Supporting the Design Pattern "Object Structures as Plain Values"*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

September 25, 2009

Abstract

We sketch the principles of a type system for an object-oriented language such as Java which allows to statically ensure that an object structure is not modified by a method call, if the primary reference of that object structure is stored in a local variable of the method and this variable does not syntactically occur in the call. The object structure thus behaves like a "plain value", say a machine number, stored in a local variable in that no hidden side-effects can change it. We call the corresponding design pattern "object structures as plain values". The model is presented in an informal style; its validity still remains to be shown by a formal definition and soundness proof.

Contents

*Supported by the Austrian Academic Exchange Service (ÖAD) under the contract HU 14/2009.

1 Introduction

Our goal is to investigate which constraints are sufficient in order to consider an object structure as a "plain value". By this, we mean the following: if one declares in a method a local variable "x" of a primitive datatype and subsequently calls some method "m", e.g.

```
int x = ...;
...
r = o.m(y)
...
```

one can be sure that, if the name "r" of the result variable is different from "x", the value of "x" remains unchanged. In other words, the effect of the execution of "m" is restricted to its output variable r and any global variables to which m has access (such as "static" class variables); if the programming language supports "inout" parameters ("call by reference"), also variable "y" may be affected. In any case, "x" is "protected" from the execution of "m" by being a local variable that is not syntactically mentioned in the method call.

However, the situation changes if we consider objects represented by pointers (as is the case in most object-oriented languages such as Java or C#). If "x" denotes some object o (i.e. "x" contains a pointer to o), e.g. as in

```
C x = ...; // class C { ... }
...
r = o.m(y)
```

one can in general not be sure any more that o is unaffected by the call of "m" because of the following issues:

Referencing If some argument of "m" is "x" itself, "m" might (even in a language that supports only "input" parameters, i.e. "call by value") update the content of o (respectively the content of any object reachable via the fields of o, i.e. any object denoted by some expression $o.f_1.\ldots.f_n$ for corresponding object variables $f_1, \ldots, f_n$).

Aliasing Even if all arguments of "m" are different from "x", some might contain a pointer to o and thus update the content of o (or of any object reachable via the fields of o). Consequently, an expression "x.f" (for any variable "f" of object o) might have a different value after the call of "m" than before the call, even if "x" is not mentioned in the call.

Sharing Even if no argument of "m" does not contain a pointer to "o" itself, some might contain a pointer to some object o' reachable via the fields of o and thus update this object. Consequently, an expression "x.f.g" (for any variable "g" of the object denoted by "x.f") might have a different value after the call of "m" than before.

We are going to investigate these issues in more detail.

Referencing This issue is an immediate consequence of the fact that objects represented by pointers have "reference semantics" rather than "value semantics". Actually, the issue can be handled in a quite-straight forward way, without sacrificing the view of objects as "plain values", by treating the corresponding parameters of "m" as "inout" parameters" (rather than as input parameters), which only slightly complicates the corresponding reasoning: in a call

```
r = o.m(x,y,z)
```

each argument denoting an object must appear as an assignable variable; after the call of m, each of these variables has a new value (in addition to the result variable "r"). However, to prevent the subsequent "aliasing" issue, no object may appear in multiple argument positions of m, i.e. a call

```
r = o.m(...,x,...,x,...)
```

is prohibited.

Aliasing This issue arises from all statements where an object pointer is copied from one variable to another such that the same object may be denoted by different expressions. In particular, the issue arises in assignment statements

```
y = x
```

where subsequently both y and x refer to the same object o and in method calls

```
r = o.m(x)
```

provided that "x" represents a variable to which also "m" has access. This may be e.g. a class variable of any class D

```
class D
{
  static C c; // class C { ... }
  ...
}
```

because both $D.c$ and the method parameter denote o. Aliasing complicates reasoning about programs a lot because the effect of updating an aliased object

```
x.f = ...
```

cannot be contained to the variable "x" but might also affect a syntactically unrelated local variable "y" or class variable "D.c". To deal with this problem one either has to add numerous "non-aliasing" constraints to the specifications of methods and classes or resort to special approaches to program reasoning such as separation logic [5].

Sharing This issue arises if not only plain variables can receive object pointers but also objects themselves contain object pointers, e.g. as in

```
class D
{
  C c; // class C { ... }
  ...
}
```

Assume that "y" is a variable of type "D". After an assignment

```
y.c = x
```

both $y.c$ and x refer to the same object and in method calls

```
r = o.m(y.c)
```

both $y.c$ and the parameter of m refer to the same object. Consequently an update

```
x.f = ...
```

may have an effect on a syntactically unrelated object field "y.c.f" in an object o' reachable from "y". Different object structures may thus share substructures such that an update on one object structure has also an effect on another; furthermore from a class representing a data structure a pointer to a substructure may "leak" to some user of the class such that the user may directly update the substructure bypassing the interface provided by the class.

Object Structures as Plain Values Our goal is now to set up a framework which allows one to ensure by static modular reasoning on the program text that in a code pattern of form

```
C x = ... // class C { ... }
...
r = o.m(y)
```

the call of method "m" does not affect "x" provided that "x" does syntactically not appear as argument variable "y" or result variable "r" (respectively, if y and r denote expressions, *within* "r" and "y"). We call this the design pattern "object structures as plain values" (which is however not among the classical design patterns [1]).

2 Notions

To make our elaboration reasonably precise, we introduce a couple of notions.

Definition (Source Code Expression)**.** A *source code expression* (short *expression*) is a syntactic phrase in a program that may denote a value (possibly an object).

Definition (Access Path)**.** An *access path* is an expression $e\ a_1 \ldots a_n$ with $n \geq 0$ such that e is an expression and each *selector* a_i is either of the form $.v_i$ (object field access with field variable v_i) or of the form $[e_i]$ (array element access with index expression e_i).

Remark (Access Paths and Pattern Matching). In the following, when we refer to an "access path $e\ a_1 \ldots a_n$", we always implicitly assume that e that e does not end in a selector. Consequently, the individual elements, $e, a_1, \ldots, a_n$ are uniquely defined.

Definition (Reachability)**.** An object o' is *reachable* from object o, if there is some expression e that denotes o and some access path $e\ a_1 \ldots a_n$ that denotes o'.

Remark (Reflexivity of Reachability). Every object denoted by some program expression is reachable from itself.

Definition (Encapsulation)**.** An object o' is *encapsulated* by object o, if for every access path $e\ a_1 \ldots a_n$ that denotes o' some access path $e\ a_1 \ldots a_i$ with $i \leq n$ denotes o.

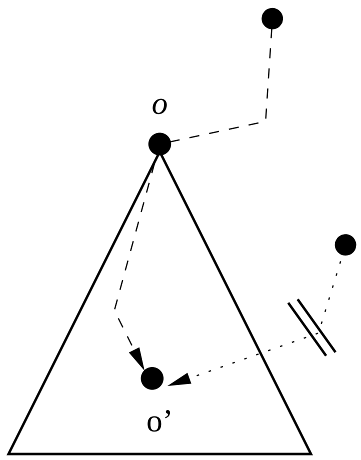

Figure 1: Object o' is encapsulated by the closed object o

Remark (Transitivity of Encapsulation)*.* If object o' is encapsulated by object o and object o'' is encapsulated by o', then o'' is also encapsulated by o.

Definition (Closedness)**.** An object o is *closed*, if every object o' reachable from o is encapsulated by o.

The idea of above definitions is illustrated in Figure 1 where o denotes a closed object and the triangle denotes the set of objects reachable from (and thus encapsulated by) o. Every access path to an object o' in this set must pass through o. As a result, no object outside the set may directly refer to an object inside and the whole object o behaves like an "atomic" value.

Definition (Plain Value)**.** An object o is a *plain value* if

1. o is closed, and
2. there exists at most one variable that contains a reference to o.

While the first condition hides the fact that a plain value o may contain references to objects, the second condition hides the fact that o itself is an object represented by a reference.

Definition (Local Objects)**.** An object o is a *local* if only local variables (method parameters or variables declared inside a method) contain references to o.

Proposition (Uniqueness of Local Plain Values)**.** *If a plain value o is local, then there exists at most one access path denoting o, namely a reference x to some local variable x.*

Proof. The proposition is self-evident. □

Definition (Modification)**.** An object o is *modified* by a method call, if for some expression e denoting o some access path $e\ a$ has after the call a value that is different from the value before the call.

Definition (Effect)**.** An object o is *affected* by a method call, if some object that is reachable by o is modified by the call.

Remark (Modification and Effect). If some access path $e\ a_1 \dots a_n$ denotes an object that is modified by a method call, then the objects denoted by the access paths e, $e\ a_1$, …, $e\ a_1 \dots a_{n-1}$ are affected by the call.

The behavior of method calls is captured by the following axiom.

Axiom (Method Execution)**.** *For a method call*

```
e_r = e_o.m(e_1, ..., e_n)
```

every object modified by the method call is reachable by an object that is denoted by a "static" class variable or by some expression that appears among (or as a subexpression within some of) the expressions $e_r, e_0, e_1, \dots, e_n$.

Remark (Static Methods). This definition easily generalizes to the execution of a static method of class C

```
e_r = C.m(e_1, ..., e_n)
```

by taking e_0 as the "empty" expression. The following proposition thus also applies to static methods.

The following proposition captures the core idea of this paper.

Proposition (Local Plain Values and Method Calls)**.** *Take a method call*

```
e_r = e_o.m(e_1, ..., e_n)
```

and a local variable x *such that*

- x *denotes a plain value, and*
- x *does not occur (possibly as a subexpression) among the expressions* $e_r, e_0, e_1, \dots, e_n$.

Then the object denoted by x *is not affected by the method call.*

Proof. Take the plain value o denoted by x; since x is a local variable, o is local. We assume that o is affected by the method call and show a contradiction. From the definition of "effect", we know that some object o' that is reachable from o is modified by the call. From the definition of "reachability", we know that o' is denoted by some access path $x\ a_1 \dots a_n$. From the axiom "method execution", we know that o' is also reachable from some object o'' that is denoted by some expression $e'' = e'''\ b_1 \dots b_m$ which is either a reference to a static class variable or appears (possibly as a subexpression) among the $e_r, e_0, e_1 \dots, e_n$. Thus there also exists an access path $e'''\ b_1 \dots b_{m+p}$ that denotes o'.

Since o is a plain value and o' is reachable from o, by the definition of "plain value", o' is encapsulated by o. Thus, since o' is denoted by $e'''\ b_1 \dots b_{m+p}$, we know that some access path $e'''\ b_1 \dots b_i$ with $i \leq m + p$ denotes o. Since o is a local plain value denoted by x, this implies that x equals $e'''\ b_1 \dots b_i$ and thus $i = 0$, $e''' = x$, and $e'' = x\ b_1 \dots b_m$. Since x does neither denote a static class variable nor does it occur (possibly as a subexpression) among the $e_r, e_0, e_1 \dots, e_n$, this contradicts our knowledge about e''. □

Our problem is thus reduced to ensuring that a variable x denotes a plain value. The remainder of the paper deals with this problem.

3 Typechecking Plain Values

In order to ensure by static modular reasoning ("type checking") that a variable denotes a plain value, we introduce a class annotation `value`

```
/* value */ class T
{
  ...
}
```

which indicates that every instance of T (respectively of a subclass of T) is a plain value. Every instance of class T thus receives implicitly type `value` T (which is different from a normal class type T).

Likewise, we introduce a class annotation `local`

```
/* local */ class T
{
  ...
}
```

which indicates that every instance of T (respectively of a subclass of T) is encapsulated by some plain value. Every instance of class T thus receives implicitly type `local` T (which is different from a normal class type T).

Since arrays also have object behavior, but are instances of a builtin "array" type, we also `value` and `local` to appear in array types as

```
/* value */ T[]
/* local */ T[]
```

indicating arrays that are plain values respectively arrays that are encapsulated by a plain value. To create such arrays, we generalize the array creation operator `new T[...]` to the two variants

```
new /* value */ T[...]
new /* local */ T[...]
```

Furthermore we allow `local` to appear as a type modifier such that for a `value class T` the type expression

```
/* local */ T
```

denotes type `local T`.

There are various constraints for the occurrences of `value`/`local` types:

1. A (static) class variable cannot have a `local` type.

2. Any other kind of variable can have a `local` type only, if the variable is declared within a `value` or a `local` class.

3. In a `value` or `local` class, *all* (non-static) object variables that denote a reference to an object/array must have `value` or `local` types.

We are now going to present the type compatibility rules with respect to the three types T, `value` T, `local` T. We call T the *base type* of these three types. The usual subtyping rules of an object-oriented language are preserved i.e. whenever a base type T is expected, also a base type T' may appear, provided that T is an ancestor of T' in the inheritance hierarchy.

1. A variable x of type T may only receive a value of type T (not of type `value` T and not of type `local` T).

2. A variable x of type `value` T may only receive a value of type `value` T (not T or `local` T). Furthermore, it must be possible to statically ensure that, before the object referenced by x is used the next time (i.e. a field of the object is dereferenced or a method of the object is called), that there exists only one reference to o (see below).

3. A variable x of type `local` T may only receive a value of type `local` T or `value` T (not T). In the later case, it must be possible to statically ensure that, before the object referenced by x is used the next time (i.e. a field of the object is dereferenced or a method of the object is called), that there exists only one reference to o (see below).

   Rules 1–3 above also apply to the parameters of methods (here x denotes the function parameter) and to the return value of a method (here x represents all variables that may receive the return value). We will elaborate later further what this means in detail for methods whose parameters respectively return values are `local` or `value` types.

4. For an object o that is different from `this` and has a `value` type, the use of an access path $\ldots o \ldots s$ with at least one selector s at the end of the path is prohibited, if the type of the value denoted by the whole path is a `local` type or a `value` type, i.e. if one of the following conditions holds:

   (a) s is a reference $.x$ to a (non-static) object variable x of a `local` or `value` type,

   (b) s denotes the access $[\ldots]$ to an array element of a `local` or `value` type,

   (c) s is a method call $.m(\ldots)$ of a method m whose return type is a `local` or `value` type.

We give an example that type-checks correctly according to above rules. In the presented code, an object of type `IntArrayList` is a plain value that encapsulates multiple doubly linked `IntArrayNode` objects each of which encapsulates an `IntArray` plain value which in turn encapsulates an `int[]` plain value.

```
/* value */ class IntArray /* encapsulates the array a */
{
  /* value */ int[] a = new /* value */ int[100];
  int p = 0;
  void add(int i)
  {
    if (p == 100) return; a[p] = i; p++;
  }
}

/* local */ class IntArrayNode /* encapsulates the array object */
{                                /* the node referenced by next is */
  IntArray array;                /* encapsulated by the owner of this node */
  IntArrayNode next;
  IntArrayNode prev;             /* doubly linked list */
```

```
  IntArrayNode(IntArray a, IntArrayNode n)
  {
    array = a; next = n; n.prev = this;
  }
}

/* value */ class IntArrayList
{
  IntArrayNode head = null;     /* encapsulates the head object */
  void insert(IntArray a)       /* a is encapsulated by this list */
  {
    head = new IntArrayNode(a, head);
  }
  IntArray remove(int n)        /* result is removed from this list */
  {
    IntArrayNode node = head;
    for (int i=0; i<n; i++) node = node.next;
    IntArray array = node.array;
    node.array = null;
    return array;
  }
}

class Main
{
  static void main()
  {
    IntArrayList list = new IntArrayList(); /* a plain value */
    IntArray array = new IntArray();        /* a plain value a */
    array.insert(5);
    list.insert(array);                     /* list is a plain value */
    array = new IntArray();                 /* a plain value b */
    array.insert(5);
    array = list.remove(0);                 /* a is retrieved again */
  }
}
```

We also give some examples that do *not* type-check correctly:

```
/* value */ class IntArray /* encapsulates the array a */
{
  ...
  /* value */ int[] leak()
  {
    /* INVALID (Rule 2): object still refers to a */
    return a;
  }

  /* value */ int[] extract()
  {
    /* value */ int[] b = a;
    a = null;

    /* CORRECT: object does not refer to b any more */
    return b;
```

```
  }
}

class Main
{
  static void main()
  {
    ...
    IntArray array = new IntArray();
    array.insert(5);
    list.insert(array);

    /* INVALID (Rule 2): array used after passed to list.insert() */
    array.insert(5);
    ...

    /* INVALID (Rule 4): array.a has a value type */
    array.a[0] = 1;
  }
}
```

Above examples do not use the access specifier `private` to protect the fields of `value` or `local` types. While one might/should actually do so, the effect of declaring a (non-static) object variable with a `value` or `local` type is different from annotating it with `private`:

- On the one hand, declaring a field x in class C as `private` does not prevent an object o of type C to access (in the body of an object method m of class C) the field $o'.x$ of a *different* object o' (which is prohibited by Rule 4, if the type of x is a `value` type).
- On the other hand, given an object o of type C, declaring a field x in C with a `local` type D still allows to acess the field from another class as $o.x$ (which is prohibited, if the declaration of x is tagged as `private`).

Above description of the type system leaves two issues open:

1. A plain value passed as a method argument is always assumed to be "grabbed" by the method (i.e. it is assumed that the receiver object of the method retains a reference to the object). Consequently, the value cannot be used further by the caller of the method who thus has to duplicate the argument even if the method does actually not grab it.
2. The phrase "it must be possible to statically ensure that, before the object referenced by x is used the next time (i.e. a field of the object is dereferenced or a method of the object is called), that there exists only one reference to o" needs to be explicated.

We are now going to address these.

Borrowed Types To overcome the first issue, we introduce a type annotation `borrowed` such that the type expression

```
/* borrowed */ T
```

with `value` type T indicates that, for any variable x declared with this annotation, no reference to the object o denoted by x (or to any object reachable from x) may be stored in a static class variable or in an non-static object variable. The type of x becomes `borrowed` T' (where T' is the base type of T); this type behaves exactly like `value` T' (and is subject to the corresponding constraints of that type) except for the following:

- Only a local variable in a method or a method parameter or a method return value may have a `borrowed` type.
- A variable of type `borrowed` T may receive a value of type `borrowed` T or of type `value` T.
- A value of type `borrowed` T may be only stored in a variable of type `borrowed` T (neither in a variable of type `value` T nor in a variable of type `local T`).

As a consequence, the call of a method with an argument v of type `value` T for a parameter declared as `borrowed` T does not invalidate any subsequent use of v *after* the method call; however, it still invalidates any other use of v *during* the method call, i.e., it must not appear as (part of) another method argument.

Thus for instance the following piece of code is legal:

```
/* value */ class Counter
{
  int x = 1;
  void add(/* borrowed */ Counter c)
  {
    x = x+c.x;
  }
}

class Main
{
  static void main()
  {
    Counter c1 = new Counter();
    Counter c2 = new Counter();
    c1.add(c2);
    c1.add(c2); /* legal, c2 was just borrowed by c1 */
  }
}
```

Ensuring Reference Uniqueness In the following, discuss how to ensure the second issue i.e. making sure that objects of `value` type are uniquely referenced. For this purpose, we give a simple algorithm that checks programs to satisfy this constraint; however, due to its simplicity it also rejects correct programs. More sophisticated analysis techniques [4] are needed to develop a checker that delivers more precise results.

The following is the syntax of the commands of a simple object-oriented language that is input to the checker. The checker takes a method body (a command); if the checker returns true, the execution of the method body does not construct a (permanent) duplicate reference to a value object; in particular, to every method invoked by the current method at most one reference to every `value` object is visible.

We assume in the analysis that every reference x to a (non-static) object method has been previously expanded to `this.`*x.*

Furthermore, some commands have to be previously annotated by the type checker (assignment statements with the types of the assigned values and method calls with the types of the method parameters.

$$
\begin{aligned}
& B \in \text{Body} \\
& C \in \text{Command} \\
& E \in \text{Exp} \\
& Es \in \text{Exps} \\
& R \in \text{Ref} \\
& T \in \text{Type} \\
& Ts \in \text{Types} \\
& I \in \text{Ident}
\end{aligned}
$$

$$
\begin{aligned}
& B := C \\
& C := \\
& \quad R \texttt{ =}^{T}\ E \mid \texttt{return} \mid \texttt{return}\ E \\
& \quad \mid E_o\texttt{.}I^{Ts}(Es) \mid R \texttt{ =}^{T}\ E_o\texttt{.}I^{Ts}(Es) \\
& \quad \mid C_1\texttt{;}C_2 \mid \texttt{if (}E\texttt{)}\ C \mid \texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while (}E\texttt{)}\ C \\
& \mathrm{E} := \texttt{null} \mid \texttt{new}\ T\ (Es) \mid R \\
& \mathrm{R} := \texttt{this} \mid I \mid R\texttt{.}I \\
& \mathrm{T} := \ldots \\
& Es := Es\ E \mid \_ \\
& Ts := Ts\ T \mid \_
\end{aligned}
$$

The subsequent algorithm is based on the following domains:

$$
\begin{aligned}
& \mathit{RefSet} := \mathbb{P}(\mathit{Ref}) \\
& \mathit{RefSetPair} := \mathit{RefSet} \times \mathit{RefSet} \\
& \mathit{Error} := \{()\} \\
& \mathit{isValue}(T) :\Leftrightarrow \exists T' : T = \texttt{value}\ T' \\
& \mathit{isBorrowed}(T) :\Leftrightarrow \exists T' : T = \texttt{borrowed}\ T'
\end{aligned}
$$

The algorithm consists of several relations/functions which process syntactic phrases (the name $[\![\ldots]\!]$ is overloaded to denote all functions, which function is uniquely determined by the types of the arguments of the function call):

$$
\begin{aligned}
& [\![\ ]\!] \subseteq \text{Body} \\
& [\![\ ]\!] : \text{Command} \times \mathit{RefSet} \to \mathit{RefSet} + \mathit{Error} \\
& [\![\ ]\!] : \text{Exp} \to \mathit{RefSet} \\
& [\![\ ]\!] : \text{Ref} \to \mathit{RefSet} \\
& [\![\ ]\!] : \text{Exps} \to \mathit{RefSet} \\
& [\![\ ]\!] : \text{Exps} \times \text{Types} \times \mathit{RefSetPair} \to \mathit{RefSetPair} + \mathit{Error}
\end{aligned}
$$

The top-level relation application $[\![C]\!]$ takes a method body C and calls the function $[\![C]\!]$ on commands which takes a set of object references that have to be assigned a new object such that the application of the command is valid (initially empty) and returns such a set. If the result set is empty, the method body passes the check.

$$[\![\ ]\!] \subseteq \text{Body}$$
$$[\![C]\!] \Leftrightarrow [\![C]\!]\emptyset = \emptyset$$

Before describing the function $[\![C]\!]$ on commands, we turn our attention to the other auxiliary functions.

A function application $[\![E]\!]$ returns the set of all references contained in the expression E, likewise $[\![R]\!]$ returns all (sub)references in reference R, and $[\![Es]\!]$ returns all references in the expression sequence Es:

$$[\![\ ]\!] : \text{Exp} \to \mathit{RefSet}$$
$$[\![\texttt{null}]\!] = \emptyset$$
$$[\![\texttt{new}\ T\ (Fs)]\!] = [\![Fs]\!]$$
$$[\![R]\!] = [\![R]\!]$$
$$[\![\ ]\!] : \text{Ref} \to \mathit{RefSet}$$
$$[\![\texttt{this}]\!] = \{\texttt{this}\}$$
$$[\![I]\!] = \{I\}$$
$$[\![R.I]\!] = [\![R]\!] \cup \{R.I\}$$
$$[\![\ ]\!] : \text{Exps} \to \mathit{RefSet}$$
$$[\![Es\ E]\!] = [\![Es]\!] \cup [\![E]\!]$$
$$[\![\_]\!] = \emptyset$$

The function application $[\![C]\!]rs$, takes the set of references that have to receive a new value before the object denoted by these references may be used; if C violates this constraint, the function returns an *Error* value, otherwise it results in another set of references that must receive a new value before the denoted objects may be used.

The definition of this function is based on the constraint, that there may at every time at most *two* references to a *value* object one of which is contained in rs. In an assignment $R \texttt{=}^T E$, no subreference of R and no reference in E must be in rs; by the assignment R receives a new value E and is thus removed on the set. If E denotes a reference r, every occurrence of r (also as a subreference) in rs is replaced by R and r itself is added to the set:

$$[\![\ ]\!] : \text{Command} \times \mathit{RefSet} \to \mathit{RefSet} + \mathit{Error}$$
$$
\begin{array}{l}
[\![R \texttt{=}^T E]\!]rs = \\
\quad \textsc{if}\ [\![R]\!] \cap rs \backslash \{R\} \neq \emptyset \vee [\![E]\!] \cap rs \neq \emptyset\ \textsc{then} \\
\quad\quad \textsc{is}\mathit{Error}() \\
\quad \textsc{else case}\ E\ \textsc{of} \\
\quad\quad \text{Ref}(r): \\
\quad\quad\quad \textsc{if}\ \neg \mathit{isValue}(T) \\
\quad\quad\quad\quad \textsc{then}\ rs[R/r] \\
\quad\quad\quad\quad \textsc{else}\ (rs \backslash \{R\})[R/r] \cup\ \{r\} \\
\quad\quad \textsc{otherwise}: rs \backslash \{R\}
\end{array}
$$

Above algorithm is restricted in that is assumes that in an assignment $R := r$ always the reference r has to be invalidated; a more general version might keep track of all aliases of r and make sure that all but one are eliminated before the denoted object is used. Likewise, we replace in rs every occurrence of r (also as a subreference in other references) by R rather than keeping track of reference equalities in a more general way. Nevertheless the approach suffice to detect e.g. in a sequence of assignments

```
node.next.val = node.val; // rs = { node.val }
temp = node;              // rs = { temp.val }
node = node.next;         // rs = { temp.val }
temp.val = null;          // rs = { }
```

with object field `val` of some `value` type that the temporary duplicate reference to the value is deleted.

A `return` statement must make sure that all duplicate references are deleted (more general, we could ignore duplicate references stored in local variables of the method):

$[\![\,\texttt{return}\,]\!] rs =$
$\quad$ IF $rs = \emptyset$ THEN $\emptyset$ ELSE IS$Error()$
$[\![\,\texttt{return}\ E\,]\!] rs =$
$\quad$ IF $rs = \emptyset$ THEN $\emptyset$ ELSE IS$Error()$

Before a method is called, no duplicate variable references may exist (again, more general, we might ignore duplicate references stored in local variables of the method). Furthermore, we need to collect the set ra of all references to value objects used as object parameters for subsequent invalidation; from this set we may remove the elements rb of all references to value objects passed as `borrowed` parameters, and R itself (if it denotes a value object):

$[\![\,E\,.\,I^{Ts}(Es)\,]\!] rs =$
$\quad$ IF $rs \neq \emptyset$ THEN
$\quad\quad$ IS$Error()$
$\quad$ ELSE CASE $[\![\,Es\,]\!][\![\,Ts\,]\!]\ \emptyset$ OF
$\quad\quad$ IN$Error()$: IS$Error()$
$\quad\quad$ IN$RefSetPair(ea, eb)$: $ea \backslash eb$
$[\![\,R\ \texttt{=}^{T}\ E\,.\,I^{Ts}(Es)\,]\!] rs =$
$\quad$ IF $rs \neq \emptyset$ THEN
$\quad\quad$ IS$Error()$
$\quad$ ELSE CASE$[\![\,Es\,]\!][\![\,Ts\,]\!]\ \emptyset$ OF
$\quad\quad$ IN$Error()$: IS$Error()$
$\quad\quad$ IN$RefSetPair(ea, eb)$:
$\quad\quad\quad$ LET $rs = ea \backslash eb$ IN
$\quad\quad\quad$ IF $[\![\,R\,]\!] \cap rs \backslash \{R\} \neq \emptyset$ THEN
$\quad\quad\quad\quad$ IS$Error()$
$\quad\quad\quad$ ELSE IF $\neg isValue(T)$ THEN
$\quad\quad\quad\quad$ rs
$\quad\quad\quad$ ELSE
$\quad\quad\quad\quad$ $rs \backslash \{R\}$

A function application $[\![\,Es\,]\!][\![\,Ts\,]\!]\langle ea, eb\rangle$ takes the set of all references of `value` objects used as arguments in the current method call and the corresponding set of all `borrowed` values and updates these with respect to the remaining method arguments Es with corresponding types Ts of method parameters:

$[\![\;]\!] : \text{Exps} \times \text{Types} \times \mathit{RefSetPair} \to \mathit{RefSetPair} + \mathit{Error}$

$[\![\,\_\,]\!][\![\,\_\,]\!]\mathit{esp} = \mathit{esp}$

$[\![\,\mathit{Es}\ E\,]\!][\![\,\mathit{Ts}\ T\,]\!]\mathit{esp} =$
$\quad \text{CASE } [\![\,\mathit{Es}\,]\!][\![\,\mathit{Ts}\,]\!]\mathit{esp} \text{ OF}$
$\quad\quad \text{IN } \mathit{Error}() : \text{IS}\mathit{Error}()$
$\quad\quad \text{IN } \mathit{RefSetPair}(\mathit{ea}, \mathit{eb}) :$
$\quad\quad\quad \text{ELSE IF } [\![\,E\,]\!] \cap \mathit{ea} \neq \emptyset \text{ THEN}$
$\quad\quad\quad\quad \text{IS}\mathit{Error}()$
$\quad\quad\quad \text{ELSE IF } \mathit{isBorrowed}(T) \text{ THEN}$
$\quad\quad\quad\quad \langle \mathit{ea} \cup \{E\}, \mathit{eb} \cup \{E\} \rangle$
$\quad\quad\quad \text{ELSE IF } \mathit{isValue}(T) \text{ THEN}$
$\quad\quad\quad\quad \langle \mathit{ea} \cup \{E\}, \mathit{eb} \rangle$
$\quad\quad\quad \text{ELSE}$
$\quad\quad\quad\quad \langle \mathit{ea}, \mathit{eb} \rangle$

The following composed commands are checked in the expected way:

$[\![\,C_1 ; C_2\,]\!]\mathit{rs} =$
$\quad \text{CASE } [\![\,C_1\,]\!]\mathit{rs} \text{ OF}$
$\quad\quad \text{IN}\mathit{Error}() : \text{IS}\mathit{Error}()$
$\quad\quad \text{IN}\mathit{RefSet}(\mathit{rs}') : [\![\,C_2\,]\!]\mathit{rs}'$

$[\![\,\texttt{if (}E\texttt{)}\ C\,]\!]\mathit{rs} =$
$\quad \text{IF } [\![\,E\,]\!] \cap \mathit{rs} \neq \emptyset \text{ THEN IS}\mathit{Error}() \text{ ELSE } [\![\,C\,]\!]\mathit{rs}$

$[\![\,\texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2\,]\!]\ \mathit{rs} =$
$\quad \text{IF } [\![\,E\,]\!] \cap \mathit{rs} \neq \emptyset \text{ THEN}$
$\quad\quad \text{IS}\mathit{Error}()$
$\quad \text{ELSE CASE } [\![\,C_1\,]\!]\mathit{rs} \text{ OF}$
$\quad\quad \text{IN}\mathit{Error}() : \text{IS}\mathit{Error}()$
$\quad\quad \text{IN}\mathit{RefSet}(\mathit{rs}_1) :$
$\quad\quad\quad \text{CASE } [\![\,C_2\,]\!]\mathit{rs} \text{ OF}$
$\quad\quad\quad\quad \text{IN}\mathit{Error}() : \text{IS}\mathit{Error}()$
$\quad\quad\quad\quad \text{IN}\mathit{RefSet}(\mathit{rs}_2) : \mathit{rs}_1 \cup \mathit{rs}_2$

The treatment of loops is restricted in that, before the loop body C is entered, no `value` object must be aliased, and that, after the execution of C, also any temporary aliasing must be resolved:

$[\![\,\texttt{while (}E\texttt{)}\ C\,]\!]\mathit{rs} =$
$\quad \text{IF } \mathit{rs} \neq \emptyset\ \vee [\![\,E\,]\!] \cap \mathit{rs} \neq \emptyset \text{ THEN}$
$\quad\quad \text{IS}\mathit{Error}()$
$\quad \text{ELSE CASE}[\![\,C\,]\!]\emptyset \text{ OF}$
$\quad\quad\quad \text{IN}\mathit{Error}() : \text{IS}\mathit{Error}()$
$\quad\quad\quad \text{IN}\mathit{RefSet}(\mathit{rs}) : \text{IF } \mathit{rs} \neq \emptyset \text{ THEN IS}\mathit{Error} \text{ ELSE } \emptyset$

4 Related Work and Conclusions

The work described in this paper has been essentially inspired by the "ownership" model developed by Peter Müller and developed in numerous papers, see e.g. [2]. His model is more general by supporting a methodology of multiple nestings of object structures using `rep` pointers from one level to the next and `peer` pointers within a level. The model was later refined to allow transfer of objects between different named contexts introduced by `context` declarations [3].

The model described in this paper is more restricted in that only the issue of "object structures" as plain values is considered. However, by having uniquely referenced object structures, we can easily transfer values from one context to another (by passing and subsequently invalidating a pointer). So for this particular purpose, our model seems appealing. However, we should note that in this paper the model has been only informally sketched; so the results should be taken with great care. The validity of the model still remains to be investigated by a formal definition of the type system and corresponding formalized soundness proofs.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education, 1995.

[2] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[3] Peter Müller and Arsenii Rudich. Ownership Transfer in Universe Types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 461–478. ACM, 2007.

[4] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Principles of Program Analysis, 2005.

[5] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 22–25, 2002. IEEE.