Alias-based Reasoning for Object-Oriented Programs

Bernd Schoeller
bernd.schoeller@inf.ethz.ch
Chair of Software Engineering
Eidgenössische Technische Hochschule, Zürich, Switzerland

November 18, 2005

Abstract

Aliasing is the key problem that makes reasoning about reference structures hard. Large predicates have to be constructed that capture all aliasing properties of a given state. Instead of deducing the aliasing properties from a state that uses heaps and objects, we declare the alias-relation to be the state itself. We explore if such a state model provides a new and beneficial approach to the verification of object-oriented programs. As a demonstration, we introduce a small reference language SOL and describe an axiomatic semantics for it. Then we prove a non-trivial program using this semantics.

1 Introduction

Software verification promises to overcome the current software crisis of bug-ridden software. Object-orientation is the most widely used paradigm for software development. One major hurdle [10] on the way towards the verification of object-oriented software is aliasing [8]. Aliasing is created by the use of references (also known as pointers) and dynamic memory management — two concepts that are mandatory for object-oriented development.

Formal reasoning (on software) is a process that develops theories about a (most of the time) textual representation of a program to predict the effect that this program produces when run on a physical machine. Software verification uses formal reasoning to relate the program to a formal specification by telling if the program will satisfy the specification during execution.

To achieve this goal, formal reasoning requires a theory describing the semantics of the programming language. This theory should describe effects based on the constructs of the language. To simplify proving, it is necessary to minimize the size of the theory. This is specially true for concepts that are not part of the syntax.

Many semantic descriptions of object-oriented programming languages introduce concepts such as heaps, arrays, stacks, objects that are not part of the syntax ([13], [7], [1]). These concepts are introduced to evaluate the main problem of references: aliasing. Aliasing is the phenomenon that two different names (textual artefacts) refer to the same storage location during execution. By changing the value associated with one name, the value of the other name changes as well. As a consequence, aliasing complicates reasoning about the program.

The purpose of the theory presented in this paper is to explore the possibility to directly talk about the aliasing properties of a given state without using stacks, heaps or objects. We introduce the state as an equivalence relation on names and chains of names (called *paths*) and present axioms that work on this state model. The idea of using such a structure for

$$
\begin{array}{lcl}
\mathit{prog} & \triangleq & \texttt{skip} \\
 & | & \mathit{prog}\ \texttt{;}\ \mathit{prog} \\
 & | & \mathit{path}\ \texttt{:=}\ \mathit{path} \\
 & | & \texttt{create}\ \mathit{path} \\
 & | & \texttt{if}\ \mathit{pred}\ \texttt{then}\ \mathit{prog}\ \texttt{else}\ \mathit{prog}\ \texttt{end} \\
 & | & \texttt{until}\ \mathit{pred}\ \texttt{loop}\ \mathit{prog}\ \texttt{end}
\end{array}
$$

Figure 1: Syntax of SOL

pointer analysis was pioneered by Jonkers [9]. The we show how to verify programs with references using the axioms.

2 SOL: a simple object-oriented language

In the following section, we will introduce SOL, a "simple object-oriented language". It concentrates on the main problems of pointer computation: pointer swing and object creation. Though it lacks common constructs (arithmetic, subroutines), it is sufficiently complex to describe many standard algorithms that work on pointer structures and expose aliasing.

2.1 Syntax

The syntax of SOL is based on IMP [18], a minimal imperative language used in different tutorials and libraries. IMP offers loops, conditionals and assignments. As we are dealing only with objects, the assignment operator has been redefined to handle arbitrary pointer swings. A new construct for object creation is added to the language.

The language syntax has been changed slightly to resemble a syntax closer to Eiffel [11], the main object of our research (SOL is not a subset of Eiffel as we allow foreign assignments like `a.x := y`). The loop has been changed from a `while`-`do`-`end`-loop into a `until`-`loop`-`end`-loop. Object creation uses the syntax "`create` *path*" ("*path* `:=` `new`" would be the Java-like equivalent).

The syntax of SOL is described in figure 1. The construct *path* is defined in section 2.2. The construct $pred$ is a *shallow embedding* of arbitrary predicates of our logic into the language.

2.2 Roots, Attributes and Paths

We introduce now the concept of *roots*, *attributes* and *paths*. We motivate paths by referencing to more "classical" structures like *objects* and *storage locations*. This makes it easier to understand the semantics of the new concepts. The theory itself does neither have the concept of storage location nor of object.

The idea is to describe objects based on how they can be accessed. Every object that is accessible in an object structure can be accessed by starting from some root and then following a number of attributes. The root and the sequence of attributes to access an object is called a *path*.

A *path* is a sequence of identifiers, thus a sequence of words. The identifiers are separated by dots. Each identifier describes a storage location holding the reference to an object. The first identifier in a path describes a *global storage location*. Such an identifier is called a *root*. Each additional identifier describes a *relative storage location*; a storage location that is relative to an object. Such an identifier is called an *attribute*.

An example for a path might be:

`self.first.right.next`

Assuming that we have two disjoint sets of identifiers *root* and *attribute*. All paths start with a root, followed by a finite set of attributes. A *path* is defined by the following recursive data-structure:

$$path \triangleq root \mid path.attribute$$

Path are not defined in terms of some state model. Paths are purely syntactical constructs. The state will later be described in terms of paths.

In the following we will use $p, q, p_1, p', \ldots$ for logical variables of paths. $a, b, a', \ldots$ will be used for attributes. r, r_1, r' will be used for roots. Concrete elements of $root$ and $attribute$ are written in type-font e.g. `first` or `self`.

2.3 State model

The state during the execution of an object-oriented program can be described as an equivalence relation on paths:

$$\equiv_s : path \times path \rightarrow bool$$

This equivalence relation describes which paths lead to the same object and which lead to different objects. From the perspective of storage locations, $a \equiv_s b$ means that both storage locations referenced by a and b contain the same value.

The state model is based on two observations: The main property of an object is its identity and only reachable objects matter (non-reachable objects are normally called garbage and automatically freed by a garbage collector).

The first observation makes it possible to capture the state as an equivalence relation. The second observation ensures that the full state can be captured by the equivalence relation: for every reachable object there has to be a path that leads to this object.

In addition to the rules for equivalence relations (transitivity, symmetry, reflexivity), the state has to fulfill the rule of "path extension" (as described by Morris [12]):

$$p_1 \equiv_s p_2 \Rightarrow p_1.a \equiv_s p_2.a$$

This axiom ensures that a single attribute describes a well-defined function from one equivalence class (meaning object) to another.

There are two derived, state-dependant relations between paths: attribute equivalence and path influence.

2.4 Attribute Equivalence

Conceptually paths lead to objects. But they also describe attributes: the last attribute of the path. It is the attribute changed by the assignment to the path. For example, `first.right.item` leads to the second item stored in a linked list. But it also describes the attribute `item` of the cell `first.right` and an assignment to this attribute `first.right.item := x` will change exactly this attribute.

Attribute equivalence is very similar to path equivalence, except that it describes if two paths denote the same attribute/root and not the same object. It is defined as follows:

$$\begin{aligned} r_1 \asymp_s r_2 &\triangleq r_1 = r_2 \\ p_1.a \asymp_s p_2.b &\triangleq (a = b) \wedge (p_1 \equiv_s p_2) \\ r_1 \asymp_s p_1.a &\triangleq false \end{aligned}$$

As seen by this definition, only the attribute equivalence of two non-roots is state dependent, which will be very helpful during proving. Attribute equivalence is also an equivalence relation and also allows path extension ($p_1 \asymp_s p_2 \Rightarrow p_1.a \asymp_s p_2.a$).

2.5 Path Influence and Independence

Path influence is a state-dependent relation between two paths. It describes whether an assignment to path q may change the target of the path p. This is true if any attribute used in p is final attribute of q. Writing $p \succ q$ reads as: after an assignment to p, the object that q is referencing may not be the same.

The relation can be described by the following primitive recursive definition:

$$\begin{aligned} p \succ_s r &\triangleq p \asymp_s r \\ p \succ_s q.a &\triangleq (p \asymp_s q.a) \vee (p \succ_s q) \end{aligned}$$

Path influence and specially its negation called *path independence* ($p \not\succ_s q$) is an important property to describe aliasing conditions.

3 Axiomatic Semantics

The axiomatic semantics for the language is based on the classical description by Hoare [6]. We only highlight the differences.

3.1 Assignment Axioms

The effect of the assignment is that certain paths lead to new objects. This effect can be described as a computation of a new state from a given state. Such a description may lead to the introduction of very complex and impractical formulas (see [7]).

Fortunately in the context of an axiomatic semantics, there is no need to fully describe the effect of a swing on the state. Instead, the effect can be described on the basis of the given predicates.

Having an equivalence relation as a state, there are two possible basic predicates: equivalence ($p_1 \equiv_s p_2$) and its negation ($p_1 \not\equiv_s p_2$). If we want to prove the Hoare-Triplet $\{P\}$ `p := q` $\{Q\}$, then the $Q \models p_1 \equiv_s p_2$ can either hold

1. as it held before the assignment ($P \models p_1 \equiv_s p_2$) and was not affected by the assignment or

2. as it was created by the assignment.

Using a pessimistic approach, we know that the equivalence is not destroyed if both paths (p_1 and p_2) were not affected by the assignment. For the two primitive predicates, we get the following axioms:

$$\frac{}{\{q \not\succ p_1 \wedge q \not\succ p_2 \wedge p_1 \equiv p_2\} \quad \texttt{q := x} \quad \{p_1 \equiv p_2\}}$$

$$\frac{}{\{q \not\succ p_1 \wedge q \not\succ p_2 \wedge p_1 \not\equiv p_2\} \quad \texttt{q := x} \quad \{p_1 \not\equiv p_2\}}$$

Also, the assignment creates an equivalence. We give two axioms for this case: one for the assignment to a root and the other for the assignment to foreign attribute:

$$\frac{}{\{p \equiv p \wedge r \not\succ p\} \quad \texttt{r := p} \quad \{p \equiv r\}}$$

$$\frac{}{\{q.a \not\succ q_1 \wedge q.a \not\succ p_1 \wedge q \equiv q_1 \wedge p \equiv p_1\} \quad \texttt{q.a := p} \quad \{q_1.a \equiv p_1\}}$$

3.2 Object Creation

Again we have two axioms for equivalences that are not affected by the creation:

$$\overline{\{q \not\succ p_1 \wedge q \not\succ p_2 \wedge p_1 \equiv p_2\} \quad \texttt{create q} \quad \{p_1 \equiv p_2\}}$$

$$\overline{\{q \not\succ p_1 \wedge q \not\succ p_2 \wedge p_1 \not\equiv p_2\} \quad \texttt{create q} \quad \{p_1 \not\equiv p_2\}}$$

Reasoning on the basis of paths makes it easy to describe the effect of object creation: the creation of an object makes a path different from all paths that do not lead over the attribute that the created object was attached to.

$$\overline{\{true\} \quad \texttt{create q} \quad \{p \asymp q \Leftrightarrow p \equiv q\}}$$

3.3 Soundness

The soundness of the approach has been proven manually against a classical operational semantics for SOL using stacks and heaps. Manual proves have been done for the soundness of the assignment and the object creation axiom. As a next step, we will implement both semantics in the formal prove environment Isabelle [15] and create an automated proof.

3.4 Method Invocation

In the context of aliasing, a modification clause [14] is needed. We assume that we already have this modification clause, although we have not described how to derive the modification clause from the implementation. The modification clause is a set of paths MOD_f. This set describes the paths that may have changed by the method.

We are also talking about two equivalence relations. The equivalence relation of the called method is $\equiv_f$, the equivalence relation of the calling method is $\equiv_c$.

The deduction rule is:

$$\frac{\begin{array}{c}\forall x'', a'' : \{P[Current \rhd x'', arg \rhd a'']\} x''.f(a'') \{Q[Current \rhd x'', arg \rhd a'']\} \vdash \\ \{P\} Body_f \{Q\} \wedge x \equiv x' \wedge a \equiv a' \wedge \forall p \in MOD_f : p \not\succ x' \wedge p \not\succ a'\end{array}}{\{P[Current \rhd x', arg \rhd a']\} x.f(a) \{Q[Current \rhd x', arg \rhd a']\}}$$

The substitution rule $P[x \rhd y]$ is a path prefix substitution. Every path starting with x is replaced by a path starting with y in the predicate P.

By showing that the references used in the pre- and postcondition of the method invocation are not influenced by the method call, we are preventing aliasing problems.

4 Proving List-Reversal

After defining an axiomatic semantics for SOL based on path properties, we explore how this semantics behaves when proving code. In-place list reversal is an algorithm that works on a pointer structure to create a reversed version of a linked list by reusing the cells of the linked list.

In-place list reversal is a simple, but non-trivial algorithm. It works by traversing through a linked list using three cursors (called first, next and previous, see figure 3). The implementation in SOL is captured in figure 2. The listing assumes a normal linked list with `first` pointing to the first cell. The cells themselves are connected with an attribute `right`. Cells lead to values using the `item` attribute (omitted in figure 3).

Beyond `first`, there are three other roots necessary for the algorithm. `next` and `previous` are temporary variables needed for the traversal. `Void` (`null` in Java) is a

```
next := first
first := Void
until next = Void loop
    previous := first
    first := next
    next := next.right
    first.right := previous
end
```

Figure 2: Implementation of list reversal

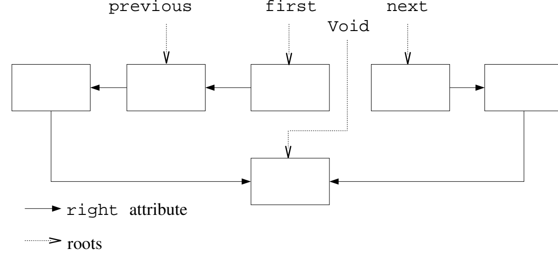

Figure 3: State during the execution

special root pointing to a distinguished object that is different from the cells stored in the list.

The boolean expression `next = Void` should actually be written as $\lambda S.\mathtt{next} \equiv_S \mathtt{Void}$, following the rules of the shallow embedding. We use the more programming-language like construct to improve readability.

4.1 Abbreviations and Definitions

To keep the proof brief, we will introduce a number of abbreviations and definitions. First, we use an exponent to describe a repeated attribute ($\mathtt{first.right}^3$ instead of `first.right.right.right`). Formally, this notation is defined by the following primitive recursive definition:

$$\begin{aligned} p.a^0 &\triangleq p \\ p.a^{n+1} &\triangleq p.a.a^n \end{aligned}$$

We want to verify the property that the list pointed to by `first` will be reversed after the execution of the above algorithm. This is a relation between a pre- and a post-state. The values of the pre-state will be access by add an index 0 (like in $\mathtt{first}_0$).

Let *count* be the length of the list. We assume that we have a cycle-free, Void-terminated linked list. We define a class-invariant INV.

$$\begin{aligned} INV \triangleq\ & \forall n, m \in 0 \ldots count : n \neq m \Rightarrow \mathtt{first.right}^n \not\equiv \mathtt{first.right}^m \wedge \\ & \mathtt{first.right}^{count} \equiv \mathtt{Void} \end{aligned}$$

The list will be split into two lists, one still in the right order and one in the reverse order. Both lists are Void-terminated and do not share cells. Given two lists starting with p_1 and p_2 and the lengths c_1 and c_2, we get the following property:

$$\begin{aligned} NC(p_1, c_1, p_2, c_2) \triangleq\ & \forall n, m \in 0 \ldots c_1 : n \neq m \Rightarrow p_1.\mathtt{right}^n \not\equiv p_1.\mathtt{right}^m \wedge \\ & p_1.\mathtt{right}^{c_1} \equiv \mathtt{Void} \wedge \\ & \forall n, m \in 0 \ldots c_2 : n \neq m \Rightarrow p_2.\mathtt{right}^n \not\equiv p_2.\mathtt{right}^m \wedge \\ & p_2.\mathtt{right}^{c_2} \equiv \mathtt{Void} \wedge \\ & \forall n \in 0 \ldots c_1 - 1, m \in 0 \ldots c_2 - 1 : p_1.\mathtt{right}^n \not\equiv p_2.\mathtt{right}^m \end{aligned}$$

4.2 Specifications on Paths

We have to specify that the list-reversal algorithm as presented here does indeed produce a reversed version of the list as it was before. We limit ourselves to sketch this proof in

the paper. Other properties to prove are that the class-invariant is reestablished and that the algorithm has no Void calls.

We access the elements by a function seq:

$$seq(n) = \texttt{first.right}^n\texttt{.item}$$

Equally, we define a function seq_0 for the pre-state:

$$seq_0(n) = \texttt{first}_0\texttt{.right}_0^n\texttt{.item}_0$$

With these abbreviations, specifying list reversal is straight forward. Before the execution, both functions lead to the same object:

$$PRE \triangleq \forall i \in 0 \ldots count - 1 : seq(i) \equiv seq_0(i)$$

After the execution, the list is reversed:

$$POST \triangleq \forall i \in 0 \ldots count - 1 : seq(i) \equiv seq_0(count - i - 1)$$

4.3 Loop Invariant

As a loop invariant, we first make sure that the list is split up into two sublists, the part of the list starting with `next` that is still in the old order and the part of the list starting with `first`, that is in the new order.

$$LI_1(i) \triangleq NC(\texttt{next}, count - i, \texttt{first}, i)$$

The two different parts of the list contain different subparts of the original linked list:

$$LI_2(i) \triangleq \forall j \in 0..count - i - 1 : \texttt{next.right}^j\texttt{.item} \equiv seq_0(i + j)$$
$$LI_3(i) \triangleq \forall j \in 0..i - 1 : \texttt{first.right}^j\texttt{.item} \equiv seq_0(i - j - 1)$$

The full loop-invariant is

$$LI \triangleq \exists i.LI_1(i) \wedge LI_2(i) \wedge LI_3(i)$$

4.4 Proof Outline

From the class-invariant and the precondition, we have to establish the loop-invariant.

$\{INV \wedge PRE\}$

```
next := first
```

$\{INV \wedge PRE \wedge next \equiv first\}$

$\{\forall n, m \in 0 \ldots count - 1 : n \neq m \Rightarrow \texttt{next.right}^n \not\equiv \texttt{next.right}^m \wedge$
$\texttt{next.right}^{count} \equiv \texttt{Void} \wedge$
$\forall i \in 0..count - 1 : \texttt{next.right}^i\texttt{.item} \equiv seq_0(i)\}$

```
first := Void
```

$\{\forall n, m \in 0 \ldots count : n \neq m \Rightarrow \texttt{next.right}^n \not\equiv \texttt{next.right}^m \wedge$
$\texttt{next.right}^{count} \equiv \texttt{Void} \wedge$
$\forall i \in 0..count - 1 : \texttt{next.right}^i\texttt{.item} \equiv seq_0(i) \wedge first \equiv Void\}$

$\{NC(next, count, first, 0) \wedge$
$\forall i \in 0 \ldots count : \texttt{next.right}^i\texttt{.item} \equiv seq_0(i)\}$

$\{LI_1(0) \wedge LI_2(0) \wedge LI_3(0)\}$

$\{LI\}$

Next we have to prove that the loop invariant is retained by the body of the algorithm:

$\{LI\}$

```
previous := first
```

$\{LI \wedge \texttt{previous} \equiv \texttt{first}\}$

$\{NC(\texttt{next}, count - i, \texttt{previous}, i) \wedge$
$\forall j \in 0..count - i - 1 : \texttt{next.right}^j\texttt{.item} \equiv seq_0(i + j) \wedge$
$\forall j \in 0..i - 1 : \texttt{previous.right}^j\texttt{.item} \equiv seq_0(i - j - 1)\}$

```
first := next
```

$\{NC(\texttt{next}, count - i, \texttt{previous}, i) \wedge$
$\forall j \in 0..count - i - 1 : \texttt{next.right}^j\texttt{.item} \equiv seq_0(i + j) \wedge$
$\forall j \in 0..i - 1 : \texttt{previous.right}^j\texttt{.item} \equiv seq_0(i - j - 1) \wedge$
$\texttt{first} \equiv \texttt{next}\}$

$\{NC(\texttt{first}, count - i, \texttt{previous}, i) \wedge$
$\forall j \in 0..count - i - 1 : \texttt{first.right}^j\texttt{.item} \equiv seq_0(i + j) \wedge$
$\forall j \in 0..i - 1 : \texttt{previous.right}^j\texttt{.item} \equiv seq_0(i - j - 1) \wedge$
$\texttt{first.right} \equiv \texttt{next.right}\}$

```
next := next.right
```

$\{NC(\texttt{first}, count - i, \texttt{previous}, i) \wedge$
$\forall j \in 0..count - i - 1 : \texttt{first.right}^j\texttt{.item} \equiv seq_0(i + j) \wedge$
$\forall j \in 0..i - 1 : \texttt{previous.right}^j\texttt{.item} \equiv seq_0(i - j - 1)\} \wedge$
$\texttt{first.right} \equiv \texttt{next}\}$

$\{NC(\texttt{next}, count - i - 1, \texttt{previous}, i) \wedge$
$\forall j \in 0..count - i - 2 : (\texttt{next.right}^j\texttt{.item} \equiv seq_0(i + j + 1) \wedge \texttt{first} \not\equiv \texttt{next.right}^j) \wedge$
$\forall j \in 0..i - 1 : (\texttt{previous.right}^j\texttt{.item} \equiv seq_0(i - j - 1) \wedge first \not\equiv \texttt{previous.right}^j) \wedge$
$\texttt{first.item} \equiv seq_0(i) \wedge \texttt{first.right} \equiv \texttt{next}\}$

```
first.right := previous
```

$\{NC(\texttt{next}, count - i - 1, \texttt{previous}, i) \wedge$
$\forall j \in 0..count - i - 2 : (\texttt{next.right}^j\texttt{.item} \equiv seq_0(i + j + 1) \wedge first \not\equiv \texttt{next.right}^j) \wedge$
$\forall j \in 0..i - 1 : (\texttt{previous.right}^j\texttt{.item} \equiv seq_0(i - j - 1) \wedge first \not\equiv \texttt{previous.right}^j)$
$\wedge \texttt{first.item} \equiv seq_0(i) \wedge \texttt{first.right} \equiv \texttt{next} \wedge \texttt{first.right} \equiv \texttt{previous}\}$

$\{NC(\texttt{next}, count - i - 1, \texttt{first}, i + 1) \wedge$
$\forall j \in 0..count - i - 2 : \texttt{next.right}^j\texttt{.item} \equiv seq_0(i + j + 1) \wedge$
$\forall j \in 0..i : \texttt{first.right}^j\texttt{.item} \equiv seq_0(i - j)\}$

$\{LI_1(i + 1) \wedge LI_2(i + 1) \wedge LI_3(i + 1)\}$

$\{LI\}$

After the termination of the loop, we know that $LI \wedge \texttt{next} = \texttt{Void}$ holds. According to LI_1, this can only hold if $i = count$. So we can derive:

$$LI \wedge i = count \Rightarrow$$

$$LI_1(count) \wedge LI_2(count) \wedge LI_3(count) \Rightarrow$$

$$\forall j \in 0..count - 1 : \texttt{first.right}^j\texttt{.item} \equiv seq_0(count - j - 1)$$

This is the required postcondition.

4.5 Remarks

The property of paths influence and independence has to be deduced from the formulas. Though this is not hard, it still requires many small proofs and is skipped for brevity. These proofs would be intermediate steps and lemmas in an interactive proving environment.

All proofs for assignments to local variables were trivial and followed the same strategy: first the predicate P was changed to a predicate P' that did not include the a, the target of the assignment. After the assignment `a := b` the predicate was $P' \wedge \mathtt{a} \equiv \mathtt{b}$. The only assignment that really required complex reasoning on path independence was `first.right := previous`.

The specification of the proof goal was straight forward. Talking about the effect of a list reversal did not require higher level specifications like models or closures. The ability to refer to a sequence of n attributes in the form of $\forall i \in 0 \ldots n-1 : \mathtt{first.right}^i$ has been very helpful.

5 Related Work

The concept of using paths to reason over pointer structures is not new. The terminology might be different though. In one of the very first papers on proving data-structures [3], Burstall talks about *unit strings*.

The basis to the approach of using equivalence relations for modelling pointers was developed by Jonkers [9]. It had been further studied by Deutsch [5]. Recent publications include the work by Schieder [17], Bozga, Iosif, and Laknech [2] and Hoare and Jifeng [7]. Schieder and Bozga et. al. both develop a weakest precondition calculus for Jonker's store model.

We can see four contributions of this paper in relation to the previous work. First, we are integrating a storeless semantic description into an object-oriented language by the implicit use of `Current` as a root for attributes. Second, we develop a semantic description for object creation that is significantly simpler than [17], [2] and [7]. Third, we introduce *path independence* and *path influence*, which we think are important concepts for storeless models. Finally, we approach the problem of feature invocation by giving a precise rule that uses prefix substitutions.

The term *trace* has been used in recent publications (see [7], [17]). As the term trace has a strong association with a dynamic behavior of a program ("a trace of states"), we have decided to choose the term *path* (as used by Cartwright et al. [4] and Morris [12]).

A very different approach to the aliasing problem is *separation logic* [16]. Instead of capturing aliasing conditions, separation logic tries to partition the heap. It is a very powerful description technique to rule out aliasing between storage locations. As a drawback, separation logic introduces a mathematical notation that is quite far away from the notations used to describe the program.

6 Future Work

Key to the successful application of a theory for practical purposes is the implementation of the theory in a theory proving environment. This is the long-term goal of the research efforts.

We are planing to implement the theory in the Isabelle/HOL[15] prover. Isabelle/HOL allows the implementation of complex rules and offers a rich set of specification techniques like data-structures, primitive recursion and inductive reasoning.

On top of Isabelle, we are planning to develop an interactive proving environment that creates and organizes the proof obligations based on a given application or software component written in Eiffel.

One of the problematic points seems to be that the theory relies on equivalence relations. These relations are normally hard problems for automated verification algorithms, as they tend to create infinite reductions and resist greedy simplification strategies.

7 Conclusion

Path properties remove heaps, objects and pointers from proving object-oriented programs. Instead we have focused on paths: lists of identifiers that are part of the textual representation of the program.

We have developed a small theory of paths and described the state as an equivalence relation on paths. We have captured important properties of this state description like attribute equivalence and path influence and independence. We have shown how these properties can be used to describe the axioms in an axiomatic semantics for pointer swing and object creation.

The proof of the list reversal algorithms has given insights on how reasoning on such a state description takes place. The ability to remove the target of the assignment from the predicate in the pre-state and thus making the non-interference proof trivial seems to be a good strategy for practical proofs. In general, path properties allow reasoning on pointer structures without an explicit model building process.

In the future, we are hoping to extend the approach to a full subset of an object-oriented language and cover aspects like inheritance or modularity within the theory.

8 Acknowledgements

I would like to thank my supervisor, Prof. Bertrand Meyer, for his guidance, insight and motivations for this work. I would also like to thank Prof. Peter Müller for being available for detailed discussions and questions on programming language semantics and aliasing. Finally I would like to thank Joseph Ruskiewicz, Vijay d'Silva, Susanne Cech and Ronny Zakhejm for listening to my explanations and giving me valuable feedback.

References

[1] Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. In *SPACE 2004*, 2004.

[2] Marius Bozga, Radu Iosif, and Yassine Laknech. Storeless semantics and alias logic. In *Proceedings Of The 2003 Acm Sigplan Workshop On Partial Evaluation And Semantics-Based Program Manipulation*, pages 55–65. ACM Press, 2003.

[3] R. M. Burstall. Some techniques for provng correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[4] Robert Cartwright, Robert Hood, and Philip Matthews. Paths: an abstract alternative to pointers. *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–27, 1981.

[5] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 Conference on Computer Languages*, pages 2–13, April 1992.

[6] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–583, October 1969.

[7] C.A.R. Hoare and He Jifeng. A trace model for pointers and objects. In *Programming Methodology*, Monographs in Computer Science, pages 223–245. Springer-Verlag New York, Inc., 2003.

[8] J. Hogg, D. Lea, A. Wills, and D. de Champeaux. Report on ECOOP'91 workshop W3: The Geneva convention on the treatment of object aliasing. In *OOPS Messenger*, volume 3 of *2*, pages 11–16, 1992.

[9] H.B.M. Jonkers. Abstract storage structures. In de Bakker / van Vliet, editor, *Algorithmic Languages*, pages 321–343. IFIP, North-Holland Publishing Company, 1981.

[10] Joseph Kiniry and Erik Poll. Opportunities and challenges for formal specification of java programs. `http://www.cs.ru.nl/~erikpoll/publications/prato.html`, January 2003. Position paper for trusted components workshop, Parto, 2003.

[11] Bertrand Meyer. *Eiffel: the language*. Prentice Hall, New York, NY, 1992.

[12] Joseph M. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, (Lecture Notes International Summer School, Markoberdorf), pages 25–34. Reidel, Dordrecht, Netherlands, 1982.

[13] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[14] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003. Available from`ftp://ftp.cs.iastate.edu/pub/techreports/TR01-03/TR.pdf`.

[15] Tobias Nipkow, Laurence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2004.

[16] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, July 2002.

[17] Birgit Schieder. Pointer theory and weakest precondition without addresses and heap. In *MPC2004*, LNCS 3125, pages 357–380. Springer-Verlag Berlin, 2004.

[18] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.