Shape and Content
Incorporating Domain Knowledge into Shape Analysis

D. Calvanese\textsuperscript{1}, T. Kotek\textsuperscript{2}, M. Šimkus\textsuperscript{2}, H. Veith\textsuperscript{2}, and F. Zuleger\textsuperscript{2}

\textsuperscript{1} Free University of Bozen-Bolzano
\textsuperscript{2} Vienna University of Technology

Abstract. The verification community has studied dynamic data structures primarily in a bottom-up way by analyzing pointers and the shapes induced by them. Recent work in fields such as separation logic has made significant progress in extracting shapes from program source code. Many real world programs however manipulate complex data whose structure and content is most naturally described by formalisms from object oriented programming and databases. In this paper, we attempt to bridge the conceptual gap between these two communities. Our approach is based on description logic, a widely used knowledge representation paradigm which gives a logical underpinning for diverse modeling frameworks such as UML and ER. We show how description logic can be used on top of an existing shape analysis to add content descriptions to the shapes. Technically, we assume that we have separation logic shape invariants obtained from a shape analysis tool, and requirements on the program data in terms of description logic. Thus, we obtain a modular description logic based verification methodology which is able to exploit shape information.

1 Introduction

The manipulation and storage of complex information in imperative programming languages is often achieved by dynamic data structures. The verification of programs with dynamic structures however is notoriously difficult, and is a highly active area of current research. This paper aims to put a new perspective on this problem. We discuss how the analysis of the shape can be complemented by an analysis of the content to be stored.

Shape analysis is concerned with the analysis of pointers and of the structures induced by them. Recent years have seen considerable progress in automatic methods for inferring basic shape properties such as lists and trees and variations thereof (cyclic lists, doubly-linked lists, etc.). This success has been enabled by succinct formalisms for representing heap structures, most notably separation logic \cite{25,16}. With a few exceptions e.g. \cite{20,21} the majority of papers on shape analysis has focused on the (graph-theoretic) shape of the data structures rather than their information content. For instance, classical shape analysis does not capture simple concepts such as “a list of students where each student has a list of teachers” but only the more combinatorial concept “list of lists”.


The question of content representation has been studied by other disciplines, in particular databases, modeling and knowledge representation. They typically model reality by classes and binary relationships between these classes. For example, the database community uses entity-relationship (ER) diagrams, UML diagrams have been studied in requirements engineering, and the knowledge representation community uses description logic for various modeling tasks. Importantly, these formalisms address an abstraction layer which is intermediary between the program requirements and the implementation. In particular, they are not intended to specify the shape of the data structures but rather their information content.

Problem Statement. We believe that the verification of complex programs with dynamic data structures needs to combine methods from shape analysis and content representation. Complex software projects typically come with extensive documentation or models which contain a description of data structure content e.g. in UML, ER, diagrams etc. But even in an informal setting, the programmers always have a high-level concept of what they want to achieve with their program, in particular what information has to be represented in the program and how this information is stored in data structures. The goals of verification are often related to the content and its representation, e.g. the verification of complex heap invariants in a file system, the adherence to security policies, etc. Thus, the technical challenge is to find a verification methodology along with a suitable formalism that bridges the gap between shape analysis and content representation.

Our Approach and Methodology. In this paper we explore the use of a description logic as an assertion language for verifying high-level properties of programs on top of a preceding shape analysis. Description Logics (DLs) are a well established and highly popular family of logics for representing knowledge in artificial intelligence [2]. DLs are mature and well understood, they have good algorithmic properties and enjoy efficient reasoners. DLs vary in expressivity and complexity, and are usually selected according to the expressivity needed to formalize the given target domain. Note that DLs can be used as a precise framework for reasoning over UML class diagrams and ER diagrams [61]. Moreover, DLs are the logical backbone of the Web Ontology Language (OWL) for the Semantic Web [22]. In this paper we employ a very expressive description logic, based on the so called \textit{ALCHOIF}, which we specifically tailor to better support reasoning about complex pointer structures.

This paper consists of two parts: In Section 2, we study description logic as a formalism for program invariants, and its relationship to separation logic. In particular, we show that shape invariants in separation logic can be translated into description logic in such a way that the translation is implied by the shape invariant (but possibly weaker than the shape invariant). In Section 3, we describe a program model for sequential imperative programs without procedures, and describe a systematic method for the verification of description logic invariants.

We thus obtain a semi-manual methodology which leverages the results of shape analysis for content analysis. As presented here, our methodology assumes
human guidance, and it is part of future work to automatize the role of the user. The verification method then proceeds in two steps:

1. **Shape Analysis.** We assume that the user has specified a set of program locations $L$ to be annotated in a Hoare-style proof; typically, $L$ is the set of loop headers as well as the beginning and end of the program. The user then annotates the locations $L$ with shape predicates, possibly supported by a shape analysis tool which proves the validity of the shape predicates. We choose a specific separation logic ([8]) for the shape annotations, but our methodology is independent of this choice.

2. **Content Analysis.** The user translates the shape invariants into description logic as discussed above, and annotates the program locations in $L$ with the description logic invariants describing content. The validity of the description logic predicates is then reduced to a DL decision procedure.

Note that our method clearly separates shape analysis from content analysis; throughout this paper, we treat shape analysis as a black box.

1.1 **Running Example: Employees, Projects, Managers**

Our running example will be a simple information system for a company. We start with an informal description of the programmers' intention. The company has employees working on projects. The employees and projects are stored in two lists, both using the next pointer. The heads of the two lists are `projHead` and `emplHead` respectively. Each employee in the list of employees has a pointer `worksFor` to a project on the list of projects, indicating the project that the employee is working on (or to null, in case no project is assigned to that employee). Each project in the list has a pointer `managedBy` to the employee list, indicating the manager of the project (or to null, in case the project does not have a manager). Every employee managing a project should also work for that project. Some of the employees are marked as managers via a Boolean field `isManager`, and only they can manage projects. We will refer to these properties as the system invariants.

The programmer has written two program (stated below), which she wants to verify. We will refer to the program on the left by $S_1$ and on the right by $S_2$.

\[
\begin{align*}
    l_b: & \quad \text{proj}.\text{managedBy} := \text{emp}; \\
         & \quad \text{emp}.\text{worksFor} := \text{proj}; \\
         & \quad \text{emp}.\text{isManager} := \text{T}; \\
    l_e: & \quad \text{end}; \\
    l_b: & \quad \text{proj} := \text{new}; \\
         & \quad \text{proj}.\text{next} := \text{projHead}; \\
         & \quad \text{projHead} := \text{proj}; \\
         & \quad \text{e} := \text{emplHead}; \\
    l_e: & \quad \text{while } \neg (\text{e} = \text{null}) \text{ do} \\
         & \quad \quad \text{if } (\text{e}.\text{worksFor} = \text{null}) \text{ then} \\
         & \quad \quad \quad \text{e}.\text{worksFor} := \text{proj}; \\
         & \quad \quad \quad \text{e} := \text{e}.\text{next}; \\
         & \quad \quad \text{od} \\
    l_e: & \quad \text{end};
\end{align*}
\]
The programmer has the following intuition about her programs: The code $S_1$ assigns the employee $emp$ to manage project $proj$. If $emp$ is not a manager already, $emp$ is promoted. The code $S_2$ adds a new project $proj$ to the project list, and assigns to it all employees in the employee list which are not assigned to any project.

For both pieces of code the programmer wants to verify that the system invariants are true after the execution of the code, if they were true in the beginning (1). Note that during the execution of the code, they might not be true! Additionally the programmer wants to verify that after executing $S_2$ the project list has been extended by $proj$, the employee list still contains the same employees and indeed all employees who did not work for a project before now work for project $proj$ (2). In this paper we will formally prove the correctness of $S_2$ following our verification methodology discussed in the introduction. In Sections 2.4 and Section 2.5 we describe how the description logic $L$ can be used for specifying the verification goals (1) and (2). In Section 3.4 we state verification conditions that allow to conclude the correctness of (1) and (2) for program $S_2$.

2 Description Logic for Invariant Specification

2.1 Structures

We use ordinary first order structures to represent memory in a precise way. A structure (or, interpretation) is a tuple $M = (M, \tau, \cdot)$, where (i) $M$ is an infinite set (the universe), (ii) $\tau$ is a set of constants and relation symbols with an associated non-negative arity, and (iii) $\cdot$ is an interpretation function, which assigns to each constant $c \in \tau$ an element $c^M \in M$, and to each $n$-ary relation symbol $R \in \tau$ an $n$-ary relation $R^M$ over $M$. In this paper, each relation is either unary or binary (i.e. $n \in \{1, 2\}$). Given $A \subseteq M$ and a binary $R^M$, $R^M$ is a (total) function from $A$ if there is exactly one $e'$ with $(e, e') \in R^M$ for every $e \in A$. For such $R^M$, we use $R^M(e)$ to denote $e'$.

2.2 Memory Structures

A Memory structure will describe a snapshot of the heap and the local variables. To this end, we assume a set $\tau_{\text{var}} \subseteq \tau$ of constants and a set $\tau_{\text{fields}} \subseteq \tau$ of binary relation symbols. We will later employ these symbols for variables and fields in programs. A memory structure is a structure $M = (M, \tau, \cdot)$ that satisfies the following conditions:

1. $\tau$ includes the constants $o_{\text{null}}, o_T, o_F$.
2. $\tau$ has the unary relations $\text{Addresses}, \text{Alloc}, \text{MemPool}$, and $\text{Aux}$.
3. $\text{Aux}^M = \{o_{\text{null}}^M, o_T^M, o_F^M\}$ and $|\text{Aux}^M| = 3$.
4. $\text{Addresses}^M \cap \text{Aux}^M = \emptyset$ and $\text{Addresses}^M \cup \text{Aux}^M = M$.
5. $\text{Alloc}^M \cap \text{MemPool}^M = \emptyset$ and $\text{Alloc}^M \cup \text{MemPool}^M \subseteq \text{Addresses}^M$.
6. $c^M \in M \setminus \text{MemPool}^M$ for every constant $c$ of $\tau$. 

(7) For all \( f \in \tau_{\text{fields}} \), \( f^M \) is a function from \( \text{Addresses}^M \) to \( M \setminus \text{MemPool}^M \).

(8) If \( e \in \text{MemPool}^M \), then \( f^M(e) \in \{ \text{null}^M, \text{F}^M \} \).

(9) \( R^M \subseteq (M \setminus \text{MemPool}^M)^n \) for every \( n \)-ary \( R \in \tau \setminus (\{ \text{MemPool} \} \cup \tau_{\text{fields}}) \), \( n = 1, 2 \).

(10) \( \text{Alloc}^M \) and \( \text{Addresses}^M \setminus \text{MemPool}^M \) are finite. \( \text{MemPool} \) is infinite.

The intuition behind memory structures is as follows. Variables in programs will either have a Boolean value or be pointers. Thus, to represent null and the Boolean values \( \text{T} \) and \( \text{F} \), we employ the auxiliary relation \( \text{Aux}^M \) storing 3 elements corresponding to the 3 values. Clearly, \( \text{Addresses}^M \) represents the memory cells. The relation \( \text{Alloc}^M \) contains the allocated cells. \( \text{MemPool}^M \) contains the cells which are not allocated, do not have any field values other than null and \( \text{F} \), are not pointed to by any field, do not participate in any other relation and do not interpret any constant (see (6-9)). The memory cells in \( \text{MemPool} \) are the candidates for allocation during the run of a program. \( \text{Addresses}^M \setminus \{ \text{Alloc}^M \cup \text{MemPool}^M \} \) is the set of memory cells which are not allocated, but are pointed to from cells in \( \text{Alloc}^M \). Since the allocated memory should by finite at any point of the execution of a program, \( \text{Alloc}^M \) and \( \text{Addresses} \setminus \{ \text{Alloc}^M \cup \text{MemPool}^M \} \) are finite (see (10)). However, the available memory \( \text{Addresses}^M \) and the memory pool \( \text{MemPool}^M \) are infinite. Finally, each cell is seen as a record with exactly the fields \( \tau_{\text{fields}} \).

2.3 The Description Logic \( \mathcal{L} \)

Domain information in description logics is expressed by means of atomic concepts, atomic roles, constants (also known as individuals), and complex formulae built from them. The various constructors available to build formulae determine the particular description logic, giving rise to a wide family of logics with varying expressivity and complexity of reasoning. The semantics to formulae is given in terms of structures, where atomic concepts and atomic roles are interpreted as unary and binary relations in a structure, respectively, and constants are interpreted as elements in the structure’s universe.

Complex formulae can be used to create and interrelate complex concept descriptions. For instance, the (complex) concept \( \text{Employee} \sqcap \exists \text{worksFor}.\text{Project} \), where \( \text{Employee} \) and \( \text{Project} \) are atomic concepts and \( \text{worksFor} \) is an atomic role, describes persons who work for some project. We can use the formula \( \exists \text{worksFor}.\text{Project} \subseteq \text{Employee} \) to express the requirement that every object working for a project must be an employee. In this paper, we will use complex concepts and formulae to express “lower-level” requirements on the content of data structures in a program. E.g. for a program operating on a list of employees and a list of projects, we may use \( \text{EmpList} \sqcap \exists \text{worksFor}.\text{ProjList} \) to refer to elements of the employee list whose value of the \( \text{worksFor} \) field appears in the project list.
We next make these intuitions more formal by introducing the following expressive description logic $\mathcal{L}$.

**Definition 1 (Syntax of $\mathcal{L}$).** The set of roles and concepts of $\mathcal{L}$ is defined inductively as follows:

- every unary relation symbol is a concept (called atomic concept);
- every constant symbol is a concept (called nominal concept);
- every binary relation symbol is a role (called atomic role);
- if $r$ and $s$ are roles, then $r \sqcup s$, $r \cap s$, $r \setminus s$ and $r^-$ are also roles;
- if $C$ and $D$ are concepts, then so are $C \sqcap D$, $C \sqcup D$, and $\neg C$;
- if $r$ is a role and $C$ is a concept, then $\exists r.C$ is also a concept;
- if $C$ and $D$ are concepts, then $C \times D$ is a role (called product role).

The set of formulae of $\mathcal{L}$ is defined inductively as well.

- The following are formulae:
  - $C \subseteq D$ (concept inclusion), where $C, D$ are concepts;
  - $r \subseteq s$ (role inclusion), where $r, s$ are roles;
  - $\text{func}(r)$ (functionality assertion), where $r$ is a role;
  - if $\varphi$ and $\psi$ are formulae, then $\varphi \land \psi$, and $\neg \psi$ are formulae;
  - If $\varphi$ and $\psi$ are formulae, then $\varphi \land \psi$, and $\neg \psi$ are formulae.

**Definition 2 (Semantics of $\mathcal{L}$).** The semantics to an $\mathcal{L}$-formula $\varphi$ is given in terms of structures $\mathcal{M} = (M, \tau, \cdot)$, where $\tau$ contains all atomic concepts, atomic roles and constants occurring in $\varphi$. The function $\cdot^\mathcal{M}$ is extended to the remaining concepts and roles inductively below. The satisfaction relation $\models$ between structures and formulae is also given in below.

$$(C \sqcap D)^\mathcal{M} = C^\mathcal{M} \cap D^\mathcal{M}$$

$$(C \sqcup D)^\mathcal{M} = C^\mathcal{M} \cup D^\mathcal{M}$$

$$(-C)^\mathcal{M} = M \setminus C^\mathcal{M}$$

$$(r \sqcup s)^\mathcal{M} = r^\mathcal{M} \cup s^\mathcal{M}$$

$$(r \cap s)^\mathcal{M} = r^\mathcal{M} \cap s^\mathcal{M}$$

$$(r \setminus s)^\mathcal{M} = r^\mathcal{M} \setminus s^\mathcal{M}$$

$$(\neg r)^\mathcal{M} = \{ (e, e') \mid (e', e) \in r^\mathcal{M} \}$$

$$(C \times D)^\mathcal{M} = C^\mathcal{M} \times D^\mathcal{M}$$

$$(\exists r.C)^\mathcal{M} = \{ e \mid \exists e' : (e, e') \in r^\mathcal{M} \}$$

If $\mathcal{M} \models \varphi$, then $\mathcal{M}$ is a model of $\varphi$. We write $\psi \models \varphi$ if every model of $\psi$ is also a model of $\varphi$.

For the remainder of the paper we will use the following abbreviations:

$\top = C \sqcup \neg C$, where $C$ is an arbitrary atomic concept and $\bot = \neg \top$; $\alpha \equiv \beta$ for the formula $\alpha \sqsubseteq \beta \land \beta \sqsubseteq \alpha$; $\exists r$ for the concept $\exists r.\top$; $(o, o')$ for the role $o \times o'$; and $\varphi \lor \psi$ and $\varphi \rightarrow \psi$ for formulae $\neg(\neg\varphi \land \neg\psi)$ and $\neg\varphi \lor \psi$, respectively. Note that $\top^\mathcal{M} = M$ and $\bot^\mathcal{M} = \emptyset$ for any structure $\mathcal{M} = (M, \tau, \cdot)$.

---

3 In DL terms, $\mathcal{L}$ corresponds to Boolean $\mathcal{ALCHOIF}$ knowledge bases with the additional support for role intersection, role union, role difference and product roles.
Definition 3 (Satisfiability and implication in memory structures). An \( L \)-formula \( \varphi \) is satisfiable in a memory structure if there is a memory structure \( M \) such that \( M \models \varphi \). We write \( \psi \models_m \varphi \) if \( M \models \psi \) implies \( M \models \varphi \) for every memory structure \( M \).

The following is an easy consequence of the existing results in the description logic literature.

**Theorem 1.** Checking whether an \( L \)-formula is satisfiable in a memory structure is NExpTime-complete.

**Proof (Sketch).** The upper bound can be shown by reducing the problem to finite satisfiability in \( C_2 \), the two-variable fragment of the first-order logic with counting quantifiers [24]. The lower bound comes from the NExpTime-hardness of satisfiability in the DL \( ALC\text{-}HT(O) \) [27].

Clearly, since \( \psi \models_m \varphi \) iff \( \psi \land \neg \varphi \) is not satisfiable in a memory structure, we get that deciding \( \psi \models_m \varphi \) for a pair \( \psi, \varphi \) of \( L \)-formula is coNExpTime-complete.

In the rest of the paper, we will only be interested in satisfiability and implication of formulae in memory structures. Thus, the term satisfiability will always refer to satisfiability in a memory structure, and we will use \( \models \) instead of \( \models_m \).

2.4 Expressing Content Invariants in \( L \)

Here we show how content requirements can be expressed in \( L \). We start with the formulae describing the content and only then move to the shape.

**Employees, Projects, Managers** We can now make the example from Section 1.1 more precise. The concepts \( EmplList \) and \( ProjList \) are interpreted as the sets of elements in the employee list resp. the project list. \( managedBy \), \( isManager \) and \( worksFor \) are roles. \( o_{emplHead} \) and \( o_{projHead} \) are the constants which correspond to the heads of the two lists. The invariants of the systems are:

1. The project and employee lists are allocated: \( ProjList \sqcup \EmplList \sqsubseteq \text{Alloc} \)
2. Projects and employees are distinct: \( ProjList \cap \EmplList \sqsubseteq \bot \)
3. \( worksFor \) is set to null for projects: \( ProjList \sqsubseteq \exists worksFor.o_{null} \)
4. \( managedBy \) is set to null for employees: \( \EmplList \sqsubseteq \exists managedBy.o_{null} \)
5. \( managedBy \) of projects in the list point to managers:
   \( \exists managedBy^{-}.ProjList \sqsubseteq \EmplList \sqcap \exists isManager.o_T \sqcup o_{null} \)
6. \( worksFor \) of employees in the list point to projects in the list or to null:
   \( \exists worksFor^{-}.EmplList \sqsubseteq ProjList \sqcup o_{null} \)
7. \( isManager \) is a Boolean field: \( \exists isManager^{-}.EmplList \sqsubseteq \text{Boolean} \)
8. The manager of a project must work for the project:
   \( \text{worksFor} \cap (\top \times \EmplList \cap isManager.o_T) \sqsubseteq managedBy \)
Let the conjunction of the invariants be given by $\varphi_{\text{invariants}}$.

Now consider the code $S_2$ from the introduction. The states of the heap before and after the execution of $S_2$ can be related by the following $\mathcal{L}$ formulae. $\varphi_{\text{lists-updt}}$ and $\varphi_{\text{p-assgn}}$. $\varphi_{\text{lists-updt}}$ states that the employee list at the end of the program ($\text{EmplList}$) is equal to the employee list at the beginning of the program ($\text{EmplList}_{\text{gho}}$), and that the project list at the end of the program ($\text{ProjList}$) is the same as the project list at the beginning of the program ($\text{ProjList}_{\text{gho}}$), except that $\text{ProjList}$ also contains the new project $o_{\text{proj}}$. $\text{EmplList}_{\text{gho}}$ and $\text{worksFor}_{\text{gho}}$ are ghost relation symbols, whose interpretations hold the corresponding values at the beginning of $S_2$.

$$\varphi_{\text{lists-updt}} = \text{EmplList}_{\text{gho}} \equiv \text{EmplList} \land \text{ProjList}_{\text{gho}} \cup o_{\text{proj}} \equiv \text{ProjList}$$

$$\varphi_{\text{p-assgn}} = \text{EmplList}_{\text{gho}} \cap \exists \text{worksFor}_{\text{gho}} \cdot o_{\text{null}} \equiv \text{EmplList} \cap \exists \text{worksFor} \cdot o_{\text{proj}}$$

**The Virtual File System** The Virtual File System (VFS)\(^4\) is comprised of various types of objects, arranged in a large number of lists. A directory entry (dentry) corresponds to a path in the file system. Both files and directories have dentries and the dentries are arranged in a tree which is implemented by lists. Each dentry points to a file. The files are partitioned between lists depending on whether they are used or not. A file may have more than one dentry pointing to it, corresponding to different names of the same file. Other elements of the VFS which we do not discuss further here include file system types and files pointers, which are interrelated with the above mentioned elements and stored in various lists and other data structures.

The concepts containing the elements of the lists and tree are respectively $\text{UsedList}$, $\text{UnusedList}$, and $\text{DTree}$. 

dentryFile (dentry\text{F} for short) is a role corresponding to the field pointing from dentries to files. FileId is a role corresponding to a field which is considered to be the Id of the file. The tree $\text{DTree}$ is implemented as a binary tree with three roles for the fields firstChild, nextSibling and parent. Fig. 1 illustrates the VFS. $\text{DTree}$ is represented in the memory as a binary tree with fields $fC = \text{firstChild}$ and $nS = \text{nextSibling}$ (above). We think of it as a tree with unbounded degree (below). $n$ is short for next. Additionally every file in the two lists has a FileId field. The invariants of system:

1. The allocated memory contains three disjoint sets of elements: $\text{UsedList}$, $\text{UnusedList}$ and $\text{DTree}$:

   $$\text{UsedList} \sqcup \text{UnusedList} \sqcup \text{DTree} \subseteq \text{Alloc} \quad \text{UnusedList} \cap \text{DTree} \equiv \bot$$

   $$\text{UsedList} \cap \text{DTree} \equiv \bot \quad \text{UsedList} \cap \text{UnusedList} \equiv \bot$$

---

\(^4\) We present here a simplified description. The terminology used here is different from the one used in the linux kernel. E.g., files are called inodes and file systems are called superblocks in the terminology of VFS.
2. Null fields: $\text{UsedList} \sqcup \text{UnusedList} \sqsubseteq \exists \text{dentryF}.o_{\text{null}} \land \text{DTree} \sqsubseteq \exists \text{FileId}.o_{\text{null}}$

3. The root of $\text{DTree}$ is a directory:
$$\text{DTree} \land \neg \exists \text{firstChild}^- \land \neg \exists \text{nextSibling}^- \land \text{DTree} \sqsubseteq \exists \text{dentryF}.o_{\text{null}}$$

4. A file belongs to the $\text{UsedList}$ if and only if a dentry in $\text{DTree}$ points to it:
$$\exists \text{dentryF}.^- \land \text{DTree} \equiv \text{UsedList}$$

5. If a dentry in $\text{DTree}$ points to a file with the pointer $\text{dentryFile}$, then it has no children in $\text{DTree}$ (otherwise it would necessarily be a directory):
$$\text{DTree} \land \exists \text{dentryF}.^- \land \neg \text{null} \sqsubseteq \exists \neg \text{firstChild}^-$$

6. The $\text{FileId}$ is unique with respect to the lists $\text{UsedList}$ and $\text{UnusedFiles}$:
$$\text{func}(\text{FileId}^- \land \top \times (\text{UsedList} \sqcup \text{UnusedList}))$$

### 2.5 Translating Shape Invariants into $\mathcal{L}$

Here we discuss how to translate shape invariants from a preceding shape analysis into $\mathcal{L}$. First, we discuss how lists and trees can be dealt with in $\mathcal{L}$. Next, we apply this to a popular separation logic fragment.

#### Lists and Trees

A concept $L$ is a **singly linked list from $o_{\text{var1}}$ to $o_{\text{var2}}$ with respect to the field next** if it satisfies the following five conditions. Except for (5), the conditions are expressed fully in $\mathcal{L}$ to the right:

1. $o_{\text{var1}}$ belongs to $L$;
2. $o_{\text{var2}}$ is pointed to by an $L$ element; $α_2^1 = o_{\text{var2}} \sqsubseteq \exists \text{next}^- . L$
3. The next of every element in $L$ points $α_2^3 = L \sqsubseteq \exists \text{next} . (L \sqcup o_{\text{var2}})$ to an $L$ or to $o_{\text{var2}}$;
4. Every $L$ element is pointed to from an $L$ element, except possibly for $\text{var1}$; $α_4^4 = L \sqsubseteq \exists \text{next}^- . L \sqcup o_{\text{var1}}$
5. All of the elements of $L$ are reachable from $o_{\text{var1}}$ via next.

In structures $\mathcal{M}$ satisfying $α_1^1 \land \ldots \land α_4^4$, $L^\mathcal{M}$ contains a list segment from $o_{\text{var1}}^\mathcal{M}$ to $o_{\text{var2}}^\mathcal{M}$, but additionally $L^\mathcal{M}$ may contain disjoint cycles with respect to next. Due to the lack of means to express transitive closure, we only express (5) for lists of length at most $m$, where $m$ is a parameter. Let
$$α_5^{5} = D = \exists ((o_{\text{var1}}, o_{\text{var1}}) \sqcup \text{next}^-) \ldots \exists ((o_{\text{var1}}, o_{\text{var1}}) \sqcup \text{next}^-) . o_{\text{var1}}$$
where $α_5^{5}$ contains $m$ repetitions of $\exists ((o_x, o_x) \sqcup \text{next}^-)$. $D$ is a new atomic concept, which contains the elements at distance at most $m$ from $o_{\text{var1}}$. If $o_{\text{var2}}^\mathcal{M}$ belongs to $D^\mathcal{M}$, then all the elements of $L$ should belong to $D$.

We denote by $α_{ls,m}(\text{var1}, \text{var2}, \text{next}, L)$ the conjunction of $α_3^3, \ldots, α_3^4$ and $α_5^{5}$. We write e.g. $α_{ls,m}(\text{var3}, \text{var4}, \text{next2}, L_2)$ to use other start and end points, pointer or concept. We may write null instead of the second variable.

Note that due to the flexibility of $\mathcal{L}$, it is easy to express variations of singly-linked lists, such as doubly-linked lists, or lists in which every element points to a special head element via a pointer head.

The case of trees is similar. We can write $\mathcal{L}$-formulae which express all the conditions needed to express that $T$ is a tree, except for a connectivity condition. We can define a (linearly sized) formula which expresses that if $T^\mathcal{M}$ is small, then all the elements are reachable from the root.
A Separation Logic Fragment  Here we discuss how a fragment of separation logic can be translated into $\mathcal{L}$. The material in this subsection is not required to understand the remainder of the paper. Our separation logic is $\text{SL}_{\text{lis}}$, which is the logic from [8] with lists and multiple pointer fields, but without trees.

The memory model used in [8] is very similar to our memory structures. The allocated cells of the memory all have the same finite collection of fields (denoted $\text{Fields}$). Let $\text{Add}$ and $\text{Val}$ be disjoint sets. $\text{Add}$ is the set of addresses (locations in the terminology of [8], not to be confused with our use of “locations in a program”). $\text{Val}$ is a set of values which include $\text{nil}$. The description of the memory consists of two parts, a heap and a stack. A heap is a partial function $h : \text{Add} \mapsto (\text{Fields} \to \text{Val} \cup \text{Add})$ which is only defined on a finite subset of $\text{Add}$. A stack is a function from a finite set $\text{Var}$ of local variables $s : \text{Var} \to \text{Val} \cup \text{Add}$.

First we define the logic $\text{SL}$. The syntax of $\text{SL}$ is as follows:

$$\begin{align*}
\text{var}_1, \text{var}_2, \ldots \in \text{Variables} & \quad \Pi := \top | E = E \mid E \neq E \mid \Pi \land \Pi \\
\text{f}_1, \text{f}_2, \ldots \in \text{Fields} & \quad \Sigma := \emptyset | \text{var} \mapsto [f_1 : E_1, \ldots, f_k : E_k] \mid S \ast \Sigma \\
E & := \text{null} \mid \text{var}_1
\end{align*}$$

The semantics of $\text{SL}$ is given by a relation $s, h \models \phi$ where $s \in \text{Stacks}, h \in \text{Heaps}$.

We define $[\text{var}_i]s def = s(\text{x})$ and $[\text{nil}]s def = \text{nil}$, and:

$$s, h \models \text{true} \quad \text{always}$$

$$s, h \models E_1 = E_2 \quad \text{iff} \quad [E_1]s = [E_2]s$$

$$s, h \models E_1 \neq E_2 \quad \text{iff} \quad [E_1]s \neq [E_2]s$$

$$s, h \models \Pi_0 \land \Pi_1 \quad \text{iff} \quad s, h \models \Pi_0 \text{ and } s, h \models \Pi_1$$

$$s, h \models \text{var}_1 \mapsto [f_i : E_i] \quad \text{iff} \quad h = [\text{var}_1]s \mapsto r \text{ where } r(f_i) = [E_i]s$$

$$s, h \models \text{emp} \quad \text{iff} \quad h = \emptyset$$

$$s, h \models \Sigma_1 \ast \Sigma_2 \quad \text{iff} \quad \exists h_0 h_1. h = h_0 \ast h_1 \text{ and } s, h_0 \models \Sigma_0 \text{ and } s, h_1 \models \Sigma_1$$

$$s, h \models \Pi \ast \Sigma \quad \text{iff} \quad s, h \models \Pi \text{ and } s, h \models \Sigma$$

where $h_0 \ast h_1$ enforces there is no address in $\text{Add}$ on which both $h_0$ and $h_1$ are defined, and $h = h_0 \ast h_1$ denotes that $h$ is the union of $h_0$ and $h_1$. Additionally, not all fields in $\text{Fields}$ need to occur in $\text{var}_1 \mapsto [f_i : E_i]$, and those that do not are assigned $\text{nil}$ implicitly.

$\text{SL}_{\text{lis}}$ extends $\text{SL}$ by adding list segments: the syntax is extended by $\Sigma := \cdots | \text{ls}(E_1, E_2)$ and the semantics of $\text{ls}(E_1, E_2)$ is the least fixed point of the predicate given by $(E_1 = E_2 \land \emptyset) \lor (E_1 \neq E_2 \land \exists y. E_1 \Rightarrow [n : y] \ast \text{ls}(y, E_2))$.

Heaps and Stacks vs. Memory Structures. The memory model of separation logic and our memory model are easily translatable. The distinction between values in $\text{Val}$ and addresses in $\text{Add}$ does not play a major role in [8], so we simplify by setting $\text{Val} = \{\text{nil}, \text{true}, \text{false}\}$.

Given $M$ we define $s_M$ and $h_M$ as follows. $\text{Fields}$ is equal to $\tau_{\text{fields}}$ and $\text{Add} = \text{Addresses}$ and we have $M = \text{Val} \cup \text{Add}$ with $\text{nil} = \alpha_M^{\text{null}}, \text{true} = \alpha_M^{\top}$ and $\text{false} = \alpha_M^F$. For every variable $\text{var}_i$, $s(\text{var}_i) = \alpha_M^{\text{var}_i}$. For every $\text{add} \in \text{Add}$ we define $h(\text{add}) = h_{\text{add}}$ as follows: for every $f \in \text{Fields}$, $h_{\text{add}}(f) = f(M(\text{add}))$.

Given $s$ and $h$, we define $M_{s, h} = M$ as follows. The universe $M_{s, h}$ is $\text{Add} \cup \{\alpha_M^{\text{null}}, \alpha_M^{\top}, \alpha_M^F\}$, with $\alpha_M^{\text{null}} = \text{nil}, \alpha_M^{\top} = \text{true}$ and $\alpha_M^F = \text{false}$ and $\text{Addresses}_{M} = \text{Add}$. The set of addresses on which $h$ is defined is $\text{Alloc}_M$. For
every field \( f \in \text{Fields} \), \( f^M = h_f \), where \( h_f(\text{add}) = h(\text{add})(f) \) is the value the field \( f \) of address \( \text{add} \) receives under \( h \). For every variable \( \text{var}_i \), \( \alpha^M_{\text{var}_i,h} = s(\text{var}_i) \).

The above shows how to transform the memory models into each other in a natural way. More precisely:

**Lemma 1.** If \( s \) and \( h \) are a stack and a heap, and if \( s_{M,s,h} \) and \( h_{M,s,h} \) are the stack and heap obtained by applying the two transformations above on \( s \) and \( h \) in order, then \( s_{M,s,h} = s \) and \( h_{M,s,h} = h \).

**Reduction from SLs to L.** For every SLs formula \( \varphi \) and \( m \in \mathbb{N} \) we define an \( L \)-formula \( \alpha^m_\varphi \) such that the following Lemma holds:

**Lemma 2.** For every heap \( h \), stack \( s \), and \( m \in \mathbb{N} \), if \( \varphi \in \text{SLs} \) and \( s, h \models \varphi \), then \( M \models \alpha^m_\varphi \).

It is convenient to define the following notation: if \( \varphi = \Pi! \Sigma \), then there exist \( \beta_1, \ldots, \beta_r \) such that \( \Sigma = \beta_1 \cdots \beta_r \) and the \( \beta_i \) are of one of the forms \( \text{var}_1 \mapsto [f_i : E_i] \) or \( \text{ls}(E_1, E_2) \). We use the concepts \( P_1, \ldots, P_r \) which partition the allocated memory cells according to \( \Sigma \).

First we define the formula \( \alpha_\varphi \) for \( \varphi \in \text{SL} \)

\[
\begin{align*}
\alpha_{E_1 = E_2} &= (o_{E_1} \equiv o_{E_2}) \\
\alpha_{E_1 \neq E_2} &= \neg (o_{E_1} \equiv o_{E_2}) \\
\alpha_{P_1 \sqcup H_1} &= \alpha_{P_1} \land \alpha_{H_1} \\
\alpha_{\text{true}} &= (T \sqsubseteq T) \\
\alpha_{\text{emp}} &= (\text{Alloc} \equiv \bot) \\
\alpha^{E_1} &= \alpha_{E_1} \\
\alpha^{P_1} &= \alpha_{P_1} \\
\alpha^{H_1} &= \alpha_{H_1} \\
\alpha^\varphi &= \alpha^{E_1} \cdots \alpha^{P_r} \\
\end{align*}
\]

For every \( \varphi \in \text{SLs} \), \( s, h \models \varphi \) iff \( M_{s,h} \models \alpha_\varphi \).

For \( \beta_i = \text{ls}(\text{var}_1, \text{var}_2) \), we define \( \alpha^m_{\beta_i} = \alpha^m_{\text{ls}}(\text{var}_1, \text{var}_2, \text{next}, P_i) \). From the discussion above we get that, if the interpretation of \( P_1 \) in \( M \) is a list, then \( M \models \alpha_{\beta_i} \), implying Lemma 2.

**Running Example: Shape Invariants** At the loop header of the program \( S_2 \) from the introduction, the memory contains two distinct lists, namely the employee list and the project list. The employee list is partitioned into two parts, the employees who have been visited in the loop so far, and those that have not. This can be expressed in SLs by the formula:

\[
\varphi_t = \text{ls}(\text{emplHead}, e) \ast \text{ls}(e, \text{nil}) \ast \text{ls}(\text{projHead}, \text{nil}) \ast \text{true}
\]

The translation into \( L \) is

\[
\begin{align*}
\alpha^m_{\varphi_t} &= P_1 \sqcup P_2 \sqcup P_3 \sqcup P_4 \equiv \text{Alloc} \land \alpha^m_{\text{ls}}(\text{emplHead}, e, \text{next}, P_1) \\
&\quad \land \alpha^m_{\text{ls}}(\text{null}, \text{null}, P_2) \land \alpha^m_{\text{ls}}(\text{projHead}, \text{null}, \text{next}, P_3) \\
&\quad \land P_1 \sqcap P_2 \equiv \bot \land P_1 \sqcap P_3 \equiv \bot \land \ldots \land P_3 \sqcap P_4 \equiv \bot \land \top \subseteq \top
\end{align*}
\]

The translation from separation logic assigns the lists and contents concepts \( P_i \) which do not reflect their content. In order to clarify the meaning of \( \alpha^m_{\varphi_t} \), we replace the \( P_i \) with the concepts from Section 2.4 and simplify the formula.
somewhat. The union of $P_1$ and $P_2$ is $\text{EmplList}$, $P_3$ is $\text{ProjList}$. $V\text{List} = P_1$ contains the elements of $\text{EmplList}$ visited in the loop so far. $\alpha^m_{\omega_i}$ can be rewritten:

$$\varphi^m_{\omega_i} = \text{EmplList} \cup \text{ProjList} \subseteq \text{Alloc} \land V\text{List} \subseteq \text{EmplList} \land$$

$$\neg \text{ProjList} \equiv \perp \land \alpha_{b,m}(\text{emplHead}, e, \text{next}, V\text{List}) \land \text{EmplList} \land$$

$$\alpha_{b,m}(e, \text{null}, \text{next}, \text{EmplList} \land \neg V\text{List}) \land$$

$$\alpha_{b,m}(\text{projHead}, \text{null}, \text{next}, \text{ProjList}) \quad (1)$$

### 3 Content Analysis

In this section we present our general methodology. To do so we first introduce our programming language precisely. The general methodology uses a backward propagation of formulae over loopless code.

#### 3.1 Syntax and Semantics of the Programming Language

**Loopless Programs** The syntax of our loopless programming language is:

- $e :: var.f \mid var \mid \text{null}$
- $b :: (e_1 = e_2) \mid \sim b \mid (b_1 \text{ or } b_2) \mid T \mid F$
- $S :: var_1 := e_2 \mid \text{skip} \mid S_1; S_2 \mid \text{var} := \text{new} \mid \text{dispose} (\text{var})$
- $\mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid fi \mid \text{assume} (b)$

Let $\text{Exp}$ denote the set of expressions $e$ and $\text{Bool}$ denote the set of Boolean expressions $b$. We denote by $\text{LLPrograms}$ the set of loopless programs generated by the above syntax.

To define the semantics of pointer and Boolean expressions, we extend $f^M$ by $f^M(\text{err}) = \text{err}$ for every $f \in \text{fields}$. The functions $\mathcal{E}[e](\mathcal{M}) : \text{Exp} \rightarrow \text{Addresses}^M \cup \{\text{null}, \text{err}\}$ and $\mathcal{B}[b](\mathcal{M}) : \text{Bool} \rightarrow \{o_T, o_F, \text{err}\}$ (with $\text{err} \notin M$) are defined as follows:

- $\mathcal{E}[\text{var}](\mathcal{M}) = o_{\text{var}}, \text{ if } o^M_{\text{var}} \in \text{Alloc}^M$
- $\mathcal{E}[\text{var}](\mathcal{M}) = \text{err}, \text{ if } o^M_{\text{var}} \notin \text{Alloc}^M$
- $\mathcal{E}[\text{var}.f](\mathcal{M}) = f^M(\mathcal{E}[\text{var}](\mathcal{M}))$
- $\mathcal{B}[e_1 = e_2](\mathcal{M}) = \text{err} \text{ if } \mathcal{E}[e_i](\mathcal{M}) = \text{err}, i \in \{1, 2\}$
- $\mathcal{B}[e_1 = e_2](\mathcal{M}) = o_T, \text{ if } \mathcal{E}[e_1](\mathcal{M}) = \mathcal{E}[e_2](\mathcal{M})$
- $\mathcal{B}[e_1 = e_2](\mathcal{M}) = o_F, \text{ if } \mathcal{E}[e_1](\mathcal{M}) \neq \mathcal{E}[e_2](\mathcal{M})$

$\mathcal{B}$ extends naturally w.r.t. the Boolean connectives.

The operational semantics of the programming language is: For any command $S$, if $\mathcal{E}$ or $\mathcal{B}$ give the value err, then $(S, \mathcal{M}) \sim \text{abort}$. Otherwise, the semantics is as listed above. If $\mathcal{M}$ is a memory structure and $(S, \mathcal{M}) \sim \mathcal{M}'$, then $\mathcal{M}'$ is a memory structure.

1. $(\text{skip}, \mathcal{M}) \sim \mathcal{M}$.
2. $(\text{var}_1 := e_2, \mathcal{M}) \sim [\mathcal{M} | o^M_{\text{var}_1} \text{ is set to } \mathcal{E}[e_2](\mathcal{M})].$
3. \( \langle \text{var} := \text{new}, \mathcal{M} \rangle \sim \)
   \[ \mathcal{M} | \text{For some } t \in \text{MemPool}^\mathcal{M}, t \text{ is added to } \text{Alloc}^\mathcal{M} \text{ and } \sigma_{\text{var}}^\mathcal{M} \text{ is set to } t \],
4. If \( \sigma_{\text{var}}^\mathcal{M} \notin \text{Alloc}^\mathcal{M}, \langle \text{dispose}(\text{var}), \mathcal{M} \rangle \sim [\mathcal{M} | \sigma_{\text{var}}^\mathcal{M} \text{ is removed from } \text{Alloc}^\mathcal{M}].
5. \( \langle S_1; S_2, \mathcal{M} \rangle \sim \langle S_2, \langle S_1, \mathcal{M} \rangle \rangle \)
6. \( \langle \text{if } b \text{ then } S_T \text{ else } S_F, \mathcal{M} \rangle \sim \langle S_{TV}, \mathcal{M} \rangle \) where \( tV = B[b](\mathcal{M}) \).
7. \( \langle \text{if } b \text{ then } S \text{ else skip } fi, \mathcal{M} \rangle \sim \langle \text{assume}(\text{false}), \mathcal{M} \rangle \)
   otherwise \( \langle \text{assume}(\text{false}), \mathcal{M} \rangle \sim \text{abort} \).

Programs with Loops Our programs are represented as hybrids of the programming language for loopless code and control flow graphs.

**Definition 4 (Program).** A program is \( G = (V, E, \ell_{\text{init}}, \text{shp}, \text{cnt}, \lambda) \) such that \( G = (V, E) \) is a directed graph with no multiple edge but possibly containing self-loops, \( \ell_{\text{init}} \in V \) has in-degree 0, \text{shp and cnt are functions} \( \text{shp}, \text{cnt} : V \rightarrow \mathcal{L}(\tau) \) \text{ and } \( \lambda : E \rightarrow \mathcal{L}(\ell) \).

Here is the code \( S_2 \) from the introduction:

\[
\begin{align*}
V &= \{ \ell_b, \ell_e, \ell_c \} \\
E &= \{ (\ell_b, \ell), (\ell, \ell_c), (\ell_c, \ell_b) \} \\
\ell_{\text{init}} &= \ell_b \\
\lambda(\ell_b, \ell) &= \text{S}_b \\
\lambda(\ell, \ell_c) &= \text{assume}(\lambda(e = \text{null})); \text{S}_c \\
\lambda(\ell_c, \ell_b) &= \text{assume}(\text{false}); \text{S}_e
\end{align*}
\]

\( S_b, S_c \) and \( S_e \) denote the three loopless code blocks which are respectively the code block before the loop, inside the loop and after the loop. The annotations \( \text{shp} \) and \( \text{cnt} \) are described in Section 3.4.

The semantics of programs derive from the semantics of loopless programs and is given in terms of program paths. Given a program \( G \), a path in \( G \) is a finite sequence of directed edges \( e_1, \ldots, e_t \) such for all \( 1 \leq i \leq t-1 \), the tail of \( e_i \) is the head of \( e_{i+1} \). A path may contain cycles.

We extend the \( \sim \) relation to paths in programs:

**Definition 5 \( \sim^* \) for paths.** Given a program \( G \), a path \( P \) in \( G \), and memory structures \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) we define whether \( \langle P, \mathcal{M}_1 \rangle \sim^* \mathcal{M}_2 \) holds inductively.

- If \( P \) is empty, then \( \langle P, \mathcal{M}_1 \rangle \sim^* \mathcal{M}_2 \iff \mathcal{M}_1 = \mathcal{M}_2 \).
- If \( e_t \) is the last edge of \( P \), then \( \langle P, \mathcal{M}_1 \rangle \sim^* \mathcal{M}_2 \iff \text{there exists } \mathcal{M}_3 \text{ such that } \langle P \setminus \{ e_t \}, \mathcal{M}_1 \rangle \sim^* \mathcal{M}_3 \text{ and } \langle \lambda(e_t), \mathcal{M}_1 \rangle \sim^* \mathcal{M}_3 \). \( P \setminus \{ e_t \} \) denotes the path obtained from \( P \) by removing the last edge \( e_t \).

### 3.2 Methodology Revisited

Now, we have all definitions available to revisit the methodology formulated in the introduction. Our methodology assumes a program \( P \) as stated in Definition 4 as input (ignoring the \( \text{shp} \) and \( \text{cnt} \) functions for the moment).

**I. Shape Analysis.** The user annotates the program locations with shape predicates. The validity of the shape predicates is then proved by some suitable technique. For concreteness, we have stated the separation logic \( \text{SL} \)s as a possible
language for shape annotations in Section 2.4. The validity of SL\textsubscript{ls} annotations can, for example, be proven by techniques from \cite{8}.

II. Content Analysis. The user translates the shape invariants into \( L \)-formulae (stored in the \textsf{shp} function of \( P \)). For concreteness, we have defined an automatic translation from SL\textsubscript{ls} annotations to \( L \)-formulae in Section 2.5. However, we want to point out that our methodology is independent of the concrete choice of the shape predicates. We only require a translation of shape invariants to \( L \)-formulae, as discussed in Section 2.5. Now, the user annotates the program locations with \( L \)-formulae that she wants to verify (stored in the \textsf{cnt} function of \( P \)). We note that the \textit{cnt} annotations can use the concepts obtained from the translated shape invariants.

In the rest of the paper we describe how to verify the \textit{cnt} annotations. In Section 3.3 we state verification conditions and prove that the validity of the verification conditions implies the correctness of the \textit{cnt} annotations. The key point of our methodology is that the verification conditions can be discharged automatically by a satisfiability solver for \( L \)-formulae. The verification conditions rely on the backwards propagation function \( \Theta \) for \( L \)-formulae which we introduce in Section 3.5.

3.3 Content Verification

We want to prove that, for every initial memory structure \( M_1 \) from which the computation satisfies \textit{shp} and which satisfies the content pre-condition \( \textit{cnt}(\ell_{\text{init}}) \), the computation satisfies \textit{cnt}. Here are the corresponding verification conditions, which annotate the vertices of \( G \):

**Definition 6 (Verification conditions).** Given a program \( G \), \( VC \) is the function from \( E \) to \( L \) given for \( e = (\ell_0, \ell) \) by

\[
VC(e) = \neg (\textsf{shp}(\ell_0) \land \textsf{cnt}(\ell_0) \land \Theta_{\lambda(e)}(\textsf{shp}(\ell) \land \neg \textsf{cnt}(\ell)))
\]

\( VC(e) \) holds if \( \top \models VC(e) \), i.e. if \( VC(e) \) is a tautology over memory structures.

We make the notion of a computation satisfying the verification conditions precise using the following definition:

**Definition 7 (Reach(\( \ell \))).** Given a program \( G \), a node \( \ell \in V \), and a set \textit{Init} of memory structures, \textit{Reach(\( \ell \))} is the set of memory structures \( \mathcal{M} \) for which there exist \( \mathcal{M}_{\text{init}} \in \textit{Init} \) and a path \( P \) in \( G \) starting at \( \ell_{\text{init}} \) such that \( (P, \mathcal{M}_{\text{init}}) \leadsto^* \mathcal{M} \).

In particular, \( \textit{Reach}(\ell_{\text{init}}) = \textit{Init} \). Let \( J \) be a set of memory structures. We write \( J \models \varphi \) if, for every \( \mathcal{M} \in J \), \( \mathcal{M} \models \varphi \).

**Theorem 2 (Soundness of the methodology).** Let \( G \) be a program such that, for all \( \ell \in V \), \( \textit{Reach}(\ell) \models \textit{shp}(\ell) \). If \( \textit{Reach}(\ell_{\text{init}}) \models \textit{cnt}(\ell_{\text{init}}) \) and for all \( e \in E \), \( VC(e) \) holds, then \( \textit{Reach}(\ell) \models \textit{cnt}(\ell) \), for all \( \ell \in V \).
The proof of Theorem 2 is given in the appendix.

In the definition of a program, we defined the range of \( shp \) and \( cnt \) as \( \mathcal{L}(\tau) \). However, in order to allow invariants of the form

\[
\phi_{\text{lists-updt}} = \text{EmplList}_{gho} \equiv \text{EmplList} \wedge \text{ProjList}_{gho} \cup o_{proj} \equiv \text{ProjList}
\]

we add ghost concepts, roles and constants. We extended the vocabulary from \( \tau \) to \( \tau_{\text{ext}} \) by adding, for every symbol e.g. \( R \in \tau \), the symbol \( R_{gho} \). We think of \( \tau_{\text{ext}} \) structures as containing two snapshots of the memory: one is the current snapshot, and the other is a ghost snapshot, which is a snapshot of the memory at the beginning of the program. We denote the two underlying memory structures of \( M \) by \( M_{\text{cur}} \) and \( M_{\text{gho}} \). By \( \langle P, M_1 \rangle \sim^* M_2 \) we mean \( \langle P, (M_1)_{\text{cur}} \rangle \sim^* (M_2)_{\text{cur}} \).

In the sequel we refer to ghost elements of structures explicitly only when the treatment of them is different than other elements.

### 3.4 Running Example: General Methodology

The shape annotations of \( \ell_b, \ell \) and \( \ell_e \) in the program \( S_2 \) all describe the two lists \( \text{ProjList} \) and \( \text{EmplList} \) using \( \alpha_{ls,m} \) from Section 2.5.

The loop invariant also states that from \( \text{emplHead} \) to \( e \) we have a sub-list \( VList \) of \( \text{EmplList} \), where \( VList \) is a new concept. We think of \( VList \) as the subset of \( \text{EmplList} \) which has been visited so far in the loop. (\( VList \) was \( P_1 \) in Eq. (1)). At the start of every iteration, \( VList \) is a list starting at \( \text{emplHead} \) and ending at \( e \) (not inclusive). The variable \( \text{proj} \) points at the start of \( S_2 \) to an isolated cell not belonging to any of the lists. At the loop invariant and at the end of \( S_2 \), \( \text{proj} \) belongs to \( \text{ProjList} \).

\[
\begin{align*}
\varphi_{\text{two-lists}} &= \alpha_{ls,1}(\text{EmplList}, \text{null}, \text{next}, P_2) \wedge \\
& \quad \alpha_{ls,1}(\text{ProjList}, \text{null}, \text{next}, \text{EmplList}) \\
shp(\ell_b) &= \varphi_{\text{two-lists}} \wedge proj \sqsubseteq \text{Alloc} \sqcap \neg(\text{EmplList} \sqcup \text{ProjList}) \\
shp(\ell_c) &= \varphi_{\text{two-lists}} \wedge proj \sqsubseteq \text{ProjList} \\
shp(\ell_e) &= \alpha_{m}^{\varphi_{shp-\ell}}
\end{align*}
\]

where \( \varphi_{shp-\ell} \) is from Section 2.5.

The three content annotations require that the invariants of the system \( \varphi_{\text{invariants}} \) from Section 2.4 hold. The post-condition additionally requires that \( \varphi_{p-assgn} \) and \( \varphi_{\text{lists-updt}} \) hold. Recall \( \varphi_{p-assgn} \) states that every employee which was not assigned a project, is assigned to \( o_{proj} \). \( \varphi_{\text{lists-updt}} \) states that the content of the two lists remain unchanged, except that the project \( o_{proj} \) is inserted to the project list.

\[
\begin{align*}
cnt(\ell_b) &= \varphi_{\text{invariants}} \\
cnt(\ell_c) &= \varphi_{\text{invariants}} \wedge \varphi_{\text{lists-updt}} \wedge \varphi_{p-assgn} \\
cnt(\ell_e) &= \varphi_{\text{invariants}} \wedge \varphi_{\text{lists-updt}} \wedge \varphi_{p-assgn-\ell} \\
\varphi_{p-assgn-\ell} &= VList \sqcap \exists \text{worksFor}_{gho} \cdot o_{null} \equiv VList \sqcap \exists \text{worksFor}_{o_{proj}}
\end{align*}
\]
\( \varphi_{p\text{-assign}} \) states that, in the list \( VList \) containing the employees visited so far in the loop, any employee which was not assigned to a project at the start of the program (i.e., in the ghost version of \textit{worksFor}) is assigned to the project \( \text{proj} \). \( \varphi_{VList} \) makes no demands on elements of \( \text{EmplList} \) which have not been in the loop so far.

The verification conditions of \( G \) are

\[
VC(\ell_b, \ell) = \neg (\text{shp}(\ell_b) \land \text{cnt}(\ell_b) \land \Theta_{\lambda(\ell_b, \ell)}(\text{shp}(\ell) \land \neg \text{cnt}(\ell)))
\]

\[
VC(\ell, \ell_p) = \neg (\text{shp}(\ell) \land \text{cnt}(\ell) \land \Theta_{\lambda(\ell, \ell_p)}(\text{shp}(\ell_p) \land \neg \text{cnt}(\ell_p)))
\]

\[
VC(\ell, \ell_c) = \neg (\text{shp}(\ell) \land \text{cnt}(\ell) \land \Theta_{\lambda(\ell, \ell_c)}(\text{shp}(\ell_c) \land \neg \text{cnt}(\ell_c)))
\]

To apply Theorem 2 to \( G \), we only need to prove that the verification conditions \( VC(e) \) hold. Using Lemma 3, \( VC(\ell_b, \ell) \) holds if and only for every memory structures \( \mathcal{M}, \mathcal{M}' \) which satisfies the shape and content invariants \( \text{shp}(\ell_b) \) and \( \text{cnt}(\ell_b) \) that the computation of \( S_b \) on \( \mathcal{M} \) does not abort, \( \langle S_b, \mathcal{M} \rangle \sim \mathcal{M}' \) at \( \mathcal{M}' \) satisfies \( \text{shp}(\ell) \), then \( \mathcal{M}' \) satisfies \( \text{cnt}(\ell) \). The case is similar for the two other verification conditions. Since in this case the verification conditions hold, the program satisfies the annotation \( \text{cnt} \).

**Conclusion 1** Reach(\( \ell \)) \( \models \text{cnt}(\ell) \), for all \( \ell \in V \).

### 3.5 Backwards Propagation

In this section we present the backwards propagation of a formula along a loopless program \( S \). Let \( \langle S, \mathcal{M}_1 \rangle \sim \mathcal{M}_2 \). Fields and variables (and other nominals) in \( \mathcal{M}_2 \) are translated into expressions involving elements of \( \mathcal{M}_1 \). For the remaining relations (e.g. \( \text{ProjList} \) and \( \text{EmplList} \) in our example) the treatment is different, in order to permit them to be deduced from a shape analysis in Section 3.3. These relations are copied as they are from \( \mathcal{M}_2 \) and added to \( \mathcal{M}_1 \). For every such relation \( R \), we add a relation symbol \( R^{\text{end}} \) for the copied relation.

We denote by \( \langle \mathcal{M}_1, R^{\text{end}} \rangle \) the structure obtained from \( \mathcal{M} \) by adding the relations \( \langle R^{\text{end}} \rangle_{\mathcal{M}_1} = R^{\mathcal{M}_2}, R \in \tau \setminus \text{fields} \). The vocabulary \( \tau^{\text{end}} \) of \( \langle \mathcal{M}, R^{\mathcal{M}_2} \rangle \) extends \( \tau^{\text{end}} \) by relation symbols \( R^{\text{end}} \) for every relation \( R \in \tau \setminus \text{fields} \).

For every formula \( \varphi \) of \( \tau \), we first substitute in \( \varphi \) the relations \( R \) which are not fields with \( R^{\text{end}} \). Then we perform the backwards propagation, which updates the fields and variables according to the loopless code.

We need a further extension of our structures. Given a finite set \( Y \) of labels, memory structures \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) with universe \( M \), and a tuple \( d_Y = (d_y : y \in Y) \) of elements of \( M \), we denote by \( \langle \mathcal{M}_1, R^{\text{end}}, d_Y \rangle \) the structure obtained from \( \langle \mathcal{M}, R^{\text{end}} \rangle \) by adding the constants \( d_y, y \in Y \). The vocabulary \( \tau_Y^{\text{end}} \) of \( \langle \mathcal{M}, R^{\text{end}}, d_Y \rangle \) extends \( \tau^{\text{end}} \) by constant symbols \( \{a_y : y \in Y\} \).

Given a loopless program \( S \), we assign unique labels \( y \) to the commands of \( S \). We denote by \( Y_S \) the set of labels of commands in \( S \).

**Lemma 3.** Let \( S \) be a loopless program, \( Y_S \) be the set of labels of commands in \( S \), \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) be memory structures, and \( \varphi \) be an \( \mathcal{L} \)-formula over \( \tau \). There exists a program \( \overline{S} \) and a new label \( \overline{y} \notin Y_S \) such that:
The definition of $\Theta_S$ is:

**Definition 8 ($\Theta_S(\varphi)$).** $\Theta_S(\varphi)$ is obtained from $\Psi$ by substituting every relation symbol $R \in \tau_{\text{fields}}$ in $\varphi$ by $R^{\text{end}}$.

\[
\Psi_{\text{skip}}(\varphi) = \varphi
\]

\[
\Psi_{\text{var}:=e}(\varphi) = \varphi[o_{\text{var}}/o_e], \quad e = \text{var}_2 \text{ or } e = \text{null}
\]

\[
\Psi_{\text{if } b \text{ then } S_1 \text{ else } S_2}(\varphi) = \varepsilon_b \land \Psi_{S_1}(\varphi) \lor \neg \varepsilon_b \land \Psi_{S_2}(\varphi)
\]

\[
\Psi_{\text{dispose}(\text{var})}(\varphi) = \Psi_{\text{var}:=\text{null}}[\text{new}] \land \Psi_{S_1}(\varphi)
\]

\[
\Psi_{\text{dispose}(\text{var})}(\varphi) = \Psi_{\text{var}:=\text{null}}[\text{new}] \land \Psi_{S_2}(\varphi)
\]

The notation $\varphi[A/B]$ should be interpreted as the syntactic replacement of any occurrence of $A$ with $B$. We write e.g. $y: \text{var} := \text{new}$ to indicate that the command $\text{var} := \text{new}$ is executed. For $\varepsilon_1 = \varepsilon_2$ we set $\varepsilon_b = (A_{e_1} = A_{e_2})$, with $A_{\text{var}} = o_{\text{var}}$ and $A_{\text{var},f} = \exists f_{-}\text{var}$.\varepsilon_b extends naturally to the Boolean connectives. In the definition of $\Psi_{\text{dispose}(\text{var})}$, $f_{k_1}, \ldots, f_{k_w}$ are the members of $\tau_{\text{fields}}$ which occur in $\varphi$.

W.l.o.g. we assume that $S$ does not contain commands of the form $\text{if } b \text{ then } S_1 \text{ fi}$ or $\text{var}_1.f_1 := \text{var}_2.f_2$, since they can be expressed using the other commands.

We obtained $\overline{S}$ from $S$ as follows. Let $abo$ be a new variable, whose corresponding constant symbol is $o_{abo}$. We will use $abo$ to indicate whether the run of $M_1$ on $S$ has aborted. The program $\overline{S}$ does not abort. The command $abo := F$ is added at the beginning of the code. Every command $C$ of the form $\text{var}_1 := \text{var}_2.f$, $\text{var}_1.f := \text{var}_2$ or $\text{dispose}(\text{var}_2)$ is replaced with $\overline{C} = \text{if } \text{var}_2 = \text{null then } abo := T \text{ else } C \text{ fi}$. For $\text{if }$ and $\text{assume}$ commands and for general assignment $\text{var}_1.f_1 := \text{var}_2.f_2$, the case is similar, except that there may be two evaluations of the form $\text{var}_1.f_j$, which need to be reflected in the condition in $\overline{C}$.

By the construction of $\overline{S}$, the run of $M_1$ on $\overline{S}$ does not abort. Moreover, $abo$ has the value $T$ at the end of the run of $\overline{S}$ on $M_1$ if and only if $S$ aborts on $M_1$.

Lemma 3 has to undergo a small correction in order to deal with ghost concepts, roles and constants, due to the fact $(M_1)_{\text{gho}} = (M_2)_{\text{gho}}$. $\Theta$ is augmented so that every occurrence of $s_{\text{gho}}^{\text{end}}$ in $\Theta_S(\varphi)$ is substituted with $s_{\text{gho}}$, for every symbol $s$. 

1. Let $\langle S, M_1 \rangle \sim M_2$. Then: $M_2 \models \varphi$ iff there exists a tuple $\bar{a}_{\varphi \cup \{abo\}}$ of $M$ elements such that $\langle M_1, \bar{R}^{\text{end}}, \bar{a}_{\varphi \cup \{abo\}} \rangle \models \Theta_S(\varphi \land (o_{abo} \equiv o_\text{F})).$

2. If $\langle S, M_1 \rangle \sim \text{abort}$, then for every tuple $\bar{a}_{\varphi \cup \{abo\}}$ of $M$ elements, $\langle M_1, \bar{R}^{\text{end}}, \bar{a}_{\varphi \cup \{abo\}} \rangle \not\models \Theta_S(\varphi \land (o_{abo} \equiv o_\text{F})).$
3.6 Running Example: Backwards Propagation

As an example of the backwards propagation process, we consider a formula from Section 3.4, which is part of the content annotation of $\ell$:

$$\varphi_{p\text{-}assgn\text{-}\ell} = VList \cap \exists \text{worksFor}_{\text{gho\text{-}o\text{null}}} \equiv VList \cap \exists \text{worksFor}_{\text{o\text{proj}}}$$

The result of the backwards propagation on $\varphi_{p\text{-}assgn\text{-}\ell}$ is:

$$\Psi_{\ell}(\varphi_{p\text{-}assgn\text{-}\ell}) = \exists \text{worksFor}_{\text{o\text{null}}} \equiv o\text{null} \land \Psi_{\text{e.worksFor}}(\varphi_{p\text{-}assgn\text{-}\ell}) \lor \neg (\exists \text{worksFor}_{\text{o\text{null}}} \equiv o\text{null}) \land \varphi_{p\text{-}assgn\text{-}\ell}$$

$$\Psi_{\text{e.worksFor}\llbracket o\text{proj} \rrbracket}(\varphi_{p\text{-}assgn\text{-}\ell}) = VList \cap \exists \text{worksFor}_{\text{gho\text{-}o\text{null}}} \equiv VList \cap \exists ((\text{worksFor} \setminus (o\text{null} \times T)) \cup (o_e, o\text{proj})).o\text{proj}$$

$\Psi_{\text{e.worksFor}\llbracket o\text{proj} \rrbracket}(\varphi_{p\text{-}assgn\text{-}\ell})$ is obtained from $\varphi_{p\text{-}assgn\text{-}\ell}$ by substituting the $\text{worksFor}$ role with the correction $((\text{worksFor} \setminus (o\text{null} \times T)) \cup (o_e, o\text{proj}))$ which updates the value of $o_e$ in $\text{worksFor}$ to $\text{proj}$. Note $\varphi_{\text{e}\llbracket next \rrbracket}(\varphi_{p\text{-}assgn\text{-}\ell}) = \varphi_{p\text{-}assgn\text{-}\ell}$, because $\text{next}$ does not occur in $\varphi_{p\text{-}assgn\text{-}\ell}$. $\varphi_{\text{ls}\llbracket 1 \rrbracket}(\text{empHead}, e, \text{next}, VList)$, stating that $VList$ is a list with pointer $\text{next}$.

$$\Theta_{\ell}\{\varphi_{p\text{-}assgn\text{-}\ell}\}$$ is obtained from $\Psi_{\ell}(\varphi_{p\text{-}assgn\text{-}\ell})$ by substituting $VList$ with $VList_{\text{end}}$. $VList$ and $VList_{\text{end}}$ stand for the list at the start respectively end of the iteration. While $\Theta_{\ell}\{\varphi_{p\text{-}assgn\text{-}\ell}\}$ does not contain $VList$, it is intended to be a part of a larger formula, which does refer to $VList$.

4 Related Work

**Shape Analysis.** Shape analysis has attracted considerable attention in the literature. The classical introductory paper to separation logic [25] presents an expressive separation logic which has turned out to be undecidable. We have restricted our attention to the restricted and better behaved fragment in [8], see also [7]. The work on separation logic focuses mostly on shape rather than content in our sense. Separation logic has been extended to object oriented languages, cf. e.g. [23] and [11], where shape properties similar to those studied in the non object oriented case are the focus, and the main goal is to overcome difficulties introduced by the additional features of OO languages.

Other shape analyses could be potential candidates for integration in our methodology. [26] and [28] use 3-valued logic to perform shape analysis. Regional logic is used to check correctness of program with shared dynamically allocated memory areas [4]. [15] uses nested tree automata to represent the heap. [19] combines monadic second order logic with SMT solvers.

**Description Logic.** DLs have not been considered for verification of programs with dynamically allocated memory, with the exception of [12] whose use (mostly undecidable) DLs to express shape-type invariants, ignoring content information. In [10] the authors consider verification of loopless code (transactions) in graph databases with integrity constraints expressed in DLs. Verification of temporal properties of dynamic systems in the presence of DL knowledge bases...
has received significant attention (see \cite{4,3,13} and the references therein). *Temporal Description Logics*, which combine classic DLs with classic temporal logics, have also received significant attention in the last decade (see \cite{18} for a survey).

**Related Ideas.** Several recent research projects have concentrated on verification strategies which use information beyond the semantics of the source code. For instance, \cite{17} is using diagrams from design documentation to support verification. Another recent paper \cite{29} is inferring the intended use of program variables to guide a program analysis. Instead of starting from code and verifying its correctness, \cite{14} explores how to declaratively specify data structures with sharing and how to automatically generate code from this specification.

Due to the interdisciplinary nature of this work and space limitations, it is difficult to cover all the relevant literature. We welcome all suggestions.

5 Conclusion

In this paper we have demonstrated that description logic can be used as the basis of a novel program analysis methodology. Our method analyzes the data content of dynamic data structures on top of existing shape analysis methods. A central feature of our approach is the systematic translation of shape invariants into description logic. Our future work will focus on the identification of new description logics which are particularly well-suited for this application, enable us to obtain more precise translations of shapes, and have good algorithmic properties. While the current approach is semi-manual, our long term goal is to increase the automatization of the method.

**References**

Note that any finite model $M$ of $\psi$ is almost the desired memory structure with $M \models \psi$. The desired $M$ is obtained by adding to $MemPool^M$ infinitely many fresh elements $e$ and setting $f^M(e) = o_{null}^M$ for all $f \in \tau_{fields}$. 

### A Satisfiability in Memory Structures

Proof (Theorem [7]). For the upper bound, we employ the fact that finite satisfiability of $L$-formulae, i.e. truth in a structure with a finite domain, is in NExpTime. This can be shown by a fairly standard reduction (see e.g. [9]) to finite satisfiability of formulae in $C^2$, the two-variable fragment of the first-order logic with counting quantifiers. The latter problem is in NExpTime due to [24]. We give a finite satisfiability preserving translation from $L$ to $C^2$ in Table 1.

To show that satisfiability of a formula $\psi$ in a memory structure can be decided in nondeterministic exponential time, it suffices to construct in linear time a formula $\psi_m$ such that $\psi$ is satisfiable in a memory structure if and only if $\psi_m \wedge \psi$ is finitely satisfiable. The formula $\psi_m$ is the conjunction of the following formulae corresponding to requirements (3)-(9) we placed on memory structures:

- $Aux \equiv o_{null} \sqcup o_T \sqcup o_F$.
- $Addresses \subseteq \neg Aux$ and $Addresses \cup Aux \equiv T$.
- $Alloc \subseteq \neg MemPool$ and $Alloc \cup MemPool \subseteq Addresses$.
- $o \subseteq \neg MemPool$ for every nominal concept $o$ of $\psi$.
- $func(f)$ and $Addresses \subseteq \exists f. \neg MemPool$ for every $f$ of $\psi$ with $f \in \tau_{fields}$.
- $MemPool \subseteq \exists f. o_{null} \sqcup o_F$ for every $f \in \tau_{fields}$, and
- $C \subseteq \neg MemPool$ for every atomic concept $C$ in $\tau$ with $C \neq MemPool$.

Note that any finite model $M$ of $\psi_m \wedge \psi$ is almost the desired memory structure with $M \models \psi$. The desired $M$ is obtained by adding to $MemPool^M$ infinitely many fresh elements $e$ and setting $f^M(e) = o_{null}^M$ for all $f \in \tau_{fields}$.

Table 1. Translation of $L$ into $C^2$ by employing only two variables $x$ and $y$. In each translation rule, $z \in \{x, y\}$. Moreover, $z = y$ if $z = x$, and $z = x$ if $z = y$. 

\[
\begin{align*}
tr_x(C) &= C(z) & C & \text{is an atomic concept} \\
tr_x(r) &= r(z, z) & r & \text{is an atomic role} \\
tr_x(C \land D) &= tr_x(C) \land tr_x(D) \\
tr_x(C \lor D) &= tr_x(C) \lor tr_x(D) \\
tr_x(\neg C) &= \neg tr_x(C) \\
tr_x(r \sqcap s) &= tr_x(r) \land tr_x(s) \\
tr_x(r \sqcup s) &= tr_x(r) \lor tr_x(s) \\
tr_x(r \setminus s) &= tr_x(r) \land \neg tr_x(s) \\
tr_x(r^{-}) &= tr_x(r) \\
tr_x(C \times D) &= tr_x(C) \land tr_x(D) \\
tr_x(\exists x.C) &= \exists y.tr_x(z) \land tr_x(C) \\
tr_x(C \subseteq D) &= \forall x.tr_x(C) \rightarrow tr_x(D) \\
tr_x(r \subseteq s) &= \forall x.y.tr_x,y(r) \rightarrow tr_x,y(s) \\
tr_x(\varphi \land \psi) &= tr_x(\varphi) \land tr_x(\psi) \\
tr_x(func(r)) &= \exists x.\exists y.tr_x,y(r) \\
tr_x(\neg \varphi) &= \neg tr_x(\varphi)
\end{align*}
\]
The lower bound comes from the NExpTime-hardness of satisfiability in the DL ACHOTIQ [27]. We note that the proof in [27] concerns satisfiability in arbitrary structures, including structures that are not memory structures. However, the reduction from the $2^n \times 2^n$ tiling problem given in [27] can be easily recast in terms of memory structures.

\section*{B Backwards propagation: Proof of Lemma 3}

To prove Lemma 3 we need the following lemma:

**Lemma 4.** Let $S$ be a loopless program without assume commands, $Y_S$ be the set of labels of commands in $S$, $Y$ be a set of labels disjoint from $Y_S$, $\mathcal{M}_1$ and $\mathcal{M}_2$ be memory structures with universe $M$ such that $\langle S, \mathcal{M}_1 \rangle \sim \mathcal{M}_2$. $\bar{d}_Y$ be a tuple of $M$ elements and $\varphi$ be an L-formula over $\tau \cup \{o_y : y \in Y\}$. $\langle \mathcal{M}_2, \bar{d}_Y \rangle \models \varphi$ iff there exists a tuple $\bar{d}_{Y_S}$ of $M$ elements such that $\langle \mathcal{M}_1, R^{end}, \bar{d}_Y, \bar{d}_{Y_S} \rangle \models \Theta_S(\varphi)$.

**Proof.** We prove the lemma by induction.

- $S = \text{skip}: Y_S = \emptyset$, and we have $\langle \mathcal{M}_2, \bar{d}_Y \rangle \models \varphi$ iff $\langle \mathcal{M}_1, R^{end}, \bar{d}_Y \rangle \models \Theta_S(\varphi)$, as required. ($d_{Y_S}$ is the empty tuple.)
- $S = \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$: depending on whether $\varepsilon_b$ is true or false, $\Theta_{S_1}(\varphi)$ or $\Theta_{S_2}(\varphi)$ should be used.
- $\text{var} := e$, where $e$ is a variable $\text{var} 2$ or null: every reference to $\text{var}$ in $\varphi$ is replaced by a reference to $\text{var} 2$ or null, respectively.
- $\text{var} := \text{var} 2, f$: every reference to $\text{var}$ in $\varphi$ is replaced with a reference to $d_y$, the interpretation of $o_y$, and $d_y$ is required to be the result of applying $f$ on $\text{var} 2$.
- $\text{var} 1 := e$, where $e$ is a variable $\text{var} 2$ or null: the function symbol $f$ is updated by removing the current value of $f$ on $\text{var} 1$ by subtracting $(o_{\text{var} 1} \times \top)^{\mathcal{M}_1}$ from $f^{\mathcal{M}_1}$ and setting the new value explicitly by adding the pair $(o_{\text{var} 1}, o_{\text{var} 1}^{\mathcal{M}_1})$ to $f^{\mathcal{M}_1}$.
- $S = \text{new} \text{ with label } y: Y_S = \{y\}$, and we set $\bar{d}_{Y_S} = (d_y)$ such that $\langle d_y \rangle = \text{Alloc}^{\mathcal{M}_2} \setminus \text{Alloc}^{\mathcal{M}_1}$. Such $d_y$ exists by the semantics of the allocation command. $\Theta_S(\varphi)$ adds $d_y$ to $\text{Alloc}^{\mathcal{M}_1}$ and replaces every reference to $\text{var}$ by a reference to $d_y$.
- $S = \text{dispose}(\text{var}): \Theta_S(\varphi)$ removes $\text{var}$ from $\text{Alloc}$, and using an application of $\Theta$ to the program $\text{var}.f_{k_1} := \text{null}; \cdots; \text{var}.f_{k_w} := \text{null}$, sets all of the fields in $\varphi$ to null.
- $S = S_1; S_2: \Theta_S(\varphi) = \Theta_{S_1}(\Theta_{S_2}(\varphi))$. Let $\mathcal{M}_3$ be an memory structure such that $\langle S_1, \mathcal{M}_1 \rangle \sim \mathcal{M}_3$ and $\langle S_2, \mathcal{M}_2 \rangle \sim \mathcal{M}_2$.

Consider first $\Theta_{S_1}(\varphi)$. By the induction hypothesis, there exist $\bar{d}_{Y_{S_2}}$ such that $\langle \mathcal{M}_2, \bar{d}_Y \rangle \models \varphi$ iff there exists a tuple $\bar{d}_{Y_{S_2}}$ of $M$ elements such that $\langle \mathcal{M}_3, R^{end}, \bar{d}_Y, \bar{d}_{Y_{S_2}} \rangle \models \Theta_{S_2}(\varphi)$.

Let $\mathcal{M}_4$ be obtained from $\mathcal{M}_3$ by replacing every relation $R^{\mathcal{M}_3}$ with $R^{\mathcal{M}_2}$ for $R \in \tau \setminus \tau_{\text{fields}}$. In other words, $R^{end} = R^{end}$. Let $\psi = \Psi_{S_1}(\varphi)$. $\Theta_{S_2}(\varphi)$ does not use any relation $R \in \tau \setminus \tau_{\text{fields}}$ (but rather the $R^{end}$), and therefore the...
vocabulary of $\psi$ is that of $\langle M_4, \bar{d}_Y, \bar{d}_Y s_2 \rangle$. Moreover, $\psi \models \langle M_4, \bar{d}_Y, \bar{d}_Y s_2 \rangle$ iff $\Theta_{S_2}(\varphi) \models \langle M_3, R^{end}, \bar{d}_Y, \bar{d}_Y s_2 \rangle$.

Applying the induction hypothesis once again, there exists $\bar{d}_Y s_2$ such that $\langle M_4, \bar{d}_Y, \bar{d}_Y s_2 \rangle \models \varphi$ if there exists a tuple $\bar{d}_Y s_1$ of $M$ elements such that $\langle M_1, \bar{d}_Y, \bar{d}_Y s_2, \bar{d}_Y s_1 \rangle \models \Theta_{S_1}(\Theta_{S_2}(\varphi))$. Denoting the concatenation of $\bar{d}_Y s_1$ and $\bar{d}_Y s_2$ by $\bar{d}_Y s$, we get $\langle M_2, \bar{d}_Y \rangle \models \varphi$ iff $\langle M_1, R^{end}, \bar{d}_Y, \bar{d}_Y s \rangle \models \Theta_{S_1}(\Theta_{S_2}(\varphi))$.

Remark 1. In the proof of Theorem 2 only case 1. of Lemma 3 is used. The purpose of case 2. of Lemma 3 is to make the verification conditions less strict, in the sense that they require nothing of aborted executions.

Proof (Lemma 3). Using Lemma 4 with $Y = \emptyset$, if $S$ aborts on $M_1$, then for every $\bar{d}_Y s$ and $d_{abo}$ and $\bar{d}$ we have $\langle M_1, \bar{d}_Y s, d_{abo} \rangle \not\models \psi_S(\varphi \land (o_{abo} \equiv o_F))$; if $\langle S, M_1 \rangle \sim M_2$, $M_2 \models \varphi$ iff $\langle M_1, \bar{d}_Y s, d_{abo} \rangle \models \psi_{S'}(\varphi \land (o_{abo} \equiv o_F))$. Note that $d_{abo}$ is the value of $abo$ at the beginning of the run of $S$. Since the first command of $S$ assigns a new value, $d_{abo}$ plays no role (it appears because, technically, $abo$ still needs a value at the beginning of the run).

### C Proof of Theorem 2

Let $M \in Reach(\ell)$, and let $P$ be a path and $M_{init} \in Init$ as guaranteed for members of $Reach(\ell)$. We prove the claim by induction on the length of $P$.

If $P$ is empty, the claim holds.

If $P$ is not empty, let $e = (\ell_0, \ell)$ be the last edge of $P$, and let $P_0$ be the path obtained from $P$ by removing $e$. Let $M_0$ be an memory structure such that $\langle P_0, M_{init} \rangle \sim^* M_0$, and $\langle e, M_0 \rangle \sim^* M$. We have $M_0 \models shp(\ell_0)$ and $M \models shp(\ell)$. By the induction hypothesis, $M_0 \models cnt(\ell_0)$.

Since $VC(e)$ holds, in particular we have for every $\bar{d}$, $\langle M_0, \bar{d} \rangle \models VC(e)$. Let $\bar{d} = R^{end}$. For every $\bar{d}$, $\langle M_0, \bar{d} \rangle \models shp(\ell_0) \land cnt(\ell_0)$. Therefore, for every $\bar{d}$, $\langle M_0, \bar{d} \rangle \not\models \Theta_{\lambda(e)}(shp(\ell) \land \neg cnt(\ell))$. By Lemma 3, $M \not\models shp(\ell) \land \neg cnt(\ell)$. Since $M \models shp(\ell)$, we get $M \models cnt(\ell)$. 