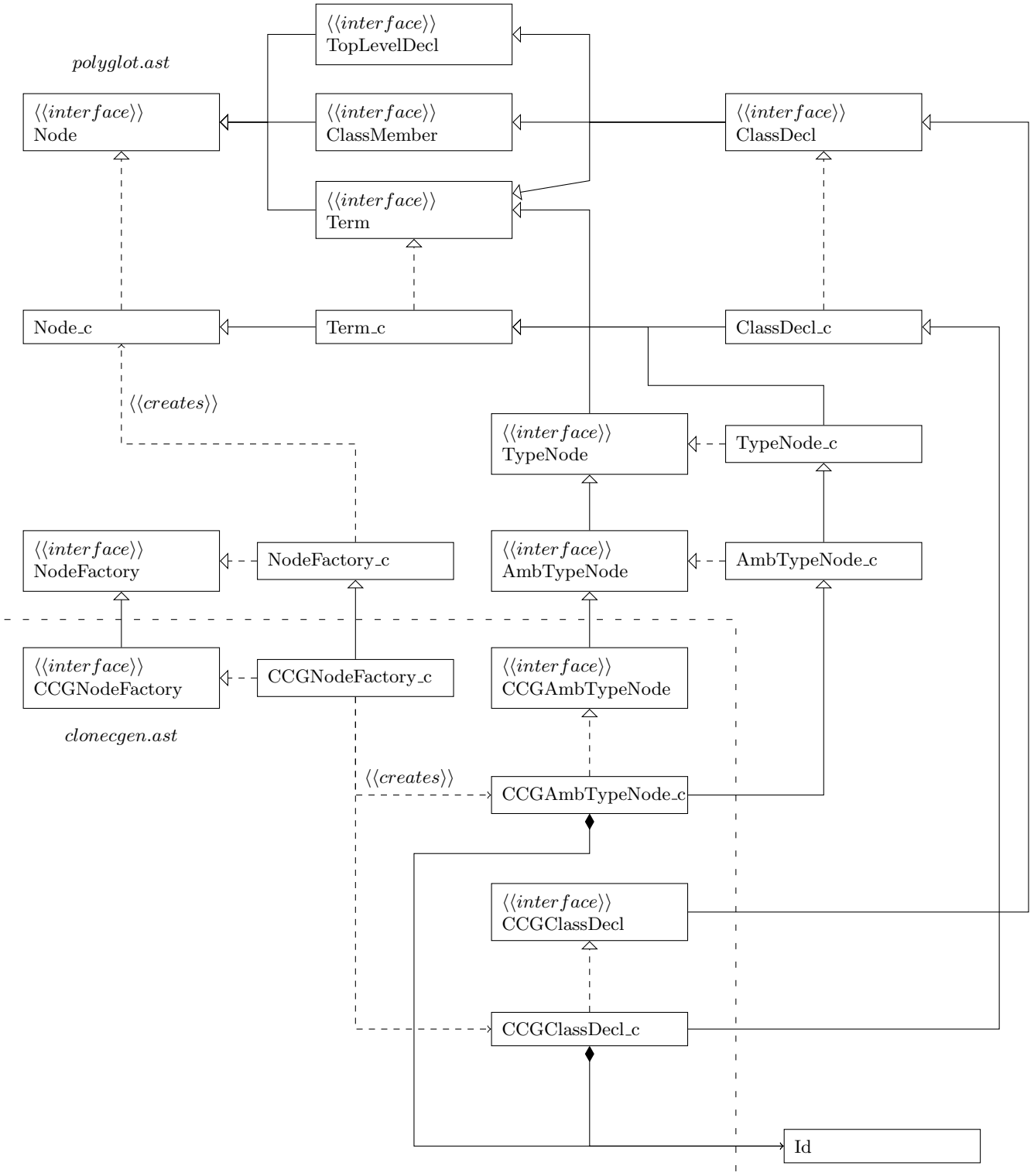


Figure 4.4: Class diagram of *CloneCodeGen*'s AST package.

and "world" are always in scope, and the formal owner parameters of a class declaration are added to the list when the class' scope is entered.

Figure 4.6 is the class diagram of the Visit package, where there is only one class, `CloneCodeGenerator`. This class visits the nodes in an AST and whenever it visits an extended class declaration node, it calls the new method in `CCGClassDecl` that generates and add clone code to the class.

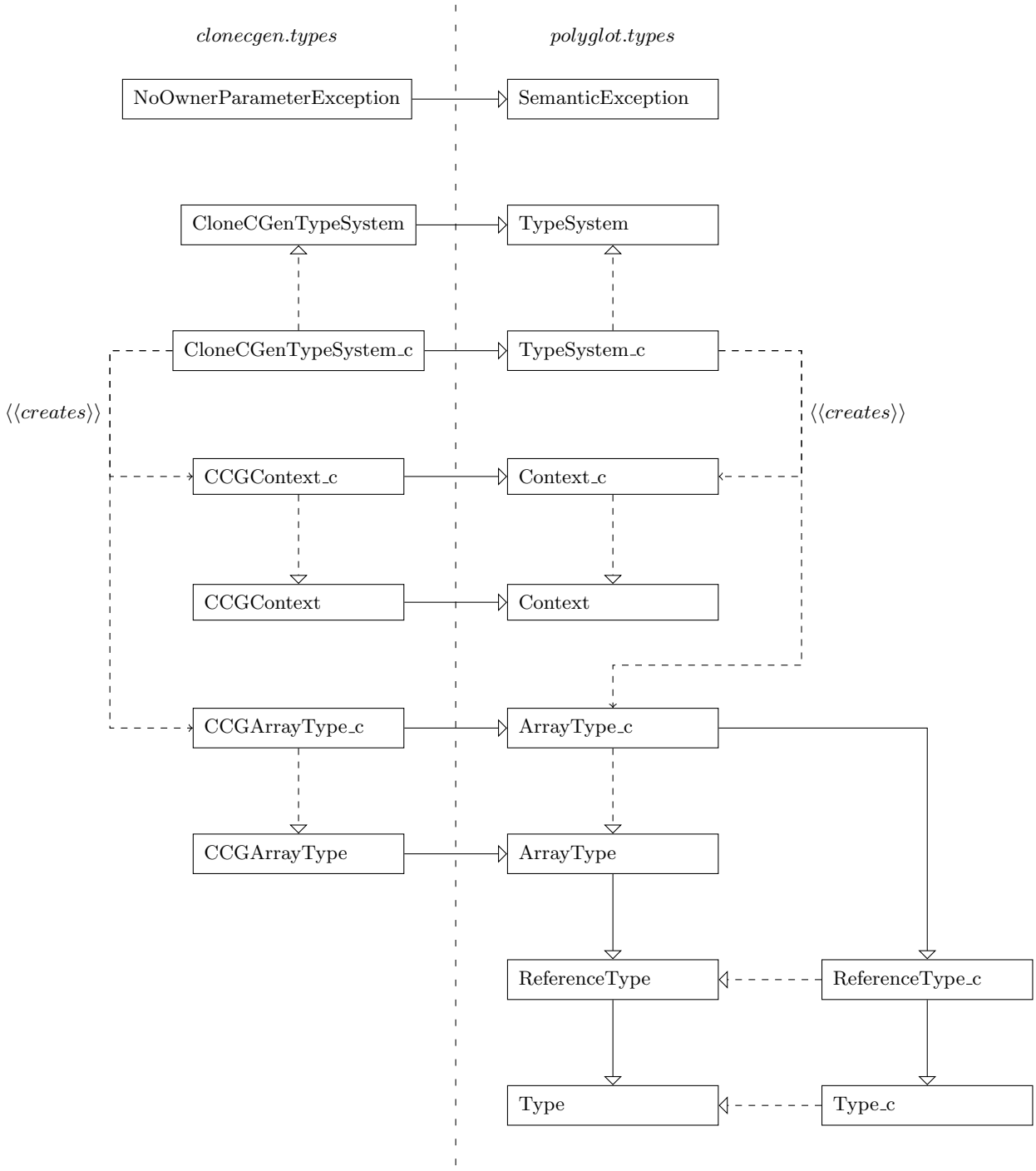


Figure 4.5: Class diagram of *CloneCodeGen*'s Types package.

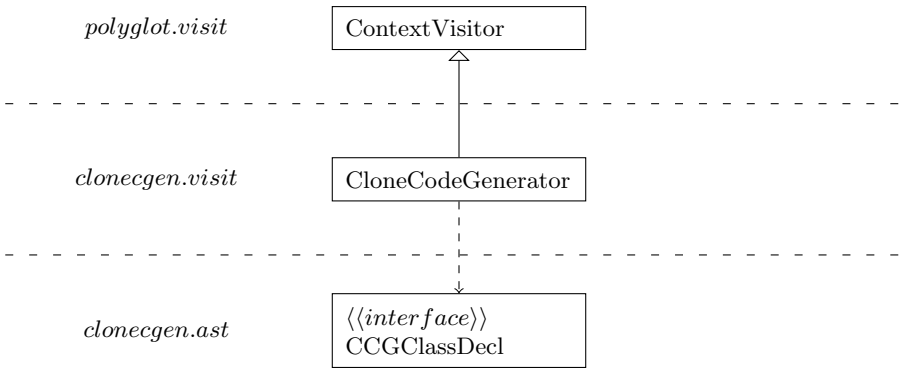


Figure 4.6: Class diagram of *CloneCodeGen*'s Visit package.

4.2 Generate Clone Code Pass

The compiler generates clone methods in the clone methods in the *GenerateCloneCode* pass and the majority of the work in performed in the class declaration AST nodes. In this section, I describe and give code examples of the key steps to generating the clone methods in the *GenerateCloneCode* pass.

Key steps of this pass are:

1. `GenerateCloneCode` object instructs class declaration nodes to generate clone methods when they are visited. This is achieved by overriding the `leaveCall` method in `GenerateCloneCode` (Listing 4.1). It passes itself as an argument so that the class declaration node can access the node factory, type system and context that `GenerateCloneCode` has. This is needed for creating new nodes and checking owner parameters of field declarations are in scope.

   Listing 4.2 is the implementation of the called method, which co-ordinates the next steps.

2. Check the owner parameters of field declarations are in scope because the implementation of clone methods depend on the owner parameters (Listing 4.2, lines 7 - 23). If there is an owner parameter that is not in scope, then an exception is thrown (lines 16 - 20).

3. Add `Cloneable` to the list of interfaces that the class declaration implements (Listing 4.2, line 26). The implementation of the called method is given in Listing 4.3, where a mutable copy of the existing list of interfaces is obtained (line 2) before the `Cloneable` interface is added in line 3 and set the interfaces in line 4.

4. Create and add method declaration nodes to the class declaration node (Listing 4.2, line 27). The implementation of the called method is in Listing 4.4, where:

   (a) Method declaration nodes are created and added to the body (lines 2 - 8). An example of how a method declaration node is created is given in Listing 4.5, which is the implementation of the method called in line 4.

   (b) Method instances (`MedthodDef` objects) corresponding to the method declarations are added to the type system (lines 10 - 11).

Listing 4.5 shows an example of creating a new method declaration node, this involves:

1. Constructing the AST nodes representing the body of the new method (lines 6 - 17).

2. Create the `MethodDecl` node by calling the node factory and passing as arguments a visibility flag node, the return type of the method, the name of the method, the formal parameters of the method, the exceptions that the method throws and the method body (lines 19 - 25).

3. Create and set the method instance object for the method declaration node (lines 27 - 31), which expects as arguments the types for the arguments passed in lines 22 - 25.

```
@Override
protected Node leaveCall(Node old, Node n, NodeVisitor v) throws SemanticException {
    if (n instanceof CCGClassDecl) {
        CCGClassDecl classNode = (CCGClassDecl)n;
        return classNode.createCloneMethod(this);
    }

    return n;
}
```

Listing 4.1: Code in `GenerateCloneCode` for creating clone methods when a class declaration node is visited.

```
@Override
public Node createCloneMethod(CloneCodeGenerator cloneCodeGenerator) throws
    SemanticException {
    CloneCGenTypeSystem ts = (CloneCGenTypeSystem) cloneCodeGenerator.typeSystem();
    CloneCGenNodeFactory nf = (CloneCGenNodeFactory) cloneCodeGenerator.nodeFactory();
    CCGContext context = (CCGContext) cloneCodeGenerator.context();

    List<ClassMember> classMembers = this.body.members();
    for(ClassMember member : classMembers) {
        if(member instanceof FieldDecl) {
            FieldDecl fd = (FieldDecl)member;
            TypeNode t = fd.type();
            if(t instanceof CCGAmbTypeNode) {
                CCGAmbTypeNode tn = (CCGAmbTypeNode) t;
                List<Id> fieldOwners = tn.ownerParams();

                for(Id owner : fieldOwners) {
                    if (!context.ownerParameterInThisScope(owner.id())) {
                        throw new NoOwnerParameterException(owner.id().toString(), owner.
                            position());
                    }
                }
            }
        }
    }

    CCGClassDecl_c cd = (CCGClassDecl_c) this.copy();
    cd = cd.addCloneableInterface(nf, ts);
    cd = (CCGClassDecl_c) cd.addCloneMethod(cloneCodeGenerator);
    return cd;
}
```

Listing 4.2: Method in class `CCGClassDecl_c` called by `GenerateCloneCode` to add clone code.

```
protected CCGClassDecl_c addCloneableInterface(CloneCGenNodeFactory nf,
    CloneCGenTypeSystem ts) {
    List<TypeNode> interfaces = TypedList.copyAndCheck(this.interfaces(), TypeNode.class,
        false);
    interfaces.add(nf.CanonicalTypeNode(this.position(), ts.Cloneable()));
    return (CCGClassDecl_c) this.interfaces(interfaces);
}
```

Listing 4.3: Implementation for adding an implemented interface to a class declaration.

```
protected CCGClassDecl_c addCloneMethod(CloneCodeGenerator cloneCodeGenerator) throws
    SemanticException {
    ClassBody newBody = body();

    MethodDecl cloneMethod = this.addTopMostCloneMethod(cloneCodeGenerator);
    MethodDecl cloneParamMethod = this.addParamCloneMethod(cloneCodeGenerator);

    newBody = newBody.addMember(cloneMethod);
    newBody = newBody.addMember(cloneParamMethod);

    this.type.addMethod(cloneMethod.methodDef());
    this.type.addMethod(cloneParamMethod.methodDef());

    return (CCGClassDecl_c) body(newBody);
}
```

Listing 4.4: Implementation for adding method declarations to the class and type system.

```
protected MethodDecl addTopMostCloneMethod(CloneCodeGenerator cloneCodeGenerator) throws
    SemanticException {
    CloneCGenTypeSystem ts = (CloneCGenTypeSystem) cloneCodeGenerator.typeSystem();
    CloneCGenNodeFactory nf = (CloneCGenNodeFactory) cloneCodeGenerator.nodeFactory();
    Position pos = position();

    Block valuesBlock = nf.Block(pos);
```

```
7
8       List<Expr> args = new LinkedList<Expr>();
9       for(int i = 0; i < this.ownerParams.size(); i++) {
10          args.add(nf.BooleanLit(pos, false));
11      }
12
13      Type hashmapType = ts.typeForName(QName.make("java.util.IdentityHashMap"));
14      args.add(nf.New(pos, nf.CanonicalTypeNode(pos, hashmapType), Collections.EMPTY_LIST));
15
16      Call call = nf.Call(this.position, nf.Id(pos,"clone"), args);
17      valuesBlock = valuesBlock.append(nf.Return(pos, call));
18
19      FlagsNode vmFlagsNode = nf.FlagsNode(pos, Flags.PUBLIC);
20      TypeNode returnType = nf.CanonicalTypeNode(pos, this.type.asType());
21      Id name = nf.Id(pos, "clone");
22      MethodDecl cloneMethod = nf.MethodDecl(
23          pos, vmFlagsNode, returnType,
24          name, Collections.EMPTY_LIST, Collections.EMPTY_LIST,
25          valuesBlock);
26
27      MethodDef md = ts.methodDef(pos, Types.ref(this.classDef().asType()),
28          vmFlagsNode.flags(), Types.ref(returnType.type()), Name.make("clone"),
29          Collections.EMPTY_LIST, Collections.EMPTY_LIST);
30
31      return cloneMethod.methodDef(md);
32  }
```

Listing 4.5: Implementation for creating method declaration node for the parameterless `clone` method.

4.3 Testing

11 test classes were written in the extended Java language to test the pre-processor that implements the basic cloning approach (without arrays as fields, subclassing and generics). For test classes that was expected to compile successfully, 22 unit tests were written using *JUnit* to test the generated clone methods behave as expected.

The *test.sh* and *pthScript* scripts provided by Polyglot were used to compile all the test classes into standard Java by executing *test.sh* once. The test files and the compiler error messages expected for each file were added to *pthScript* and when *test.sh* is executed, it reports whether each file compiled as expected.

These test classes are included in Appendix D and cover fringe cases:

- Class with no constructor.
- Class with no fields.
- Class with a field of a primitive type.
- Class contains reference types.
- Using `world` and `this` as an owner parameters.
- Cloning parameters of fields are in different orders.
- Importing and using classes from the Java Class Library (but not as fields).
- Use of arrays (aside from as fields).
- Calling and declaring methods.
- Making field assignments.
- Calling clone() in some of the code.
- Class is not compiled when there is a field with a cloning parameter that is not in scope.

Unit testing the clone behaviour involved testing each pair of clone methods with after creating objects with different structures, this is to test that the referenced objects are not cloned more than once. Different object instances were tested to cover cases where:

- Object makes a field reference to itself.
- Object's fields reference the same object.

Chapter 5

Evaluation

In this section, we evaluate this approach to ownership directed cloning by generating clone implementation for the basic generation of cloning code and for each feature extension.

5.1 Pre-processor and General Cloning Approach

The pre-processor and the basic cloning approach is evaluated in the following aspects to applicability, usability and correctness. The main issues are the applicability due to the small coverage of features, and using classes from the Java Class Library. Because the basic cloning approach was first proposed for a small subset of the Java language, this approach to cloning is still not yet applicable to writing real programs and annotating existing programs.

Applicability

A major issue related to the Java Class Library is that the pre-processor cannot compile any programs that use any class from the Java Class Library as a field (even the `String` class). This is because classes in the Java Class Library do not have the parametric cloning methods that this approach to cloning generates and uses. A potential solution is to extend the pre-processor so that it can generate methods or blocks of statements that should be called or inserted into the parametric cloning methods when cloning commonly used classes (such as `String`) when they are stored as fields.

Some investigation was carried out to see if this cloning approach was applicable to classes from the Java Class Library. For the basic cloning approach, I was unable to find classes that could be annotated with ownership types such that the implemented pre-processor could be used to generate cloning methods for them. The main reasons for this were classes used features that were not dealt with in the basic cloning approach: subclassing, interfacing, arrays, and static fields/classes. Otherwise, classes that this cloning approach was not applicable for, were used as fields in other classes.

From this investigation, I observed that there are methods with arguments that are reference types but these arguments do not get assigned to fields of any object. Adding cloning parameters to these arguments may be too restrictive when it does not really matter what the cloning parameters of the object is. For example, Listing 5.2 shows a method in `Formatter`, which takes a `String` as an argument but does not get assigned to any field.

Another observation from the investigation was very few fields will actually use `this` as an owner parameter because fields are often initialised in the constructor, where the values/objects are passed as arguments. If the object, o, has not been created yet, then it is not possible for objects with o as a cloning parameter to exist. A small example of this is the `FilterOutputStream` class (Listing 5.1).

```
1  class FilterOutputStream extends OutputStream {
2    /*
3     * The underlying output stream to be filtered.
4     */
5    protected OutputStream out;
6
7    public FilterOutputStream(OutputStream out) {
8      this.out = out;
9    }
10   ...
11 }
```

Listing 5.1: Class where fields are initialised with arguments passed in the constructor.

```
1  public final class Formatter implements Closeable, Flushable {
2    ...
3    private int width(String s) {
4      width = -1;
5      if (s != null) {
6        try {
7          width = Integer.parseInt(s);
8          if (width < 0)
9            throw new IllegalFormatWidthException(width);
10       } catch (NumberFormatException x) {
11         assert(false);
12       }
13     }
14     return width;
15   }
16   ...
17 }
```

Listing 5.2: Class where a method has a String argument that does not get stored as a field.

Usability

To apply this cloning approach to existing programs, all classes must be annotated with cloning parameters, this is difficult for large and complex programs and must not use features that have not been implemented yet by the pre-processor.

In contrast, I tried writing a small program using ownership types and the pre-processor to generate clone methods; and then compared this with writing the same program with clone implementation that achieves the same cloning behaviour by hand. The time and effort to do both was similar but the bulk of the reasons for this are different between the two cases, and the number of lines of code written were approximately the same.

The main difficultly with writing the program using ownership type was the lack of IDE support, implemented features (for example `List` could not be used) and taking care with which cloning parameters to give to fields. On the other hand, writing the clone implementation by hand that gives the same cloning behaviour took the most effort but wring the rest of the program was easier and less classes were created because the Java Class Library could be used.

In general, this cloning approach would be more usable if commonly used features are handled and classes from the Java Class Library can be used.

Correctness

The pre-processor successfully shows that cloning methods can be generated using ownership types and the testing validates that the generated clone methods behave as expected. However, limited type checking is implemented and checks for problematic situations are not performed, so there may be unexpected behaviours in cases where programs have invalid assignments and ownership structures.

5.2 Extensions to the Cloning Approach

Cloning without Owners-as-Dominators

To allow the use of iterators, we wanted to allow owners-as-dominators to be broken, I discovered without owners-as-dominators, sometimes cloning causes type errors. I formulated conditions that prevent the possibility of re-entering domain paths from occurring, whilst allowing owners-as-dominators to be broken if they do not create re-entering domain paths. There has been other approaches to allow owners-as-dominators to be broken in a controlled manner such as *Owners as Ombudsmen* [34], however, for our specific use of ownership types for cloning, the problematic situations are still possible.

Arrays

I proposed several solutions for extending this cloning approach to array objects, these solutions involved generating new cloning methods for cloning arrays specifically. Each proposed solution have the desired properties of cloning described in Section 3.1, when cloning an object o:

- All objects inside o are also cloned.
- All objects outside of o are not cloned.
- The clone is homomorphic to o (have the same shape).

- No dynamic information amount cloning parameters are stored in fields.

The selected solution for the arrays extension (without generics) was to generate an array clone method for each class and array dimension combination that appears in the program. The advantage of this solution over the other proposed solutions is Java reflection, which has slower performance, is not used. However, this solution can potentially generate many methods if many multi-dimensional arrays appear in a program.

Subclassing

The selected solution involves generating methods to override superclass' clone method and restricting assignments so that an object, o, cannot be assigned to a variable/field with static type, $t\langle\overline{ca}\rangle$, unless one of the following is true:

- o is an instance of a subclass of t and o has less or the same number of cloning parameters.
- o has dynamic type $t\langle\overline{ca}\rangle$.

Unlike of the approaches I proposed for extending cloning to deal with subclassing, this solution has all the desired cloning properties, but this means that some assignments are invalid even though the object is a valid subtype of the receiver. This is the case when subclass have more cloning parameters than the class it extends, it is not unlikely that this occurs when the superclass is very simple and subclasses declare new fields.

Generics

Extending this cloning approach to deal with generic classes had the problem that the cloning parameters of generic type parameters are not known. The solution I proposed requires a permutation-like order list (Section 3.2.2) to be added to each generic class to represent the indices of cloning parameters of each type parameter. Although it is desired for no dynamic information to be additionally stored, it is necessary in the case of generic types because the cloning parameters are unknown until an instance of the class is created. This is unless there is the restriction that generic type parameter's sequence of cloning parameters are restricted to be identical to the class's cloning parameters. The other three desired cloning properties hold and we are able to limit the dyamic information stored to only be for generic type parameters.

As a consequence of this solution, the combined extension (dealing with arrays, subclassing and generics) involves generating an additional generic array cloning method, which requires reflection to be used to create array clones since the actual type of array elements is not known.

Chapter 6

Conclusion

6.1 Achievements

Main achievements of this project are I have extended the cloning approach of generating ownership directed clone methods proposed in *Trust the Clones*[15]. I implemented the basic cloning approach as a Java language extension, using Polyglot. The resulting clone methods do not clone objects too deeply or too shallow, allowing programmers to annotate code with cloning (owner) parameters and relieves them from maintaining clone implementation directly.

The problems and solutions for extending this cloning approach for arrays, subclassing and generics have been explored. These extensions ensure the cloning policy is followed and aim to minimise the amount of dynamic information about ownership that is needed to be stored in the generated code standard Java. Permutation-like order lists are stored only when they are necessary for deriving cloning parameters of generic type parameters.

I have also investigated the problem and formulated the solution to preventing potentially problematic situations where breaking owners-as-dominators sometimes cause type errors when cloning. Our solution allows owners-as-dominators to be broken as long as problematic re-entering domain field paths are not possible.

6.2 Challenges/Difficulties

Challenges I experienced in this project were in using Polyglot, such as adding nodes to the AST, debugging and running the compiler. Developing and running using *Eclipse*, and making use of the attach source feature helped with the latter two problems. Existing documentation and many example Java language extensions using Polyglot helped solve problems with rewriting the AST.

Despite these difficulties, I would recommend using Polyglot to implement language extensions. The learning curve may be steep for Polyglot but after learning from practical experience and exploring existing examples, using Polyglot allows Java language extensions to be implemented without writing an immensely large amount of code.

6.3 Future Work

Further work in this area will help make this cloning approach more applicable to real programs. This includes extending cloning to deal with other features such as interfacing, static fields or classes, and investigating how to deal with uses of the Java Class Library (since these classes do not have parametric clone methods and many do not implement the Cloneable interface). We would also like to support existential/variable clone parameters so that this cloning approach is more permissive – this should not be a problem if objects with variable clone parameters are not stored as fields.

The pre-processor can be extended to generate cloning code for arrays, subclassing, generics and any further features explored. I would suggest using Polyglot to do this and developing in *Eclipse*, making use of the attach source and debugging features.

Appendices

Appendix A

Alternative Formalisation of Statically Prevent Problematic Case

Another formalisation of solution 2 for cloning without OAD described in Section 3.1.3, is given here, including the necessary syntax of field paths, relative position paths; auxiliary functions; and definition of well-formed classes.

Figure A.1 gives the syntax of relative position paths and field paths are extended to include repeated sequences of fields. A relative position path is a sequence of actual owner parameters (relative to some object), which may have infinitely repeated subsequence due to an infinitely repeated sequence in the corresponding field path. The asterisks ($*$) indicate an infinitely repeated sequence of fields and owner parameters in field paths and relative position paths, respectively. In addition, the functions $first$ and $last$ are defined for extracting the first and last owner from a relative position path.

$$
\begin{array}{rr}
p ::= \texttt{this} \mid \texttt{this}.fp & \text{Field Path} \\
fp ::= f* \mid (f.fp)* \mid f \mid f.fp & \text{Many fields} \\
rpp ::= ca \mid ca.rpp \mid rrp* & \text{Relative Position Path}
\end{array}
$$

$$
\begin{aligned}
first &: RelativePositionPath \rightarrow ActualOwner \\
first(ca) &= ca \\
first(ca.rpp) &= ca \\
first(rpp*) &= \text{undefined}
\end{aligned}
$$

$$
\begin{aligned}
last &: RelativePositionPath \rightarrow ActualOwner \\
last(ca) &= ca \\
last(cs.rpp) &= last(rpp) \\
last(rpp*) &= \text{undefined}
\end{aligned}
$$

Figure A.1: Syntax for Field Path and Relative Position Path; and definitions of auxiliary functions.

Other auxiliary functions, $allPaths$, $relPosPath$ and $hasRDP$ are defined in Figure A.2, the following is a description of each function:

allPaths
: Returns a set of field paths starting from the given *path* (which currently references an object instance of class C with actual owner parameters $\overline{ca}$), given a program, P. The set includes all possible paths except for those that traverse a sequence of fields (through the same objects) more

than once, however, paths that end with an infinitely repeated sequence of fields indicated with $*$ are included.

In some cases, objects in infinitely repeated sequence of fields do not have the same type, such as when the owner parameters of a field in the sequenced is `this`. However, this means that objects in the infinite loop will become more and more nested inside, I propose that if a RDP can occur from a nested object then a RDP should also be able to occur from an object in the first instance of the repeated sequence.

Arguments:

- P is the current program.
- C is the class of the object referenced by *path*
- $\overline{ca}$ is the actual owner parameters of the object referenced by *path*
- TP is a mapping from types of visited objects to the path that led to the visited object. This keeps track of visited objects and enables cycles to be detected.
- *path* is the field path constructed so far and is being extended further till no fields are available or we visit the same class and actual owners again (i.e. the same object).

Constructing the set of paths involves recursive calls to *allPaths* for each field, f, in class of C and passing the extended path, updated visited objects mapping, and updated currently referenced class and actual owner parameters. The actual owner parameters are found by applying a substitution to the owner parameters of the field as declared in class C. This substitution is replacing the appearances of formal owner parameters of C with the actual parameters, *ca*.

relPosPath

Returns the RPP corresponding to the first given field path, p_1, by recursively calling *relPosPath* to build the RPP as fields in p_1 are visited.

Arguments:

- p_1 is the target field path, whose corresponding RPP is to be returned.
- p_2 is the field path processed so far.
- *rpp* is the RPP so far (corresponding to p_2). When $p_1 = p_2$, this *rpp* is returned.
- PT is a mapping from field paths (processed so far) to the types of their objects. This mapping enables us to find the relative object position of the next object referenced by a field by looking up the formal owner parameters of fields and substituting them with actual parameters from the object referenced by p_2.

For repeated sequences of fields, we make a recursive call to *relPos* supposing there is only 1 instance of the sequence, $\overline{f}*$. From the returned RPP, we extract the subsequence, $\overline{rp}$, that represents $\overline{f}$. We can then indicate $\overline{rp}$ is a repeating sequence in the final RPP.

hadRDP

Returns whether the given RPP, *rpp*, corresponds to a field path is a RDP or contains once. This inspects the first 2 owners, ca_1 and ca_2, in *rpp* if available; if ca_1 is outside `this` and ca_2 is `this` then *rpp* contains a RDP; otherwise a recursive call is made with ca_1 removed. For the repeated sequence case, we check whether there is a RDP from the last owner to the first owner in the repeated sequence, rpp_1, and whether there is a RDP in rpp_1 itself. If the end of *rpp* is reached without finding a RDP, then there is no RDP and false is returned.

$$allPaths : (Program \times ClassId \times ActualOwners \times (Type \to Path) \times Path) \to \mathcal{P}(Path)$$

$$allPaths(P, C, \overline{ca}, TP, path) = \begin{cases} p_1.(\overline{f}*) & \text{if } TP(C\langle\overline{ca}\rangle) \text{ is defined} \\ \quad \text{where } p_1 = TP(C\langle\overline{ca}\rangle) \text{ and } path = p_1.\overline{f} & \\ \{path\} & \text{if } Fs(P,C) = \emptyset \\ \bigcup \forall f \in Fs(P,C) : & \text{Otherwise} \\ allPaths(P, C_1, \overline{ca_1}, TP_1, p_1) & \\ \quad \text{where } C_1\langle\overline{fo}\rangle = F(P,C,f) & \\ \quad \text{and } \overline{ca_1} = \overline{fo}[\overline{ca}/_{O(P,C)}] & \\ \quad \text{and } TP_1 = TP[C\langle\overline{ca}\rangle \mapsto path] & \\ \quad \text{and } p_1 = path.f & \end{cases}$$

$$relRosPath : (Path \times Path \times (Path \to Type) \times RelativePositionPath) \to RelativePositionPath$$

$$relRosPath(p_1, p_2, PT, rpp) = \begin{cases} rpp & \text{if } p_1 = p_2 \\ relPos(p_1, p_2.f, PT', rpp') & \text{if } p_1 = p_2.f.\overline{fl} \\ \quad \text{where } C\langle\overline{ca}\rangle = PT(p_2) & \\ \quad \text{and } A\langle\overline{a}\rangle = F(P,C,f) & \\ \quad \text{and } ca_1, \overline{ca_2} = \overline{a}[\overline{ca}/_{O(P,C)}] & \\ \quad \text{and } rpp' = rpp.ca_1 & \\ \quad \text{and } PT' = PT[p_2.f \mapsto A\langle ca_1, \overline{ca'}\rangle] & \\ rpp.(\overline{rp}*) & \text{if } p_1 = p_2.(\overline{f}*) \\ \quad \text{where } rpp.\overline{rp} = relPos(p_2.\overline{f}, p_s, PT, rpp) & \end{cases}$$

$$hasRDP : RelativePositionPath \to Boolean$$

$$hasRDP(rpp) = \begin{cases} ca_1 \neq \texttt{this} \wedge ca_2 = \texttt{this} & \text{if } rpp = ca_1.ca_2 \\ hasRDP(ca_1.ca_2) \vee hasRDP(ca_2.rpp') & \text{if } rpp = ca_1.ca_2.rpp' \\ hasRDP(ca.first(rpp_1)) \vee hasRDP(rpp_1*) & \text{if } rpp = ca.(rpp_1*) \\ hasRDP(last(rpp_1).first(rpp_1)) \vee hasRDP(rpp_1) & \text{if } rpp = rrp_1* \\ false & \text{otherwise} \end{cases}$$

Figure A.2: Auxiliary functions for finding all paths from a class instance (*AllPaths*), calculating the RPP of a given path (*relPosPath*) and determining whether a given RPP corresponds to a RDP (*hasRDP*).

The conditions of a well-formed class is extended to require all possible field paths from an instance of the class are not RDP's: Using the functions given in Figure A.2, we extend the conditions of a well-formed class C in program P ($ClssW_{P,C}$), with a condition that checks whether an instance of a class could reference objects through a RDP. This condition is given in Figure A.3, where the set of paths to check is calculated and for each path, p, we require that it does not contain a RDP.

$$ClssWF_P(C) = \begin{cases} \text{Original conditions of well-formed class} \\ \text{and} \\ \forall p \in AllPaths(P, C, O(P, C), [], \texttt{this}) : \\ \text{not } hasRDP(relRosPath(p, \texttt{this}, [\texttt{this} \rightarrow C\langle O(P, C)\rangle], \varepsilon)) \end{cases}$$

Figure A.3: Definition of well-formed class extended to prevent the possibility of RDP's.

To demonstrate, we will now use the formalisation to detect a RDP for the program with the class declarations in Listing 3.1, which we have previously shown may give rise to RDP's:

$$\begin{aligned} ClssWF_P(A) =& \forall p \in AllPaths(P, A, O(P, A), [], \texttt{this}) : \\ & \text{not } hasRDP(relRosPath(p, \texttt{this}, [], \varepsilon)) \\ =& \forall p \in AllPaths(P, A, c, [], \texttt{this}) : \\ & \text{not } hasRDP(relRosPath(p, \texttt{this}, [], \varepsilon)) \end{aligned}$$

$$\begin{aligned} AllPaths(P, A, c, [], \texttt{this}) =& AllPaths(P, B, \texttt{this}, [A\langle c\rangle \mapsto \texttt{this}], \texttt{this.fb}) \\ & \cup AllPaths(P, C, (c, \texttt{this}), [A\langle c\rangle \mapsto \texttt{this}], \texttt{this.fc}) \\ =& \{\texttt{this.fb}\} \\ & \cup AllPaths(P, B, \texttt{this}, [A\langle c\rangle \mapsto \texttt{this}, C\langle c, \texttt{this}\rangle], \texttt{this.fc.fb}) \\ =& \{\texttt{this.fb}\} \\ & \cup \{\texttt{this.fc.fb}\} \\ =& \{\texttt{this.fb}, \texttt{this.fc.fb}\} \end{aligned}$$

$$\begin{aligned} & relPosPath(\texttt{this.fb}, \texttt{this}, [\texttt{this} \mapsto A\langle c\rangle], \varepsilon) \\ & \quad = relPosPath(\texttt{this.fb}, \texttt{this.fb}, [\texttt{this} \mapsto A\langle c\rangle, \texttt{this.fb} \mapsto B\langle \texttt{this}\rangle], \texttt{this}) \\ & \quad = \texttt{this} \end{aligned}$$

$$\begin{aligned} & relPosPath(\texttt{this.fc.fb}, \texttt{this}, [\texttt{this} \mapsto A\langle c\rangle], \varepsilon) \\ & \quad = relPosPath(\texttt{this.fc.fb}, \texttt{this.fc}, [\texttt{this} \mapsto A\langle c\rangle, \texttt{this.fc} \mapsto C\langle c, \texttt{this}\rangle], c) \\ & \quad = relPosPath(\texttt{this.fc.fb}, \texttt{this.fc.fb}, [\texttt{this} \mapsto A\langle c\rangle, \texttt{this.fc} \mapsto C\langle c, \texttt{this}\rangle, \texttt{this.fc.fb} \mapsto B\langle \texttt{this}\rangle], c.\texttt{this}) \\ & \quad = c.\texttt{this} \end{aligned}$$

$$\begin{aligned} hasRDP(\texttt{this}) &= false \\ hasRDP(c.\texttt{this}) &= true \end{aligned}$$

From class A, there is a RPP that corresponds to a RDP, hence A is not a well-formed class.

Appendix B

Detecting Re-entering Domain Paths Attempts

In this section, I describe other attempts at detecting RDP's and were found to be insufficient. These are based on the observation that objects can only reference objects owned by objects represented by the formal owner parameters of their class, `this` and `world`. Hence, RDP's depend on the actual owner parameters in field declarations and their relative position to the current class. Field referenced objects can only make an ICBR if the owner of objects that breaks owners-as-dominators is passed as an additional actual owner parameter for a field. For example, given class declaration in Listing B.1, we know that for objects of class `A`, `this.fa` is outside of `this` since its owner is `c` and `this.fa` can reference an object owned by `this` because `this` is passed as the second owner parameter of the field `fa`. Hence, instances of class `A` may have RDP's if fields are not null.

The attempts made to detect and prevent RDP's from occurring are:

1. *Restrict use of* `this` *owner parameter* – If the first actual owner parameter of a field declaration is not `this` then disallow `this` to appear as additional actual owner parameters for that field. This is not a sufficient condition, which will be illustrated by an example where a RDP can occur without passing `this` as additional owner parameters. The reason for this is that it only prevents RDP's from occurring immediately from the current class – within a path, p, the violating references to an object outside and then back into the domain of the current class may still occur later in p.

2. *Restrict use of the owner of the class* – In addition to Attempt 1, if the first actual owner parameter of a field declaration is not `this`, then disallow the owner of the current class to appear as additional actual owner parameters for that field. This is not a sufficient condition, which will be illustrated by an example where the RDP involves another object, o, in which the path has references from an object inside o to an object outside and then back into o.

```
1  class A<c> {
2      B<c,this> fa;
3      B<this,this>fa2;
4  }
5
6  class B<c1,c2> {
7      C<c2> fb;
8  }
9
10 class C<c> {
11 }
```

Listing B.1: Example class declarations where for `A` objects, `this.fa` is outside of `this` but `this.fa.fb` is inside `this`.

Attempt 1: Restrict use of `this` owner parameter

As illustrated from the previous example, a RDP can occur if `this` is passed as extra owner parameters to fields whose owner is not `this` (and hence guaranteed to be outside the domain of the current class). Therefore, fields not owned by `this` should not have `this` as an additional owner parameter. No

problem occurs for fields that have `this` as its owner, such as `fa2`, because the referenced object is inside `this` and should be allowed to reference other objects owned by `this`.

Figure B.1 shows how this condition can be expressed formally. Class `A` is not a well-formed class because its field `fb` is not well-formed.

However, this is not a sufficient condition because RDP's can occur even without having to pass `this`, such as for the class declarations in Listing B.2. Figure B.2 shows how these classes can be instantiated to give a RDP `this.fb.fa.fa` from $o1$ to $o2$ via $o3$. To also prevent such cases, *Attempt 2* was made.

$$FieldWF(C\langle ca_1, ..., ca_n\rangle f;) = ca_1 \neq \texttt{this} \rightarrow \forall ca \in \{ca_2, ..., ca_n\} : ca \neq \texttt{this}$$

Figure B.1: Condition for well-formed fields – restricts appearance of `this`.

```
1 class A<c1,c2> {
2     A<c2,c1> fa;
3 }
4
5 class B<c> {
6     A<this,c> fb;
7 }
```

Listing B.2: Example class declarations where RDP can occur without passing `this` as an extra owner parameter.

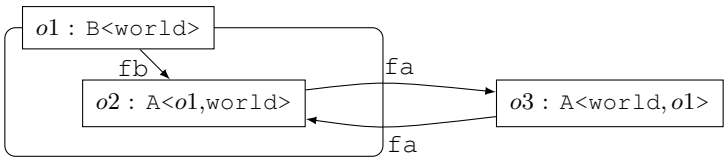


Figure B.2: Example of object instances of classes in Listing B.2, where there is a RDP.

Attempt 2: Restrict use of the owner of the class

This extends *Attempt 1* with an additional condition that fields not owned by `this` should not have the owner of the current class as an additional owner parameter. For Listing B.2, this condition does not hold for class `A` since the owner of field `fa` is outside of the current class and `c1`, the owner of the class, is passed as the second owner parameter of the field.

Figure B.3 shows how this extended condition can be expressed formally.

However, this is not a sufficient condition because an RDP, p, can occur when it involves another object, o, and p has contains references from an object inside o to an object outside and then back into o. Figure B.4 illustrates this diagrammatically and the corresponding class declarations are given in Listing B.3. The class declarations satisfy the condition but from Figure B.4, we can see that there is a RDP from $o2$ to $o6$ (inside $o2$) via $o5$, which is outside $o2$.

This suggests that every step of any possible path from instances of a class have to be checked for RDP's and this led to Solution 2 detailed in Section 3.1.3.

$$\begin{aligned} FieldWF_{P,C_1}(C\langle ca_1, ..., ca_n\rangle f;) \; &= ca_1 \neq \texttt{this} \rightarrow \forall ca \in \{ca_2, ..., ca_n\} : (ca \neq \texttt{this} \text{ and } ca \neq c_1) \\ &\quad \text{where } \texttt{class } C_1\langle c_1, ...c_k\rangle\{...\} \text{ is defined} \end{aligned}$$

Figure B.3: Condition for well-formed fields of class C_1 in program P – restricts appearance of `this` and the owner of the class.

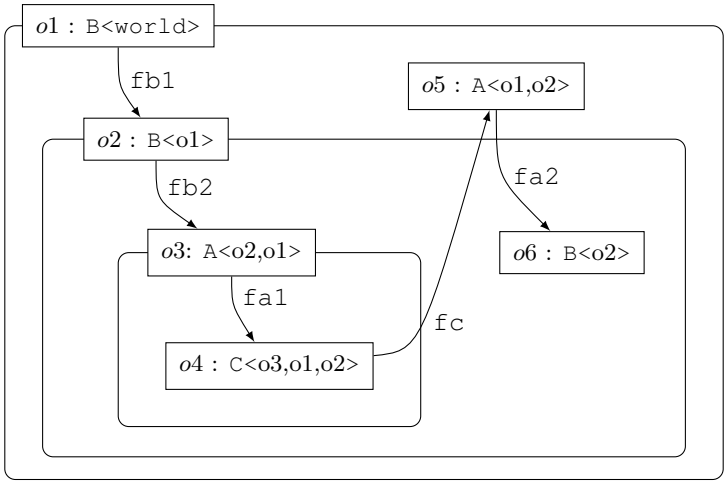


Figure B.4: Example of object instances of classes in Listing B.2, where there is a RDP.

```
1  class A<c1,c2> {
2      C<this,c1,c2> fa1;
3      B<c2>fa2;
4  }
5
6  class B<c> {
7      B<this> fb1;
8      A<this,c> fb2;
9  }
10
11 class C<c1,c2,c3> {
12     A<c2,c3> fc;
13 }
```

Listing B.3: Example class declarations where for `A` objects, `this.fa` is outside of `this` but `this.fa.fb` is inside `this`.

Appendix C

Formal Characterisations of Situations with Re-entering Domain Paths

In this section, I show how the the two subcases of situations with RDP's (described in Section 3.1.1) can be characterised; this is optional and readers may skip this chapter. Given than there is a RDP from $o1$ to $o2$, the two situations are:

(a) *Only RDP's exist* – all paths from $o1$ to $o2$ are RDP's.

(b) *RDP's and internal paths from* $o1$ *to* $o2$ *exist* – there is a path, p, from $o1$ to $o2$ where all objects referenced in p are in the cloning domain (we call p an *internal path*) as well as another path that is a RDP.

Situations A and B can be characterised formally in terms of the heap and environment as $SitA^*$ and $SitB^*$ respectively, defined as:

$$\begin{aligned} SitA^*(o,\chi,\Gamma) = \exists p, f'.[&\Gamma \not\vdash \chi(o,p) \prec^* o \text{ and } \Gamma \vdash \chi(o,p.f') \prec^* o \text{ and} \\ &\forall p''.(\chi(o,p'') = \chi(o,p.f') \implies \exists p_1,p_2.p'' = p_1p_2 \text{ and} \\ &\Gamma \not\vdash \chi(o,p_1) \prec^* o)] \end{aligned}$$

$$\begin{aligned} SitB^*(o,\chi,\Gamma) = \exists p, f'.[&(\Gamma \not\vdash \chi(o,p) \prec^* o \text{ and } \Gamma \vdash \chi(o,p.f') \prec^* o) \\ &\implies \\ &(\exists p''.\chi(o,p'') = \chi(o,p.f') \text{ and} \\ &\forall p_1,p_2.p'' = p_1p_2 \implies \Gamma \vdash \chi(o,p_1) \prec^* o)] \end{aligned}$$

In English, $SitA^*$ says that an object, $o1$, in the heap, χ, has a path to an object ($o3$) outside of it, which has a field to an object inside $o1$ but cannot be reached by $o1$ via a field. The combination of paths and fields used in this definition is sufficient because for situation A to hold, the minimum requirement is that:

- There must be an object, $o3$, outside $o1$ that can be reached by some path from $o1$.
- $o3$ references an object, $o2$, inside $o1$.
- There is no path from $o1$ to this object ($o2$) that does not go outside of $o1$ – i.e. every path from $o1$ to $o2$ goes outside of $o1$ at some point.

$SitB^*$ says that if the object, $o1$, in the heap has a path to an object ($o3$) outside of it and $o3$ has a field to an object inside $o1$, then $o1$ must have a path, p'', to that object where every step of p'' stays inside $o1$. Similar to $SitA^*$, the combination of fields and paths in this definition is sufficient.

Variants of $SitA^*$ and $SitB^*$. Other versions of these properties were considered before choosing the final formalisation. These include first characterising the concrete examples and then generalising them (with different combinations of field references and paths between $o1$, $o3$ and $o2$) to capture all cases that fall into Situation A and B.

The concrete examples in Figure 3.1a (an example of Situation A) can be detected with the property $SitA_1$ and Figure 3.1b (an example of Situation B) can be detected with the property $SitB_1$, defined as:

$$\begin{aligned} SitA_1(o,\chi,\Gamma) = \exists f, f'.(\Gamma \not\vdash \chi(o)(f) \prec^* o \text{ and } \Gamma \vdash \chi(\chi(o)(f))(f') \prec^* o \text{ and} \\ \neg\exists f''.\chi(o)(f'') = \chi(\chi(o)(f))(f')) \end{aligned}$$

$$\begin{aligned} SitB_1(o,\chi,\Gamma) = \exists f, f'.[(\Gamma \not\vdash \chi(o)(f) \prec^* o \text{ and } \Gamma \vdash \chi(\chi(o)(f))(f') \prec^* o) \\ \implies \exists f''.\chi(o)(f'') = \chi(\chi(o)(f))(f')] \end{aligned}$$

In English, $SitA_1$ says that the object at o in the heap, o, has a field to an object (o') outside of its ownership, which has a field to an object under the ownership of o but cannot be reached by o via a field. $SitB_1$ says that if the object at o in the heap has a field to an object (o') outside of its ownership, which has a field to an object under the ownership of o, then o must have a field to that object.

The properties can be extended from fields to paths (via fields of objects):

$$\begin{aligned} SitA_2(o,\chi,\Gamma) = \exists p, p'.[\Gamma \not\vdash \chi(o,p) \prec^* o \text{ and } \Gamma \vdash \chi(\chi(o,p),p') \prec^* o \text{ and} \\ \forall p''.(\chi(o,p'') = \chi(\chi(o,p),p') \implies \exists p_1, p_2.p'' = p_1p_2 \text{ and} \\ \Gamma \not\vdash \chi(o,p_1) \prec^* o)] \end{aligned}$$

$$\begin{aligned} SitB_2(o,\chi,\Gamma) = \exists p, p'.[(\Gamma \not\vdash \chi(o,p) \prec^* o \text{ and } \Gamma \vdash \chi(\chi(o,p),p') \prec^* o) \\ \implies \exists p''.\chi(o,p'') = \chi(\chi(o,p),p')] \end{aligned}$$

There can be a path of any length, as long as for all paths from object $o1$ to object $o2$, there is an object that is not owned by $o1$, in the path; then the situation will be detected by $SitA_2$.

It can be seen that $SitA_1 \not\Leftrightarrow SitA_2$, the counter example is given by figure C.1. Here, there is a path from object 1 to object 4, which is not owned by 1, and a path from 4 to object 2, which is owned by 1, but there is no path to from 1 to 2 that does not go outside of 1's ownership. $SitA_1$ fails to detect this at 1, however, $SitA_2$ detects this situation as required.

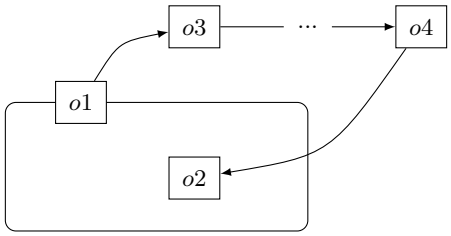


Figure C.1: Example of a path from $o1$ to $o2$ that is more than 2 fields away and goes outside of $o1$'s ownership.

Other versions of $SitA_2$ explored, using different combinations of fields and paths in the properties:

1. If p is just a field, f:

$$\begin{aligned} SitA_3(o,\chi,\Gamma) = \exists f, p'.[\Gamma \not\vdash \chi(o)(f) \prec^* o \text{ and } \Gamma \vdash \chi(\chi(o)(f),p') \prec^* o \text{ and} \\ \forall p''.(\chi(o,p'') = \chi(\chi(o)(f),p') \implies \exists p_1, p_2.p'' = p_1p_2 \text{ and} \\ \Gamma \not\vdash \chi(o,p_1) \prec^* o)] \end{aligned}$$

   This definition is not sufficient. $SitA_3$ fails to detect cases where the only path from an object $o1$ to $o2$ (is owned by $o1$) goes outside the ownership of $o1$, but it only does so at a later step in the path. Such a case is illustrated in Figure C.2.

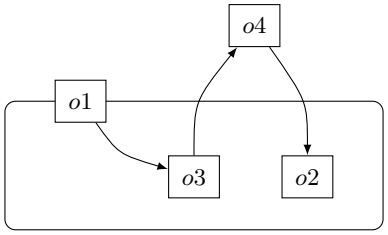


Figure C.2: Example of a path from $o1$ to $o2$ that stays within $o1$'s ownership in the first step of the path.

2. If p'' is just a field, f'':

$$SitA_5(o, \chi, \Gamma) = \exists p, p'.[\Gamma \nvdash \chi(o, p) \prec^* o \text{ and } \Gamma \vdash \chi(\chi(o, p), p') \prec^* o \text{ and } \neg\exists f''.\chi(o)(f'') = \chi(\chi(o, p), p')]$$

$SitA_5$ will hold for some cases that should be be classified as Situation B. For example, where for some object $o1$ and object $o2$, whose owner is $o1$: there is a path from $o1$ to $o2$ that goes outside the ownership of $o1$ and also a path from $o1$ to $o2$ that stays within the ownership of $o1$. This is illustrated in Figure C.3.

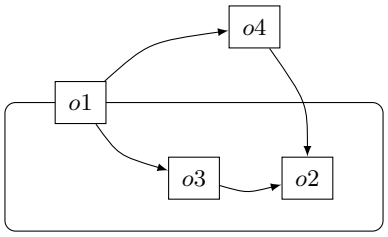


Figure C.3: Example of where there is path from $o1$ to $o2$ that stays within $o1$'s ownership and one that does not.

Other versions of $SitB_2$ use the same combinations as the variants of $SitA_2$, and are insufficient for the same reasons.

Appendix D

Test Classes

This section contain the test classes used to test the pre-processor. For tests expected to compile, the compiled class is also given.

Test class and compiled code for a class that uses an array (but not as a field).

```
1  package compilerTestClasses;
2
3  /*
4   * Test class that uses an array
5   */
6  public class Hello<c1,c2> {
7      public Hello() {}
8
9      public static void main(String<world
           >[]<c1> args) {
10         System.out.println("Hello world!");
11   String<c1> a;
12     }
13 }
```

Listing D.1: Test class that uses an array (but not stored as a field).

```
1  package compilerTestClasses;
2
3  public class Hello implements Cloneable {
4
5      public Hello() { super(); }
6
7      public static void main(String[] args)
           {
8          System.out.println("Hello world!");
9          String a;
10     }
11
12     public Hello clone() {
13         return this.clone(false, false, new
               java.util.IdentityHashMap());
14     }
15
16     public Hello clone(boolean b0, boolean
           b1, java.util.IdentityHashMap map)
           {
17         Object o = map.get(this);
18         if (o != null)
19             return (Hello) o;
20         else {
21             Hello oClone = new Hello();
22             map.put(this, oClone);
23             return oClone;
24         }
25     }
26
27     final public static String
           jlc$CompilerVersion$jl = "3.2.0";
28     final public static long
           jlc$SourceLastModified$jl =
           1339503813000L;
29     final public static String
           jlc$ClassType$jl = ...
30 }
```

Listing D.2: Generated Java class for Listing D.1.

Test class that contains no fields.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class with no fields
5  */
6 public class Simple<c> {
7     public Simple() {}
8 }
```

Listing D.3: Test class that has no fields.

```
1 package compilerTestClasses;
2
3 public class Simple implements Cloneable {
4
5     public Simple() { super(); }
6
7     public Simple clone() {
8         return this.clone(false, new java.
              util.IdentityHashMap());
9     }
10
11    public Simple clone(boolean b0, java.
          util.IdentityHashMap map) {
12        Object o = map.get(this);
13        if (o != null)
14            return (Simple) o;
15        else {
16            Simple oClone = new Simple();
17            map.put(this, oClone);
18            return oClone;
19        }
20    }
21
22    final public static String
          jlc$CompilerVersion$jl = "3.2.0";
23    final public static long
          jlc$SourceLastModified$jl =
          1338383864000L;
24    final public static String
          jlc$ClassType$jl = ...
25 }
```

Listing D.4: Generated Java class for Listing D.3.

Test class that contains a field of primitive type.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class with primitive type field
5  * and imports.
6  */
7 public class Simple2<c1,c2> {
8     public int f1;
9
10    public Simple2() {
11        this.f1 = 0;
12    }
13 }
```

Listing D.5: Test class that has a field that is of a primitive type.

```
1 package compilerTestClasses;
2
3 public class Simple2 implements Cloneable {
4   public int f1;
5
6   public Simple2() {
7     super();
8     this.f1 = 0;
9   }
10
11  public Simple2 clone() {
12    return this.clone(false, false, new
          java.util.IdentityHashMap());
13  }
14
15  public Simple2 clone(boolean b0, boolean
        b1,
16    java.util.IdentityHashMap map) {
17    Object o = map.get(this);
18    if (o != null)
19      return (Simple2) o;
20    else {
21      Simple2 oClone = new Simple2();
22      map.put(this, oClone);
23      oClone.f1 = this.f1;
24      return oClone;
25    }
26  }
27
28  final public static String
        jlc$CompilerVersion$jl = "3.2.0";
29  final public static long
        jlc$SourceLastModified$jl =
        1338383896000L;
30  final public static String
        jlc$ClassType$jl = ...
31 }
```

Listing D.6: Generated Java class for Listing D.5.

Test class that contains fields of primitive and reference types, and methods that uses a class from the Java Class Library.

```
1  package compilerTestClasses;
2
3  import java.lang.Integer;
4
5  /*
6   * Test class with a mix of primitive and
        reference type fields,
7   * method declarations and use of imported
        class from Java Class Library
8   */
9  public class Simple3<c1,c2> {
10   public Simple3<c1,c2> f1;
11
12   public int f2;
13
14   public Simple3() {
15     this.f1 = null;
16     this.f2 = 0;
17   }
18
19   public void setf1(Simple3<c1,c2> f1) {
20     this.f1 = f1;
21   }
22
23   public Simple3<c1,c2> getf1() {
24     return this.f1;
25   }
26
27   public int echo(String<world> line) {
28     try {
29       this.f2 = Integer.parseInt(line);
30     } catch (NumberFormatException<world> e
           ) {
31       this.f2++;
32     }
33
34     return this.f2;
35   }
36 }
```

Listing D.7: Test class that has primitive and reference type fields; methods and uses Integer from the Java Class Library.

```
1  package compilerTestClasses;
2
3  import java.lang.Integer;
4
5  public class Simple3 implements Cloneable {
6    public Simple3 f1;
7    public int f2;
8
9    public Simple3() {
10     super();
11     this.f1 = null;
12     this.f2 = 0;
13   }
14
15   public void setf1(Simple3 f1) { this.f1 =
         f1; }
16
17   public Simple3 getf1() { return this.f1;
       }
18
19   public int echo(String line) {
20     try {
21       this.f2 = Integer.parseInt(line);
22     }
23     catch (NumberFormatException e) { this.
         f2++; }
24     return this.f2;
25   }
26
27   public Simple3 clone() {
28     return this.clone(false, false, new
         java.util.IdentityHashMap());
29   }
30
31   public Simple3 clone(boolean b0, boolean
       b1,
32             java.util.IdentityHashMap map)
                 {
33     Object o = map.get(this);
34     if (o != null)
35       return (Simple3) o;
36     else {
37       Simple3 oClone = new Simple3();
38       map.put(this, oClone);
39       oClone.f1 = this.f1 != null && b0
40               ? (Simple3) this.f1.clone(b0,
                      b1, map)
41               : this.f1;
42       oClone.f2 = this.f2;
43       return oClone;
44     }
45   }
46
47   final public static String
         jlc$CompilerVersion$jl = "3.2.0";
48   final public static long
         jlc$SourceLastModified$jl =
         1339968401567L;
49   final public static String
         jlc$ClassType$jl = ...
50 }
```

Listing D.8: Generated Java class for Listing D.7.

Test class that uses `world` as the owner parameter of a field.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class using world as an owner of a
       field
5  */
6 public class Simple4<c1,c2> {
7   public int f1;
8
9   public Simple4<c2,c2> f2;
10
11  public Simple4<world,c2> f3;
12
13  public Simple4() {
14    this.f1 = 0;
15    this.f2 = null;
16    this.f3 = null;
17  }
18 }
```

Listing D.9: Test class that has `world` as an owner parameter of a field.

```
1 package compilerTestClasses;
2
3 public class Simple4 implements Cloneable {
4   public int f1;
5   public Simple4 f2;
6   public Simple4 f3;
7
8   public Simple4() {
9     super();
10    this.f1 = 0;
11    this.f2 = null;
12    this.f3 = null;
13  }
14
15  public Simple4 clone() {
16    return this.clone(false, false, new
          java.util.IdentityHashMap());
17  }
18
19  public Simple4 clone(boolean b0, boolean
        b1,
20            java.util.IdentityHashMap map)
                {
21    Object o = map.get(this);
22    if (o != null)
23      return (Simple4) o;
24    else {
25      Simple4 oClone = new Simple4();
26      map.put(this, oClone);
27      oClone.f1 = this.f1;
28      oClone.f2 = this.f2 != null && b1
29              ? (Simple4) this.f2.clone(b1,
                    b1, map)
30              : this.f2;
31      oClone.f3 = this.f3;
32      return oClone;
33    }
34  }
35
36  final public static String
        jlc$CompilerVersion$jl = "3.2.0";
37  final public static long
        jlc$SourceLastModified$jl =
        1338384187000L;
38  final public static String
        jlc$ClassType$jl = ...
39 }
```

Listing D.10: Generated Java class for Listing D.9.

Test class that uses `this` as the owner parameter of a field.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class with this as a owner of a
       field
5  */
6 public class Simple5<c1,c2> {
7   public int f1;
8
9   // Hence, this.f2.f3 is owned by this
10  public Simple5<this,c2> f2;
11
12  // Owned by owner of 'this'
13  public Simple5<c1,world> f3;
14
15  public Simple5() {
16    this.f1 = 0;
17    this.f2 = null;
18    this.f3 = null;
19  }
20 }
```

Listing D.11: Test class that has `this` as an owner parameter of a field.

```
1 package compilerTestClasses;
2
3 public class Simple5 implements Cloneable {
4   public int f1;
5   public Simple5 f2;
6   public Simple5 f3;
7
8   public Simple5() {
9     super();
10    this.f1 = 0;
11    this.f2 = null;
12    this.f3 = null;
13  }
14
15  public Simple5 clone() {
16    return this.clone(false, false, new
          java.util.IdentityHashMap());
17  }
18
19  public Simple5 clone(boolean b0, boolean
        b1,
20            java.util.IdentityHashMap map)
                {
21    Object o = map.get(this);
22    if (o != null)
23      return (Simple5) o;
24    else {
25      Simple5 oClone = new Simple5();
26      map.put(this, oClone);
27      oClone.f1 = this.f1;
28      oClone.f2 = this.f2 != null
29              ? (Simple5) this.f2.clone(
                    true, b1, map)
30              : this.f2;
31      oClone.f3 = this.f3 != null && b0
32              ? (Simple5) this.f3.clone(b0,
                    false, map)
33              : this.f3;
34      return oClone;
35    }
36  }
37
38  final public static String
        jlc$CompilerVersion$jl = "3.2.0";
39  final public static long
        jlc$SourceLastModified$jl =
        1338384245000L;
40  final public static String
        jlc$ClassType$jl = ...
41 }
```

Listing D.12: Generated Java class for Listing D.11.

Test class that has fields that are instances of other user defined classes.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class with reference types that are
       other classes (other than this class)
5  */
6 public class Simple6<c1,c2> {
7   public Simple4<this,c2> f1;
8
9   public Simple5<c1,world> f2;
10
11  public Simple6() {
12    this.f1 = null;
13    this.f2 = null;
14  }
15 }
```

Listing D.13: Test class that has other user defined classes as fields.

```
1 package compilerTestClasses;
2
3 public class Simple6 implements Cloneable {
4   public Simple4 f1;
5   public Simple5 f2;
6
7   public Simple6() {
8     super();
9     this.f1 = null;
10    this.f2 = null;
11  }
12
13  public Simple6 clone() {
14    return this.clone(false, false, new
          java.util.IdentityHashMap());
15  }
16
17  public Simple6 clone(boolean b0, boolean
        b1,
18                java.util.IdentityHashMap map)
                      {
19    Object o = map.get(this);
20    if (o != null)
21      return (Simple6) o;
22    else {
23      Simple6 oClone = new Simple6();
24      map.put(this, oClone);
25      oClone.f1 = this.f1 != null
26               ? (Simple4) this.f1.clone(
                     true, b1, map)
27               : this.f1;
28      oClone.f2 = this.f2 != null && b0
29               ? (Simple5) this.f2.clone(b0,
                      false, map)
30               : this.f2;
31      return oClone;
32    }
33  }
34
35  final public static String
        jlc$CompilerVersion$jl = "3.2.0";
36  final public static long
        jlc$SourceLastModified$jl =
        1338384305000L;
37  final public static String
        jlc$ClassType$jl = ...
38 }
```

Listing D.14: Generated Java class for Listing D.13.

Test class that uses has a constructor that expects parameters.

```
1 package compilerTestClasses;
2
3 /*
4  * Class with constructor that expects
       arguments
5  */
6 public class Simple7<c1,c2> {
7   public Simple4<this,c2> f1;
8
9   public Simple5<c1,world> f2;
10
11   public Simple7(Simple4<this,c2> f1,
         Simple5<c1,world> f2) {
12     this.f1 = f1;
13     this.f2 = f2;
14   }
15 }
```

Listing D.15: Test class that has a constructor with some parameters.

```
1 package compilerTestClasses;
2
3 public class Simple7 implements Cloneable {
4   public Simple4 f1;
5   public Simple5 f2;
6
7   public Simple7(Simple4 f1, Simple5 f2) {
8     super();
9     this.f1 = f1;
10    this.f2 = f2;
11  }
12
13  public Simple7() { super(); }
14
15  public Simple7 clone() {
16    return this.clone(false, false, new
          java.util.IdentityHashMap());
17  }
18
19  public Simple7 clone(boolean b0, boolean
        b1,
20            java.util.IdentityHashMap map)
                {
21    Object o = map.get(this);
22    if (o != null)
23      return (Simple7) o;
24    else {
25      Simple7 oClone = new Simple7();
26      map.put(this, oClone);
27      oClone.f1 = this.f1 != null
28              ? (Simple4) this.f1.clone(
                    true, b1, map)
29              : this.f1;
30      oClone.f2 = this.f2 != null && b0
31              ? (Simple5) this.f2.clone(b0,
                     false, map)
32              : this.f2;
33      return oClone;
34    }
35  }
36
37  final public static String
        jlc$CompilerVersion$jl = "3.2.0";
38  final public static long
        jlc$SourceLastModified$jl =
        1339501151000L;
39  final public static String
        jlc$ClassType$jl = ...
40 }
```

Listing D.16: Generated Java class for Listing D.15.

Test class that calls the `clone` method of another object.

```
1  package compilerTestClasses;
2
3  /*
4   * Calls clone on another object
5   */
6  public class Simple11<c> {
7    public Simple11<c> twin;
8
9    public void makeTwin(Simple11<c> twin) {
10     this.twin = twin.clone();
11   }
12 }
```

Listing D.17: Test class that calls the clone method of another object.

```
1
2  public class Simple11 implements Cloneable
       {
3    public Simple11 twin;
4
5    public void makeTwin(Simple11 twin) {
         this.twin = twin.clone(); }
6
7    public Simple11() { super(); }
8
9    public Simple11 clone() {
10     return this.clone(false, new java.util.
           IdentityHashMap());
11   }
12
13   public Simple11 clone(boolean b0, java.
         util.IdentityHashMap map) {
14     Object o = map.get(this);
15     if (o != null)
16       return (Simple11) o;
17     else {
18       Simple11 oClone = new Simple11();
19       map.put(this, oClone);
20       oClone.twin = this.twin != null && b0
21               ? (Simple11) this.twin.clone(
                     b0, map)
22               : this.twin;
23       return oClone;
24     }
25   }
26
27   final public static String
         jlc$CompilerVersion$jl = "3.2.0";
28   final public static long
         jlc$SourceLastModified$jl =
         1340081199930L;
29   final public static String
         jlc$ClassType$jl = ...
30 }
```

Listing D.18: Generated Java class for Listing D.17.

Test class that has 2 fields that may reference the same object.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class with 2 fields that can
       reference the same object
5  */
6 public class Simple12<c1,c2> {
7   public Simple12<c1,c2> f1;
8   public Simple12<c1,c2> f2;
9 }
```

Listing D.19: Test class that has 2 fields that may reference the same object.

```
1 package compilerTestClasses;
2
3 public class Simple12 implements Cloneable
      {
4   public Simple12 f1;
5   public Simple12 f2;
6
7   public Simple12() { super(); }
8
9   public Simple12 clone() {
10    return this.clone(false, false, new
          java.util.IdentityHashMap());
11  }
12
13  public Simple12 clone(boolean b0, boolean
        b1,
14              java.util.IdentityHashMap map
                  ) {
15    Object o = map.get(this);
16    if (o != null)
17      return (Simple12) o;
18    else {
19      Simple12 oClone = new Simple12();
20      map.put(this, oClone);
21      oClone.f1 = this.f1 != null && b0
22              ? (Simple12) this.f1.clone(b0
                    , b1, map)
23              : this.f1;
24      oClone.f2 = this.f2 != null && b0
25              ? (Simple12) this.f2.clone(b0
                    , b1, map)
26              : this.f2;
27      return oClone;
28    }
29  }
30
31  final public static String
        jlc$CompilerVersion$jl = "3.2.0";
32  final public static long
        jlc$SourceLastModified$jl =
        1340083168492L;
33  final public static String
        jlc$ClassType$jl = ...
34 }
```

Listing D.20: Generated Java class for Listing D.19.

Test class that has a field with a cloning parameter that does not exist.

```
1 package compilerTestClasses;
2
3 /*
4  * Clone parameter c2 for field f1 does not exist
5  */
6 public class Simple8<c> {
7   public Simple4<this,c2> f1;
8 }
```

Listing D.21: Test class where the clone parameter `c2` of field `f1` does not exist.

Test class where less cloning parameters are given to a field that its type expects.

```
1 package compilerTestClasses;
2
3 /*
4  * Test class where not enough cloning parameters are
5  * given to the field
6  */
7 public class Simple9<c> {
8   public Simple6<c> f1;
9 }
```

Listing D.22: Test class where not enough cloning parameters are given to a field.

Bibliography

[1] Jaco. http://lamp.epfl.ch/~zenger/jaco/.

[2] Jastadd. http://jastadd.org/web/.

[3] Java with parameterized types. http://www.cs.cornell.edu/polyj/.

[4] Marwan Abi-Antoun and Jonathan Aldrich. Ownership domains in the real world. In Tobias Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, 2007. Available at http://www.cs.purdue.edu/homes/wrigstad/iwaco/.

[5] Daniel Lee Andrew Jonas and Andrew Myers. J0: A java extension for beginning (and advanced) programmers. http://www.cs.cornell.edu/Projects/j0.

[6] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 56–69, New York, NY, USA, 2001. ACM.

[7] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 324–337, New York, NY, USA, 2003. ACM.

[8] Michael Brukman and Andrew C. Myers. Ppg, parser generator for extensible grammars. http://www.cs.cornell.edu/Projects/polyglot/ppg.html.

[9] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 441–460, New York, NY, USA, 2007. ACM.

[10] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[11] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *European Conference for Object-Oriented Programming (ECOOP*, pages 176–200. Springer-Verlag, 2003.

[12] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64. ACM Press, 1998.

[14] David Gerard Clarke. *Object ownership and containment*. PhD thesis, New South Wales, Australia, Australia, 2003.

[15] Sophia Drossopoulou and James Noble. Trust the Clones. In *FoVEOOS - preproceedings*, September 2011.

[16] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[17] Peter Grogono and Markku Sakkinen. Copying and comparing: Problems and solutions. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 226–250, London, UK, 2000. Springer-Verlag.

[18] Scott Hudson. Cup, lalr parser generator in java. `http://www2.cs.tum.edu/projects/cup/`.

[19] Yuuji Ichisugi. The extensible java pre-processor epp homepage. `http://staff.aist.go.jp/y-ichisugi/epp/`.

[20] Thomas P. Jensen, Florent Kirchner, and David Pichardie. Secure the clones - static enforcement of policies for secure object copying. In *ESOP'11*, pages 317–337, 2011.

[21] Paley Li, Nicholas Cameron, and James Noble. Cloning in ownership. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 63–66, New York, NY, USA, 2011. ACM.

[22] Todd Millstein and Milan Stanojevic. Polyglot for java 5. `http://www.cs.ucla.edu/~todd/research/polyglot5.html`.

[23] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for java. In *Principles of Programming Languages (POPL)*, pages 132–145, 1997.

[24] Steve Zdancewic Andrew Myers Stephen Chong Nate Nystrom, Lantian Zheng and K. Vikram. Jif: Java information flow. `http://www.cs.cornell.edu/jif/`.

[25] Sun Developer Network. Secure coding guidelines for the java programming language, version 4.0. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`.

[26] James Noble, David Clarke, and John Potter. Object ownership for dynamic alias protection. In *Proceedings TOOLS '99*, pages 176–187. Society Press, 1999.

[27] James Noble and Brian Foote. Attack of the clones. In *Proceedings of the 2002 conference on Pattern languages of programs - Volume 13*, CRPIT '02, pages 99–114, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[28] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP'98*, pages 158–185. Springer-Verlag, 1998.

[29] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[30] Johan Östlund and Tobias Wrigstad. Welterweight java. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 97–116, Berlin, Heidelberg, 2010. Springer-Verlag.

[31] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 311–324, New York, NY, USA, 2006. ACM.

[32] Raoul-Gabriel Urma. Swapj: An introduction to polyglot. `http://www.cs.cornell.edu/projects/polyglot/doc/swapJ-tutorial.pdf`.

[33] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *ECOOP*, pages 54–78, 2007.

[34] Tobias Wrigstad and Johan Östlund. Owners as ombudsmen : Multiple aggregate entry points for ownership types. 2011.