Reasoning about Assignments in Recursive Data Structures

Alejandro Tamalet and Ken Madlener

Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, The Netherlands
{a.tamalet,k.madlener}@cs.ru.nl

Abstract. This paper presents a framework to reason about the effects of assignments in recursive data structures. We define an operational semantics for a core language based on Meyer's ideas for a semantics for the object-oriented language Eiffel. A series of field accesses, e.g. $\mathtt{f}_1 \bullet \mathtt{f}_2 \bullet \ldots \bullet \mathtt{f}_n$, can be seen as a path on the heap. We provide rules that describe how these *multidot* expressions are affected by an assignment. Using multidot expressions to construct an abstraction of a list, we show the correctness of a list reversal algorithm. This approach does not require induction and the reasoning about the assignments is encapsulated in the mentioned rules. We also discuss how to use this approach when working with other data structures and how it compares to the inductive approach. The framework, rules and examples have been formalised and proven correct using the PVS proof assistant.

1 Introduction

In order to verify pointer programs that manipulate recursive data structures, one generally identifies the pointer structure embedded in the heap with an abstract model. A concrete instance is a mapping of a set of objects on the heap connected by a field such as `next` to an abstract list of objects. The mapping is called the *abstraction* and the abstract list is called the *abstract model*. An operation performed by the program on a pointer structure on the heap has a corresponding operation on the abstract model. For example, the operations performed by a list reversal algorithm have the combined effect that the abstract list is reversed at the end of the execution. The standard way to define data abstractions is by recursion on the structure of (the data type of) the abstract model.

Verification of pointer programs is a non-trivial task due to the possibility of aliasing. Modifying data through one name implicitly modifies the values associated to all aliased names. If two portions of the heap are disjoint, an assignment in one part of the heap does not affect the other; this is called *local reasoning*. Local reasoning is essential for scalability and several approaches to obtain it have been studied, see e.g. Separation Logic [14] and Region Logic [15].

When it is not known how the heap is partitioned or when working within a region that may contain aliases, we have to reason about how a change to (a

portion of) the heap affects the corresponding abstract model. This complements local reasoning. In this paper we focus on the effects of assignments to abstract models. We present our work in the setting of a core language, inspired by Meyer's ideas for a semantics for the object-oriented language Eiffel [12].

Our framework allows us to express multidot field access expressions, multidot expressions for short, of the form $\mathtt{f_1} \bullet \mathtt{f_2} \bullet \ldots \bullet \mathtt{f_n}$. A multidot expression consisting of a series of `next`-fields describes a path from the head of a list to one of its elements. If we instantiate it with a series of `left` and `right`-fields we can describe the path from the root of a binary tree to any node or leaf. In general, a multidot expression describes a path on the heap where the elements are connected by field accesses.

The main contribution of this paper is to provide a set of rules that precisely describe the value of a multidot expression after an assignment, and to show how these rules can be applied for verification of programs that manipulate recursive data structures. The given rules are categorised into separation rules, where the assignment has no effect on the multidot expression, and interference rules, where the assignment does have effect on the multidot expression. We have applied these rules to show the correctness of an in-place list reversal algorithm by mapping each element of the list to a multidot expression. We also discuss how to apply the same principles to other recursive data structures and we make a comparison with the standard inductive approach. Our work has been completely carried out in the theorem prover PVS [13].

This paper is organised as follows. Section 2 gives a short introduction to PVS and introduces the notation. Section 3 defines the language we shall work with. In Section 4 we present the rules that describe the effects an assignment can have on a multidot expression and in Section 5 we apply these rules to prove the correctness of a list reversal algorithm and we discuss the applicability to other data structures. We compare the approach described in this paper with the standard inductive approach and we give pointers for future work in Section 6. Related work is discussed in Section 7 and conclusions are drawn in Section 8.

2 Preliminaries

PVS is based on higher-order logic with dependent types and predicate subtyping. Subtyping based on predicates makes type checking undecidable, so when PVS cannot infer the desired type itself, it will generate a proof obligation. Its intuitive syntax is reminiscent of functional languages like Haskell. For reasons of presentation, we slightly simplify the actual PVS syntax. We will briefly introduce the notation used in this paper.

Formulas are terms of type `bool`. We shall use the standard notation for connectives $(\wedge, \vee, \Rightarrow, \neg)$, and for quantifiers $(\forall, \exists)$. There is a conditional term IF φ THEN M ELSE N, for terms M, N of the same type.

Given the types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$, function types are written as $[\sigma \to \tau]$ and record types as $[\mathtt{lab}_1 : \sigma_1, \ldots, \mathtt{lab}_n : \sigma_n]$. Given the record types $\rho_1, \ldots, \rho_m$, labelled coproduct types are written as $\{\mathtt{lab}_1(\rho_1), \ldots, \mathtt{lab}_m(\rho_m)\}$. Terms of co-

product type can be constructed with $\mathtt{lab}_i(M)$, where $M : \rho_i$, and recognised with $\mathtt{lab}_i$?. Standard set comprehension notation can be used to define predicate subtypes. New types can be introduced via definitions, like

$$\mathtt{lift}[\sigma] \;:\; \text{TYPE} = \{\mathtt{bottom},\; \mathtt{up}(\mathtt{down} : \sigma)\}$$

(bottom is the unit type and we omit its argument). The lift type constructor adds a bottom element to an arbitrary type σ given as a parameter, written in short as $\sigma_\perp \mathrel{\widehat{=}} \mathtt{lift}[\sigma]$.

Lists are defined as

$$\mathtt{list}[\sigma] \;:\; \text{TYPE} = \{\mathtt{null},\; \mathtt{cons}(\mathtt{car} : \sigma,\; \mathtt{cdr} : \mathtt{list})\}$$

There is an infix function $+\!\!+$ that appends two lists. It is overloaded so that when one of its arguments is of type σ, then this argument is converted to a list. The ith element of a list l can be accessed using $\mathtt{nth}(\mathtt{l}, \mathtt{i})$.

3 The Model

This section describes an operational semantics of a core object-oriented language. The focus is on the features needed to understand the properties discussed in the next section, i.e., we do not model some typical object-oriented features like inheritance. The interested reader can find the full PVS formalisation at http://cs.ru.nl/~tamalet.

3.1 The heap

In our model we consider all values to be an object or void. The set Object is defined as an uninterpreted type that represents non-void objects. Instances of $\mathtt{Object}_\mathtt{v}$ have the possibility of being void:

$$\mathtt{Object}_\mathtt{v} \;:\; \text{TYPE} = \{\mathtt{obj}(\mathtt{obj} : \mathtt{Object}),\; \mathtt{void}\}$$

A basic approach to model the heap, due to Burstall [5] and more recently emphasised by Bornat [3], is to model it as a collection of functions of type $\mathtt{Object} \to \mathtt{Object}_\mathtt{v}$, one for each class field (i.e. the component). This modelling syntactically encodes the fact that two fields can not be aliased. This has the important consequence that whenever one field is updated, we have for free that all the other fields are left untouched. This is sometimes called the component-as-array model [7, 9].

Our heap is a grouping of field functions, indexed by their field names:

$$\mathtt{Heap} \;:\; \text{TYPE} = [\mathtt{Name} \to [\mathtt{Object} \to \mathtt{Object}_\mathtt{v}]]$$

where Name is a set representing the field names. Given a heap h and a field name f, $\mathtt{h}(\mathtt{f})$ is the corresponding field function. This indexing allows us to reason about field names, which is not possible when using a loose set of field functions as in the component-as-array model. There, the names of the fields are fixed by the names of the functions that model them. We use this to express meta-properties

about multidot field expressions in Section 4. The separation by syntax provided by the component-as-array model is lost in this model, because a field update is now an update on the heap function. With the meta-level properties presented in the rest of this paper, we obtain a reincarnation of separation by syntax.

The above definition of the heap highlights the relationship with the component-as-array model. However, defining the heap as a function of type [Object → [Name → $\texttt{Object}_\texttt{v}$]] may seem more intuitive. In this definition we first fix an object and then we ask for a field name to obtain its value. As the functions are total (required by PVS), both definitions are in fact equivalent. This means that every field should be defined at every object. This is of course not realistic, however, accesses to undefined fields can be handled by a preliminary static analysis.

3.2 Expressions, statements and compositions

We model expressions, statements and their compositions following Meyer's ideas for a semantics for Eiffel [12]. A distinctive aspect of this approach is that expressions and statements are evaluated relative to an object, which is provided together with the heap as argument.

We deal with null-pointer dereferencing in language constructs, as opposed to avoiding it by type constraints. In our experience, the second approach leads to cumbersome specifications because the result of each expression and statement must be checked for definedness before composing them.

There are two syntactic categories: expressions (without side-effects) and statements:

```
Expr : TYPE = {e : [Object_v⊥ , Heap_⊥ → Object_v⊥] |
  ∀ (o : Object_v⊥ , h : Heap_⊥) :
    bottom_or_void?(o, h) ⇒ bottom?(e(o, h))}

Stmt : TYPE = {S : [Object_v⊥ , Heap_⊥ → Heap_⊥] |
  ∀ (o : Object_v⊥ , h : Heap_⊥) :
    bottom_or_void?(o, h) ⇒ bottom?(S(o, h))}
```

To define a semantics for Eiffel, Meyer works with partial functions [12]. In most theorem provers, including PVS, functions have to be total. For this reason we use lifted arguments, to represent undefinedness. The `bottom_or_void?(o,h)` predicate returns `true` if and only if `o` is undefined or void or `h` is undefined. By using predicate subtypes, we ensure that whenever an expression or statement is evaluated in `void` or in an undefined object or state, the result is undefined. This shifts checking for void or bottom from the specification to type correctness obligations that PVS generates automatically.

The expression `Current` (called `this` or `self` in some languages) returns the current object:

```
Current : Expr = λ (o : Object_v⊥ , h : Heap_⊥) :
  IF bottom_or_void?(o, h) THEN bottom ELSE o
```

The operators $\bullet$ and ; compose expressions and statements. If $\mathtt{x}$ is an expression, $\mathtt{S}$ an statement and $\mathtt{r}$ is either of them, we define:

$$\mathtt{S} \,;\, \mathtt{r} = \lambda\ (\mathtt{o} : \mathtt{Object}_{\mathtt{v}\perp},\ \mathtt{h} : \mathtt{Heap}_{\perp}) : \mathtt{r}(\mathtt{o},\ \mathtt{S}(\mathtt{o},\ \mathtt{h}))$$
$$\mathtt{x} \bullet \mathtt{r} = \lambda\ (\mathtt{o} : \mathtt{Object}_{\mathtt{v}\perp},\ \mathtt{h} : \mathtt{Heap}_{\perp}) : \mathtt{r}(\mathtt{x}(\mathtt{o},\ \mathtt{h}),\ \mathtt{h})$$

The normal uses are state compositions $\mathtt{S};\mathtt{T}$ and field access $\mathtt{x} \bullet \mathtt{y}$. The overloading allows us also to write $\mathtt{S}\,;\,\mathtt{x}$, which returns the value of evaluating $\mathtt{x}$ after the statement $\mathtt{S}$, and $\mathtt{x} \bullet \mathtt{S}$, which can be thought as a qualified call of $\mathtt{S}$ from $\mathtt{x}$.

We define in PVS an automatic conversion that translates a name $\mathtt{f}$ into the expression $\lambda\ (\mathtt{o}\text{: }\mathtt{Object}_{\mathtt{v}\perp},\ \mathtt{h}\text{: }\mathtt{Heap}_{\perp})\text{: }\mathtt{h}(\mathtt{f})(\mathtt{o})$ whenever needed. This allows us to express a field access directly as $\mathtt{x} \bullet \mathtt{f}$. We also define a conversion that translates an $\mathtt{o} : \mathtt{Object}$ into $\mathtt{obj}(\mathtt{o}) : \mathtt{Object}_{\mathtt{v}}$, and one that translates an $\mathtt{o} : \mathtt{Object}_{\mathtt{v}}$ into $\mathtt{up}(\mathtt{o}) : \mathtt{Object}_{\mathtt{v}\perp}$, to reduce the amount of syntax.

IF-statements are mapped to IF-expressions in the logic of PVS. For reasons of succinctness we omit the treatment of WHILE.

3.3 Assignments

At its core, an assignment is an update of a heap-function in a particular point (consisting of a field and an object):

$$\begin{array}{l}\mathtt{update}(\mathtt{f} : \mathtt{Name},\ \mathtt{p} : \mathtt{Object},\ \mathtt{h} : \mathtt{Heap},\ \mathtt{q} : \mathtt{Object_v}) : \mathtt{State} = \\ \quad \lambda\ (\mathtt{g} : \mathtt{Name})(\mathtt{o} : \mathtt{Object}) : \\ \quad\ \ \textbf{IF}\ \mathtt{p} = \mathtt{o} \wedge \mathtt{f} = \mathtt{g}\ \textbf{THEN}\ \mathtt{q}\ \textbf{ELSE}\ \mathtt{h}(\mathtt{g})(\mathtt{o})\end{array}$$

Our model forces us to explicitly deal with undefinedness due to dereferencing of void. The $\mathtt{update}$ operation is encapsulated in an operator := that assigns an object $\mathtt{q}$ to the field $\mathtt{f}$ of the object $\mathtt{p}$ in the heap $\mathtt{h}$:[1]

$$\begin{array}{l}:= (\mathtt{f} : \mathtt{Name},\ \mathtt{q} : \mathtt{Object}_{\mathtt{v}}) : \mathtt{Stmt} = \\ \quad \lambda\ (\mathtt{p} : \mathtt{Object}_{\mathtt{v}\perp},\ \mathtt{h} : \mathtt{Heap}_{\perp}) : \\ \quad\quad \textbf{IF}\ \mathtt{bottom_or_void?}(\mathtt{p}) \vee \mathtt{bottom?}(\mathtt{h})\ \textbf{THEN}\ \mathtt{bottom} \\ \quad\quad \textbf{ELSE}\ \mathtt{update}(\mathtt{f},\ \mathtt{obj}(\mathtt{down}(\mathtt{p})),\ \mathtt{down}(\mathtt{h}),\ \mathtt{q})\end{array}$$

If the assignment is made in an undefined state or tries to assign to void, the error is propagated. This is required by the definition of $\mathtt{Stmt}$. We shall use the above variable names throughout the rest of this paper. The next step is to define local assignments $\mathtt{f} := \mathtt{e}$ and qualified assignments $\mathtt{e}_1 \bullet \mathtt{f} := \mathtt{e}_2$. These definitions are not relevant for the development of this paper and we therefore omit them.

An assignment affects a field access if and only if the object where the field is evaluated is the one where the assignment was made and the field being accessed is the one that was assigned to. This is summarised in the following two basic separation and interference properties (both assume that $\mathtt{o}, \mathtt{h}$ is not bottom or void):

Property 1. If $\mathtt{p} \neq \mathtt{o}$ or $\mathtt{f} \neq \mathtt{g}$, then $\mathtt{g}(\mathtt{o}, (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})) = \mathtt{g}(\mathtt{o}, \mathtt{h})$.

[1] In the PVS formalisation we have called this function <=, because := is reserved.

Property 2. If $\mathtt{p} = \mathtt{o}$ and $\mathtt{f} = \mathtt{g}$, then $\mathtt{g}(\mathtt{o}, (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})) = \mathtt{q}$.

The proofs of these two properties amount to expanding the definition of := and applying several case-splits. When the assignment is replaced with a qualified assignment $\mathtt{e_1} \bullet \mathtt{f} := \mathtt{e_2}$, then analogous properties hold, but $\mathtt{p} = \mathtt{o}$ is replaced by $\mathtt{e_1}(\mathtt{o}, \mathtt{h}) = \mathtt{o}$.

One has to explicitly apply properties 1 and 2 as proof steps to reason about the effect of an assignment in the presented semantics. The key condition is $\mathtt{p} = \mathtt{o} \wedge \mathtt{f} = \mathtt{g}$. The latter is a syntactical comparison and thus can be done automatically. However, most of the time comparison between objects can not be discharged automatically, unless we have information about the layout of the heap, see Section 6.

4 The Effect of Assignments on Multidot Expressions

In this section we look at expressions of the form

$$(\mathtt{g_1} \bullet \ \dots \ \bullet \mathtt{g_n})(\mathtt{o},\ (\mathtt{f} := \mathtt{q})(\mathtt{p},\ \mathtt{h})), \qquad (1)$$

where the $\mathtt{g_i}$ and $\mathtt{f}$ are field names, $\mathtt{o}$ and $\mathtt{p}$ are $\mathtt{Object_{v\bot}}$ and $\mathtt{q}$ is of type $\mathtt{Object_v}$. Because undefinedness due to dereferencing void is not an essential part of the discussion, we shall omit it in the rest of the paper.

Properties 1 and 2 describe the result of a very simple multidot, namely one where $\mathtt{n}$ is equal to 1. There, the condition which determines the result is $\mathtt{p} = \mathtt{o} \wedge \mathtt{f} = \mathtt{g}$. In multidot field expressions of arbitrary length a similar condition determines the result, but now it must be taken into account that there can be several places in the path from $\mathtt{o}$ to $(\mathtt{g_1} \bullet \ \dots \ \bullet \mathtt{g_n})(\mathtt{o},\ (\mathtt{f} := \mathtt{q})(\mathtt{p},\ \mathtt{h}))$ such that $\mathtt{p}$ is the origin and the field name is $\mathtt{f}$. Thus we are interested in the set of indexes $\mathtt{k}$ such that:

$$\mathtt{p} = (\mathtt{g_1} \bullet \ \dots \ \bullet \mathtt{g_{k-1}})(\mathtt{o},\ \mathtt{h}) \text{ and } \mathtt{f} = \mathtt{g_k}\,.$$

The properties we present in this section are categorised into *separation rules*, where the assignment has no effect on the multidot field expression, and *interference rules*, where the assignment does have effect on the multidot field expression. Moreover, we now have a choice to look at the heap $\mathtt{h}$ before the assignment, or at the heap $\mathtt{h'} = (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})$ after the assignment. For the separation properties this does not make a difference, but for the interference properties it does.

The properties we derive about multidot expressions in this section are at the meta-level. Although it is possible to use them to reason about a particular multidot in a program, the intended use is to reason about the effects of assignments on recursive data structures. Examples that demonstrate the application are given in Section 5.

To improve readability, the notation for multidot expressions differs from the actual syntax used in PVS. In the last subsection we show the concrete PVS formalisation of a property. We will use graphs representing a portion of the heap $\mathtt{h'}$ to show examples of the properties. In these graphs nodes are objects

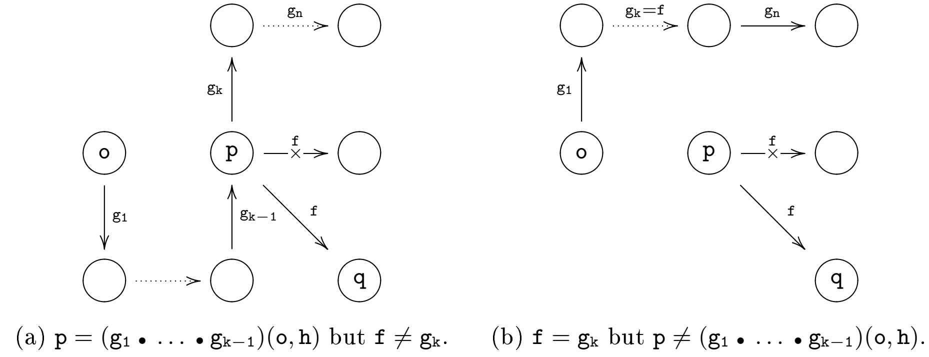
and edges are labelled with an attribute name. An edge $\xrightarrow{\texttt{f}}$ from an object o to an object p means that $\texttt{f}(\texttt{o}, \texttt{h}') = \texttt{p}$. The edge removed by the heap update is depicted as $\overset{\texttt{f}}{-\!\!\times\!\!\rightarrow}$.

4.1 Looking at the heap before the assignment

The assignment in (1) may or may not modify the multidot field expression. Graphically, what matters is whether the edge that has changed belongs to path followed by the multidot field expression or not. A particular edge is determined by its object of origin and the field name. Hence, the condition that determines whether the assignment influences the multidot expression is whether or not the following set is empty:

$$\texttt{K}_{\texttt{pre}} \;\widehat{=}\; \{\, \texttt{k} : \texttt{nat} \mid \texttt{k} < \texttt{n} \wedge \texttt{p} = (\texttt{g}_1 \bullet \ldots \bullet \texttt{g}_{\texttt{k}-1})(\texttt{o}, \texttt{h}) \wedge \texttt{f} = \texttt{g}_{\texttt{k}} \,\}.$$

We start with the case where $\texttt{K}_{\texttt{pre}}$ is empty, i.e., the edge changed by the assignment is not part of the multidot expression, as shown in Figure 1.



(a) $\texttt{p} = (\texttt{g}_1 \bullet \ldots \bullet \texttt{g}_{\texttt{k}-1})(\texttt{o}, \texttt{h})$ but $\texttt{f} \neq \texttt{g}_{\texttt{k}}$. (b) $\texttt{f} = \texttt{g}_{\texttt{k}}$ but $\texttt{p} \neq (\texttt{g}_1 \bullet \ldots \bullet \texttt{g}_{\texttt{k}-1})(\texttt{o}, \texttt{h})$.

Fig. 1: Examples where $\texttt{K}_{\texttt{pre}}$ is empty.

As the edge that changed was not part of the multidot expression, the assignment does have an effect on it.

Property 3. (empty_$\texttt{K}_{\texttt{pre}}$) If $\texttt{K}_{\texttt{pre}}$ is empty, then

$$(\texttt{g}_1 \bullet \ldots \bullet \texttt{g}_{\texttt{n}})(\texttt{o}, \texttt{h}') = (\texttt{g}_1 \bullet \ldots \bullet \texttt{g}_{\texttt{n}})(\texttt{o}, \texttt{h})\,.$$

Now consider the case where $\texttt{K}_{\texttt{pre}}$ is not empty. Figure 2 depicts an example with two indexes i and k in $\texttt{K}_{\texttt{pre}}$ such that $\texttt{k} < \texttt{i}$. If there are several indexes in $\texttt{K}_{\texttt{pre}}$, it means that there are several loops starting at p. The assignment breaks the first edge in these loops. In the heap after the assignment, the edge that joins p with q is determined by the *least* element in $\texttt{K}_{\texttt{pre}}$.

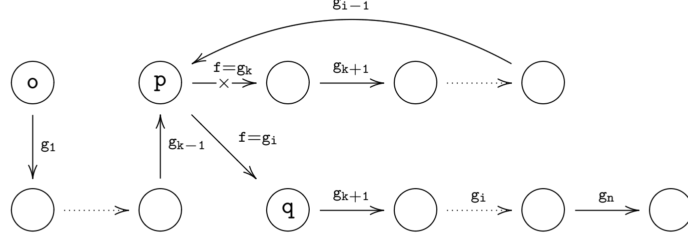


Fig. 2: Example with two indexes $\mathtt{k} < \mathtt{i}$ in $\mathtt{K_{pre}}$.

Property 4. ($\mathtt{min_K_{pre}}$) If $\mathtt{k} = \mathtt{min}(\mathtt{K_{pre}})$, then

$$(g_1 \bullet \ \ldots \ \bullet g_n)(o, h') = (g_{k+1} \bullet \ \ldots \ \bullet g_n)(q, h') \,.$$

Since the assignment may also affect the path that goes from $\mathtt{q}$ to the final value, the right hand side must still be evaluated in $\mathtt{h'}$.

4.2 Looking at the heap after the assignment

Instead of looking at when the multidot expression follows the edge that changed in $\mathtt{h}$, we will now look at when it follows the new edge in $\mathtt{h'}$. That is, we will look at the set:

$$K_{pos} \ \widehat{=} \ \{ k : \mathtt{nat} \mid k < n \wedge p = (g_1 \bullet \ \ldots \ \bullet g_{k-1})(o, h') \wedge f = g_k \}.$$

If the new edge is never traversed, the multidot expression does not change.

Property 5. ($\mathtt{empty_K_{pos}}$) If $\mathtt{K_{pos}}$ is empty, then

$$(g_1 \bullet \ \ldots \ \bullet g_n)(o, h') = (g_1 \bullet \ \ldots \ \bullet g_n)(o, h) \,.$$

Now assume that there is at least one index in $\mathtt{K_{pos}}$. In Figure 3 we see an example with two such indexes $\mathtt{i}$ and $\mathtt{k}$ with $\mathtt{i} < \mathtt{k}$. In this case the result of the multidot expression can be described as either $(g_{k+1} \bullet \ \ldots \ \bullet g_n)(q, h')$ or as $(g_{i+1} \bullet \ \ldots \ \bullet g_k \bullet g_{k+1} \bullet \ \ldots \ \bullet g_n)(q, h')$. If we take the greatest index in $\mathtt{K_{pos}}$, we get the shortest path to the resulting value and since the rest of the edges are not affected by the assignment we can describe the result in terms of $\mathtt{h}$. This is expressed in the following properties.

Property 6. ($\mathtt{forall_K_{pos}}$) For all $\mathtt{k}$ in $\mathtt{K_{pos}}$,

$$(g_1 \bullet \ \ldots \ \bullet g_n)(o, h') = (g_{k+1} \bullet \ \ldots \ \bullet g_n)(q, h') \,.$$

Property 7. ($\mathtt{max_K_{pos}}$) If $\mathtt{k} = \mathtt{max}(\mathtt{K_{pos}})$, then

$$(g_1 \bullet \ \ldots \ \bullet g_n)(o, h') = (g_{k+1} \bullet \ \ldots \ \bullet g_n)(q, h) \,.$$

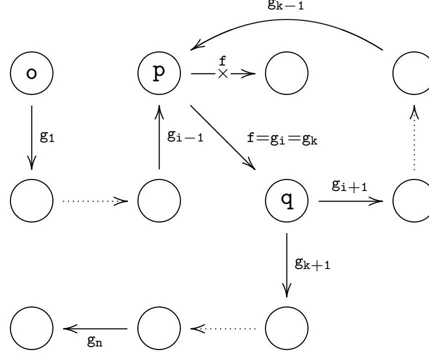


Fig. 3: Example with two indexes $\mathtt{i} < \mathtt{k}$ in $\mathtt{K_{pos}}$.

4.3 PVS formalisation

Given a list of names `fs`, the dot composition of the corresponding attributes is formalised as

```
multidot(fs : list[Name]) : RECURSIVE Expr =
  IF null?(fs) THEN Current
  ELSIF null?(cdr(fs)) THEN car(fs)
  ELSE car(fs) • multidot(cdr(fs))
MEASURE length(fs)
```

Note that because `e • Current = e` does not hold when `e` evaluates to void, we cannot simply append `Current` at the end of the multidot expression.

As an example of the PVS formalisation, we show a property that combines $\mathtt{empty_K_{pos}}$ and $\mathtt{max_K_{pos}}$ in a property at the source code level. Since it is written as an equality between functions, it can be used as a rewrite rule.

```
multidot_after_assignment_pos : LEMMA
  ∀ (f : Name, gs : list[Name], x, e : Expr,
        o : Object_v⊥, h : Heap⊥) :
    ((x • f := e ; multidot(gs))(o, h) =
      LET h′ = (x • f := e)(o, h),
          K_pos = λ (k: below[length(hs)]) :
                     x(o, h) = multidot(take(gs, k))(o, h′) ∧
                     f = nth(gs, k) IN
      IF bottom?(h′) THEN bottom
      ELSIF empty?(K_pos) THEN multidot(gs)(o, h)
      ELSE LET k = max(K_pos) IN
        IF k = length(gs) - 1 THEN e(o, h)
        ELSE (e • multidot(drop(gs, k+1)))(o, h)
```

This property describes in terms of `h` all the possible outcomes of `multidot(gs)` when evaluated in `h′`. If the assignment resulted in an error then the result is an

error. If $\mathtt{K_{pos}}$ is empty then the multidot expression is unchanged. Otherwise, let k be the greatest element in $\mathtt{K_{pos}}$. The result is then as stated in max_$\mathtt{K_{pos}}$ (with a shift of indexes due to lists starting at 0 in PVS). But again because e.Current is not equal to e when evaluated on void, we have to make a special case for when the multidot expression ends exactly at e. There is a similar lemma that combines empty_$\mathtt{K_{pre}}$ and min_$\mathtt{K_{pre}}$.

The intuitive way to prove these properties is by induction on gs. The intention is to reason about the last edge of the multidot expression and use the inductive hypothesis on the path that leads to it. The problem with this approach is that on the non-empty case we have to reason about a list of the form cons(g, gs). Therefore, we get to reason about the first edge, not the last one. To overcome this problem we defined a function multidot_rev that chains the arguments in the reverse order. Then we wrote lemmas that are adapted to work with the reversed list, and we proved them by induction on gs. Finally, the original lemmas were proven using their reversed counterpart by instantiating gs with reverse(gs).

5 Linearised Abstractions

In this section we look at examples of abstract models expressed in terms of multidot field expressions. We call this style of specifying *linearised*, because it is not by recursion on the structure of the abstract model. The properties derived in the previous section provide us a set of tools to reason about the effects of an assignment to a linearised abstraction.

5.1 Paths

The following definition abstracts a path embedded in the heap to a list l of Objects. The ith object in l is the object on the heap that can be accessed by requesting the first i fields describing the path.

```
Path(gs : list[Name], l : list[Object])
    (o : Object_v⊥, h : Heap_⊥) : bool =
  length(gs) + 1 = length(l) ∧
  ∀ (i : below[length(l)]) :
    multidot(take(gs, i))(o, h) = nth(l, i)
```

Due to the possibility of undefinedness, we define the abstractions as predicates about the heap and the current object rather than as functions because in PVS functions must be total.

With the use of the spatial separation lemmas for multidot expressions we can prove the following separation lemma for paths (recall that $\mathtt{h}' = (\mathtt{f} := \mathtt{q})(\mathtt{p}, \mathtt{h})$):

Property 8. If for all $\mathtt{i} < \mathtt{length(l)}$ it holds that $\mathtt{p} \neq \mathtt{nth(l, i)}$ or $\mathtt{f} \neq \mathtt{nth(gs, i)}$, and $\neg\,\mathtt{bottom?(f(p, h))}$, then

$$\mathtt{Path(gs, l)(o, h')} = \mathtt{Path(gs, l)(o, h)}.$$

Thinking again in terms of graphs, this lemma says that if an edge outside the path is modified, then the path is not affected by the assignment. To give an idea of how the multidot rules are applied, we sketch the proof of this lemma.

Proof sketch. We are supposed to show that the Path predicates are logically equivalent. In expanded form, we have to show that the following predicates are equivalent:

$$\forall\ (\mathtt{i_1} : \mathtt{below[length(l)]}) : (\mathtt{g_1} \bullet \ldots \bullet \mathtt{g_{i_1}})(\mathtt{o}, \mathtt{h'}) = \mathtt{nth(l,\ i_1)} \tag{2}$$

$$\forall\ (\mathtt{i_2} : \mathtt{below[length(l)]}) : (\mathtt{g_1} \bullet \ldots \bullet \mathtt{g_{i_2}})(\mathtt{o}, \mathtt{h}) = \mathtt{nth(l,\ i_2)} \tag{3}$$

To show that (2) implies (3), we instantiate $\mathtt{i_1}$ with $\mathtt{i_2}$ and we apply $\mathtt{empty_K_{pos}}$. Then we have to show that $\mathtt{K_{pos}}$ is indeed empty. If this was not the case then there would be a k such that

$$\mathtt{p} = (\mathtt{g_1} \bullet \ldots \bullet \mathtt{g_{i_k}})(\mathtt{o}, \mathtt{h'}) = \mathtt{nth(l,\ k)} \quad \text{and} \quad \mathtt{f} = \mathtt{g_k},$$

which is a contradicts the assumption that p is not in l. For the converse direction, we apply $\mathtt{empty_K_{pre}}$ in an analogous way. □

The interference property for paths describes how a path ending in p can be joined with a path beginning at q:

Property 9. If $\mathtt{p} \notin \mathtt{l_0} +\!\!+ \mathtt{q} +\!\!+ \mathtt{l_1}$ and $\mathtt{c} = \mathtt{car}(\mathtt{l_0} +\!\!+ \mathtt{p})$, then

$$\mathtt{Path}(\mathtt{gs_1} +\!\!+ \mathtt{f} +\!\!+ \mathtt{gs_2},\ \mathtt{l_0} +\!\!+ \mathtt{p} +\!\!+ \mathtt{q} +\!\!+ \mathtt{l_1})(\mathtt{c},\ \mathtt{h'}) =$$
$$(\mathtt{Path}(\mathtt{gs_1},\ \mathtt{l_0} +\!\!+ \mathtt{p})(\mathtt{c},\ \mathtt{h}) \ \wedge\ \mathtt{Path}(\mathtt{gs_2},\ \mathtt{q} +\!\!+ \mathtt{l_1})(\mathtt{q},\ \mathtt{h}))$$

The proof uses the multidot rules $\mathtt{empty_K_{pos}}$ and $\mathtt{max_K_{pos}}$ for the implication from left to right and it uses the rules $\mathtt{empty_K_{pre}}$ and $\mathtt{min_K_{pre}}$ from right to left. We omit the proof sketch of this property for reasons of space.

An important point about the proofs using linearised abstractions is that the induction is encapsulated in the rules about multidot expressions; to prove the above properties, we did not apply induction.

5.2 Example: verification of an in-place list reversal algorithm

The Path abstraction can be specialised by Path(g, l), which instantiates the regular Path with a list of g-fields. By requiring the last node of Path(next, l) to point to void, we obtain an abstraction for lists on the heap:

```
List(l : list[Object])
    (o : Object_v⊥, h : Heap⊥) : bool =
  Path(next, l)(o, h) ∧
  IF cons?(l) THEN void?(next(last(l), h))
  ELSE void?(o)
```

Note that $\texttt{List}(\texttt{null})(\texttt{o}, \texttt{h})$ is true iff $\texttt{void?}(\texttt{o})$ is true, i.e. an empty list is represented by void. Similar separation and interference properties as the ones for `Path` can be proved for `List`.

To prove the correctness of the annotated in-place list reversal algorithm listed in Figure 4, we use standard Hoare-style reasoning. The annotations have type $\texttt{Asrt} : [\texttt{Object}_{\texttt{v}\bot}, \texttt{Heap}_\bot \to \texttt{bool}]$ and a Hoare-triple has the following meaning for $\texttt{P}, \texttt{Q} : \texttt{Asrt}$ and $\texttt{S} : \texttt{Stmt}$:

$$\{\texttt{P}\}\ \texttt{S}\ \{\texttt{Q}\} \ \widehat{=}\ \begin{array}{l} \forall\,(\texttt{o} : \texttt{Object}_{\texttt{v}\bot}, \texttt{h} : \texttt{Heap}_\bot) : \\ \quad \texttt{P}(\texttt{o}, \texttt{h}) \Rightarrow \texttt{Q}(\texttt{o}, \texttt{S}(\texttt{o}, \texttt{h})) \end{array}$$

As can be seen in Figure 4, the current object `o` and the updated heap $\texttt{S}(\texttt{o}, \texttt{h})$ distribute over the connectives. So, the actual work to verify the correctness of the list reversal algorithm amounts to simplifying expressions of the form $(\texttt{g} \bullet \texttt{List}(\texttt{l}))(\texttt{o}, (\texttt{e}_1 \bullet \texttt{f} := \texttt{e}_2)(\texttt{o}, \texttt{h}))$. By expanding the definitions of dot and assignment, this can be brought into the form of $\texttt{List}(\texttt{l})(\texttt{o}', (\texttt{f} := \texttt{q})(\texttt{p}, \texttt{h}'))$, on which the separation and interference rules for the `List` abstraction can be applied.

```
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧ a • List(As)(o, h) }
b := void;
WHILE (λ(o, h) : ¬void?(a(o, h))) DO
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧
   ∃(as, bs : list[Object]) :
     (a • List(as))(o, h) ∧ (b • List(bs))(o, h) ∧
     disjoint?(as, bs) ∧ append(reverse(as), bs) = reverse(As) }
  tmp := a;
  a := a • next;
  tmp • next := b;
  b := tmp;
OD
{ λ(o, h) : ¬bottom_or_void?(o, h) ∧ (b • List(reverse(As)))(o, h) }
```

Fig. 4: In-place list reversal.

5.3 Other data structures

The linearised specification approach exemplified in the previous two sections can also be applied to other recursive data structures. Consider for example binary trees that store a value in each node:

```
binary_tree[σ]: TYPE = {leaf, node(v : σ, l, r : binary_tree)}
```

It is straightforward to define a predicate

```
get_node(bt : binary_tree[σ], path : list[Name], v : σ) : bool
```

that says that by traversing `bt` in the order specified by `path`, we arrive at `v`. Basically `get_node` maps each constructor application to the corresponding field

name. We can now describe a binary tree on the heap by mapping each of its nodes to a multidot field access:

```
binary_tree_abstraction(bt : binary_tree[Object])
                       (o : Object_v⊥, h : Heap⊥) : bool =
  ∀ (x : Object, path : list[Name]) :
    get_node(bt, path, x) ⇒
      multidot(path)(o, h) = x
```

From the properties about multidot expressions presented in Section 4 one can obtain separation and interference lemmas for binary trees.

The same ideas can be applied to other tree-like structures. First make a linearised abstraction of the data structure: obtain the path from the root to each of its elements and use that path to describe the pointer structure in terms of multidot expressions. Then use the properties of Section 4 when reasoning about assignments. Data structures with loops can also be specified, e.g., a circular list is just a path that starts and ends in the same object.

6 Evaluation and Future Work

A natural way to define abstractions is by means of recursion on the structure of the abstract model. We single out the work by Mehta and Nipkow that uses this approach to verify several pointer programs [11]. The advantage of using induction is that it is a familiar general-purpose method that is integrated in the theorem prover. Much work has been devoted to automate proofs by induction, in particular to heuristics to instantiate the inductive hypothesis, e.g. *rippling* [4]. In the inductive approach one still has to reason about the effect of the assignments to the data structure, whereas using the rules given in Section 4 the focus is on when to apply each rule and in finding the extrema of the K-sets, which requires an instantiation.

Our experience is that both approaches require a comparable amount of proof work. However, there is still work to be done on investigating specialised version of the assignment rules and on the integration with the theorem prover as tactics. For example, if we know that there is no loop on a multidot expression, as is the case in tree-like structures, then we also know that the K-sets are either empty or have only one element. This eliminates the need to find the minima or the maxima.

Because both approaches lead to definitions that are essentially equivalent, the same properties hold. Hence, our approach can be seen as a complement rather than a replacement of inductive reasoning.

Reasoning about assignments ultimately reduces to reasoning about object equality. Therefore, this framework would benefit from knowledge about the layout of the memory. The separation rules are used to provide local reasoning, but they are not a primitive of the logic as the star conjunct is in Separation Logic [14] (see also Section 7). Hubert and Marché [9] propose a static separation analysis and show how it can be integrated in the component-as-array modelling.

They split the heap into regions that are inferred by the separation analysis and accordingly relabel the field names as a combination `f_r` of the old field name `f` and a region `r`. This could be integrated into our model, for example by redefining the heap for example as

$$\texttt{Heap : } \textbf{\textsc{type}} = [\texttt{Region, Name} \rightarrow [\texttt{Object} \rightarrow \texttt{Object}_{\texttt{v}}]]$$

When it is inferred that two objects `x` and `y` lie in separate regions, the comparison between them can be avoided and the separation lemmas can be applied automatically.

7 Related Work

A first version of some of the rules presented in Section 4 first appeared in Tamalet's Master's thesis [16].

In the seminal work of Bornat [3] and also in the work by Meyer on a semantics for Eiffel [12], pointer structures on the heap are related with abstract models via repeated composition of field requests. This has been a source of inspiration for this paper. Bornat and Meyer both define a sequence closure operator that repeatedly requests a series of (the same) fields, yielding the list of objects that is traversed on the heap. This is essentially the same as our `Path` abstraction of Section 5.2. In this paper we have given a complete and formalised overview of the effects of assignments to arbitrary multidot field expressions. A treatment of the sequential operator in the context of Eiffel has been given in an unpublished work by Blanco and Castro [2], restricted to the case of lists.

A perhaps more natural way to define abstractions is by the use of recursion on the structure of the abstract model. Mehta and Nipkow [11] used this approach to verify the correctness of several pointer programs. We have compared the inductive approach and the linearised approach in Section 6.

Hoare and Jifeng [8] introduce a framework for the formulation of assertions about objects and pointers based on trace model of graphs and process algebra. They use a graphical notation very similar to the one used in this paper. However, their model uses graph transformations to describe the changes to the state whereas we use an operational semantics.

Our rules about an assignment followed by a multidot are meta-level properties of the language. To enable this meta-level reasoning we introduced a function `multidot` that maps a list of `Names` to a suitable expression, which is essentially a deep embedding of multidot expressions. The rules about multidot expressions are a reflection of the properties 1 and 2. For an instructive paper on reflection with examples in PVS we refer to [18].

7.1 Local reasoning

Local reasoning is the key to scalability in formal verification of programs. The way the heap is modelled in our framework is based on the component-as-array modelling idea of Burstall [5]. Refinements of this modelling have been used as

the core of weakest pre-condition calculus-based tools such as Krakatoa for the verification of Java programs, and Caduceus for the verification of C programs [7, 10]. A separation analysis tailored to integration with the component-as-array modelling has been proposed by Hubert and Marché [9]. Future work on the integration of this analysis with our work has been discussed in Section 6.

A well-studied approach to obtain local reasoning is that of Separation Logic, proposed by Reynolds [14], which can be seen as a radical refinement of Burstall's idea. In Separation Logic disjointness of portions of the heap is made explicit in the logic. Its frame rule allows one to reason about just the relevant portion of the heap that a piece of code manipulates and later augment it with the rest of the heap. So far, no concrete case studies on industrial software make use of Separation Logic, but there is ongoing research on its automation, see e.g. [6, 1]. An implementation of [1] has been developed inside the theorem prover HOL by Tuerk [17].

A related line of research is Region Logic, whose goal it is to preserve the local reasoning of Separation Logic, but without using non-standard semantics of Hoare-triples. See [15] for recent work.

8 Conclusions

In this paper we have presented a novel approach to reason about assignments in recursive data structures. We have shown how recursive pointer structures can be described in terms of paths obtained by a series of field accesses. We have provided a formal model of these paths as multidot expressions and we have proved a set of rules that describe how an assignment can affect them. Using these rules we have derived separation and interference lemmas for lists and verified an in-place list reversal algorithm. A complete formalisation of the presented work has been carried out in the PVS theorem prover. We have also shown how to apply this approach to reason about other data structures and we have compared it with the standard inductive approach.

Acknowledgments. The authors would like to thank Marko van Eekelen and Sjaak Smetsers for their insightful comments on a draft version of this paper and the anonymous reviewers for their comments.

References

1. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *FMCO'06: Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.
2. J. Blanco and C. Pablo. A semantics for proving class correctness. unpublished, 2005.
3. R. Bornat. Proving pointer programs in Hoare logic. In R. Backhouse and J. Oliveira, editors, *MPC'00: Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

4. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, 2005.
5. R. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence 7*, pages 22–50. Edinburgh University Press, 1972.
6. D. Distefano and I. Filipović. Memory leaks detection in Java by bi-abductive inference. In D. Rosenblum and G. Taentzer, editors, *FASE'10: Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2010.
7. J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *CAV'07: Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
8. C. A. R. Hoare and H. Jifeng. A trace model for pointers and objects. In R. Guerraoui, editor, *ECOOP'99: European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1999.
9. T. Hubert and C. Marché. Separation analysis for deductive verification. In W. Damm and H. Hermanns, editors, *HAV'07: Heap Analysis and Verification*, pages 81–93, Braga, Portugal, 2007.
10. C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *TPHOLs'05: Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2005.
11. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.
12. B. Meyer. Towards practical proofs of class correctness. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, *ZB'03: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 359–387. Springer, 2003.
13. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, 2001.
14. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
15. S. Rosenberg, A. Banerjee, and D. A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. to appear.
16. A. Tamalet. Yet another semantics for proving class correctness. Master's thesis, Universidad Nacional de Rosario, Argentina, 2006.
17. T. Tuerk. A formalisation of Smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs'09: Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 469–484. Springer, 2009.
18. F. W. von Henke, S. Pfab, H. Pfeifer, and H. Rueß. Case studies in meta-level theorem proving. In J. Grundy and M. C. Newey, editors, *TPHOLs'98: Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 461–478. Springer, 1998.