Shape and Content[⋆]
A Database-Theoretic Perspective on the Analysis of Data Structures

Diego Calvanese[1], Tomer Kotek[2], Mantas Šimkus[2], Helmut Veith[2], and Florian Zuleger[2]

[1] Free University of Bozen-Bolzano
[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 look at the verification of programs with dynamic data structures from the perspective of content representation. 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. 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. We show that the two-variable fragment of first order logic with counting and trees can be used as a joint framework to embed suitable fragments of description logic and separation logic.

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 data structures, however, is notoriously difficult, and is a highly active area of current research. While much progress has been made recently in analyzing and verifying the *shape* of dynamic data structures, most notably by separation logic (SL) [24,17], the *content* of dynamic data structures has not received the same attention.

In contrast, disciplines as databases, modeling and knowledge representation have developed highly-successful theories for *content representation and verification*. These research communities typically model reality by classes and binary

⋆ Kotek, Veith and Zuleger were supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED and ICT12-059. Simkus was supported by the FWF grant P25518 and the WWTF grant ICT12-15.

relationships between these classes. For example, the database community uses *entity-relationship (ER)* diagrams, and *UML* diagrams have been studied in requirements engineering. Content representation in the form of UML and ER has become a central pillar of *industrial software engineering.* In complex software projects, the source code is usually accompanied by *design documents* which provide extensive documentation and models of data structure content. This documentation is both an opportunity and a challenge for program verification. Recent hardware verification papers have demonstrated how design diagrams can be integrated into an industrial verification workflow [18].

In this paper, we propose the use of *Description Logics* (DLs) for the formulation of content specifications. DLs are a well established and highly popular family of logics for representing knowledge in artificial intelligence [3]. In particular, DLs allow to precisely model and reason about UML and ER diagrams [6,2]. DLs are mature and well understood, they have good algorithmic properties and have efficient reasoners. DLs are very readable and form a natural base for developing specification languages. For example, they are the logical backbone of the Web Ontology Language (OWL) for the Semantic Web [22]. DLs vary in expressivity and complexity, and are usually selected according to the expressivity needed to formalize the given target domain.

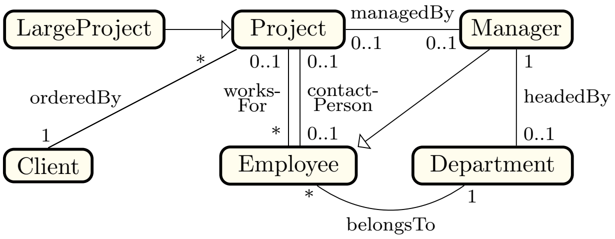
Unfortunately, the existing content representation technology cannot be applied directly for the verification of content specifications of pointer-manipulating programs. This is to due the strict separation between high-level content descriptions such as UML/ER and the way data is actually stored. For example, query languages such as SQL and Datalog provide a convenient abstraction layer for formulating data queries while ignoring how the database is stored on the disk. In contrast, programs with dynamic data structures manipulate their data structures directly. Moreover, database schemes are usually static while a program may change the content of its data structures over time.

The main goal of this paper is to develop a verification methodology that allows to employ DLs for formulating and verifying content specifications of pointer-manipulating programs. We propose a two-step Hoare-style verification methodology: First, existing shape-analysis techniques are used to derive shape invariants. Second, the user strengthens the derived shape invariants with content annotations; the resulting verification conditions are then checked automatically. Technically, we employ a very expressive DL (henceforth called $\mathcal{L}$), based on the so called $\mathcal{ALCHOIF}$, which we specifically tailor to better support reasoning about complex pointer structures. For shape analysis we rely on the SL fragment from [7]. In order to reason automatically about the verification conditions involving DL as well as SL formulae, we identify a powerful decidable logic CT^2 which incorporates both logics [10]. We believe that our main contribution is conceptual, integrating these different formalisms for the first time. While the current approach is semi-manual, our long term goal is to increase the automatization of the method.

Overview and Contributions

- In Section 2, we introduce our formalism. In particular, we formally define *memory structures* for representing the heap and we study the DL $\mathcal{L}$ as a formalism for expressing *content properties* of memory structures.
- In Section 2, we further present the building blocks for our verification methodology: We give an embedding of $\mathcal{L}$ and an embedding of a fragment of the SL from [7] into CT^2 (Lemmata 2 and 3). Moreover, we give a complexity-preserving reduction of satisfiability of CT^2 over memory structures to finite satisfiability of CT^2 (Lemma 1).
- In Section 3, we describe a program model for sequential imperative heap-manipulating programs without procedures. Our main contribution is a Hoare-style proof system for verifying content properties on top of (already verified) shape properties stated in SL.
- Our main technical result is a precise backward-translation of content properties along loop-less code (Lemma 5). This backward-translation allows us to reduce the inductiveness of the Hoare-annotations to satisfiability in CT^2. Theorem 1 states the soundness and completeness of this reduction.

1.1 Running Example: Information System of a Company



Our running example will be a simple information system for a company with the following UML diagram:. The UML gives the relationships between entities in the information system, but says nothing regarding the implementations of the data structures that hold the data. We focus mostly on projects, and on the employees and managers which work on them. Here is an informal description of the programmers' intention. The employees and projects are stored in two lists, both using the *next* pointer. The heads of the two lists are pHd and eHd respectively. Here are some properties of our information system. (i)-(iii) extends the UML somewhat. (iv)-(vi) do not appear in the UML, but can be expressed in DL:

(i) Each employee in the list of employees has a pointer $wrkFor$ 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).
(ii) Each project in the list has a pointer $mngBy$ to the employee list, indicating the manager of the project (or to null, if the project doesn't have one).
(iii) Employees have a Boolean field $isMngr$ marking them as managers, and only they can manage projects.
(iv) The manager of a project works for the project.
(v) At least 10 employees work on each large project.
(vi) The contact person for a large-scale project is a manager.

We will refer to these properties as the *system invariants.*

The programmer has written a program S (stated below) for verification. The programmer has the following intuition about her program: The code S 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.

```
ℓ_b : proj := new;
      proj.next := pHd;
      pHd := proj;
      e := eHd;
ℓ_l : while ~(e = null) do
        if (e.wrkFor = null)
        then e.wrkFor := proj;
        e := e.next;
      od
ℓ_e : end;
```

The programmer wants to verify that the system invariants are true after the execution of S, 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, 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). We will formally prove the correctness of S following our verification methodology discussed in the introduction. In Section 2.3 we describe how our DL 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 S.

2 Logics for Invariant Specification

2.1 Memory Structures

We use ordinary first order structures to represent memory in a precise way. A *structure* (or, *interpretation*) is a tuple $\mathcal{M} = (M, \tau, \cdot)$, where (i) M is an infinite set (the *universe*), (ii) τ 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 $c^{\mathcal{M}} \in M$, and to each n-ary relation symbol $R \in \tau$ an n-ary relation $R^{\mathcal{M}}$ over M. Each relation is either unary or binary (i.e. $n \in \{1, 2\}$). Given $A \subseteq M$, a binary $R^{\mathcal{M}}$, $R^{\mathcal{M}}$ and $e \in A^{\mathcal{M}}$, we may use the notation $R^{\mathcal{M}}(e)$ if $R^{\mathcal{M}}$ is known to be a function over $A^{\mathcal{M}}$.

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

(1) τ includes the constants o_{null}, $o_{\mathbf{T}}$, $o_{\mathbf{F}}$.
(2) τ has the unary relations $\mathit{Addresses}$, Alloc, $\mathit{PossibleTargets}$, $\mathit{MemPool}$, and Aux.
(3) $\mathit{Aux}^{\mathcal{M}} = \{o_{\text{null}}^{\mathcal{M}}, o_{\mathbf{T}}^{\mathcal{M}}, o_{\mathbf{F}}^{\mathcal{M}}\}$ and $|\mathit{Aux}^{\mathcal{M}}| = 3$.
(4) $\mathit{Addresses}^{\mathcal{M}} \cap \mathit{Aux}^{\mathcal{M}} = \emptyset$ and $\mathit{Addresses}^{\mathcal{M}} \cup \mathit{Aux}^{\mathcal{M}} = M$.
(5) $\mathit{Alloc}^{\mathcal{M}}$, $\mathit{PossibleTargets}^{\mathcal{M}}$ and $\mathit{MemPool}^{\mathcal{M}}$ form a partition of $\mathit{Addresses}^{\mathcal{M}}$.

(6) $c^{\mathcal{M}} \in M \backslash MemPool^{\mathcal{M}}$ for every constant c of τ.
(7) For all $f \in \tau_{\text{fields}}$, $f^{\mathcal{M}}$ is a function from $Addresses^{\mathcal{M}}$ to $M \backslash MemPool^{\mathcal{M}}$.
(8) If $e \in MemPool^{\mathcal{M}}$, then $f^{\mathcal{M}}(e) \in \{o^{\mathcal{M}}_{\text{null}}, o^{\mathcal{M}}_{\mathbf{F}}\}$.
(9) $R^{\mathcal{M}} \subseteq (M \backslash MemPool^{\mathcal{M}})^n$ for every[1] n-ary $R \in \tau \setminus (\{MemPool\} \cup \tau_{\text{fields}})$.
(10) $Alloc^{\mathcal{M}}$ and $PossibleTargets^{\mathcal{M}}$ are finite. $MemPool^{\mathcal{M}}$ is infinite.

We explain the intuition behind memory structures. Variables in programs will either have a Boolean value or be pointers. Thus, to represent null and the Boolean values **T** and **F**, we employ the auxiliary relation $Aux^{\mathcal{M}}$ storing 3 elements corresponding to the 3 values. $Addresses^{\mathcal{M}}$ represents the memory cells. The relation $Alloc^{\mathcal{M}}$ is the set of allocated cells, $PossibleTargets^{\mathcal{M}}$ contains all cells which are not allocated, but are pointed to by allocated cells (for technical reasons it possibly contains some other unallocated cells). $MemPool^{\mathcal{M}}$ contains the cells which are not allocated, do not have any field values other than null and **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 $MemPool$ are the candidates for allocation during the run of a program. Since the allocated memory should by finite at any point of the execution of a program, we require that $Alloc^{\mathcal{M}}$ and $PossibleTargets^{\mathcal{M}}$ are finite (see (10)), while the available memory $Addresses^{\mathcal{M}}$ and the memory pool $MemPool^{\mathcal{M}}$ are infinite. Finally, each cell is seen as a record with the fields of τ_{fields}.

2.2 The Description Logic $\mathcal{L}$

$\mathcal{L}$ is defined w.r.t. a vocabulary τ consisting of relation and constant symbols.[2]

Definition 1 (Syntax of $\mathcal{L}$). *The sets of* roles *and* concepts *of $\mathcal{L}$ is defined inductively: (1) every unary relation symbol is a concept (*atomic concept*); (2) every constant symbol is a concept; (3) every binary relation symbol is a role (*atomic role*); (4) if r, s are roles, then $r \cup s$, $r \cap s$, $r \backslash s$ and r^- are roles; (5) if C, D are concepts, then so are $C \sqcap D$, $C \sqcup D$, and $\neg C$; (6) if r is a role and C is a concept, then $\exists r.C$ is also a concept; (7) if C, D are concepts, then $C \times D$ is a role (*product role*).*

The set of formulae *of $\mathcal{L}$ is the closure under $\wedge$,$\vee$,$\neg$,$\rightarrow$ of the atomic formulae: $C \sqsubseteq D$ (*concept inclusion*), where C, D are concepts; $r \sqsubseteq s$ (*role inclusion*), where r, s are roles; and $func(r)$ (*functionality assertion*), where r is a role.*

Definition 2 (Semantics of $\mathcal{L}$). *The semantics is given in terms of structures $\mathcal{M} = (M, \tau, \cdot)$. The extension of $\cdot^{\mathcal{M}}$ from the atomic relations and constants in $\mathcal{M}$ and the satisfaction relation $\models$ are given below. If $\mathcal{M} \models \varphi$, then $\mathcal{M}$ is a* model *of φ. We write $\psi \models \varphi$ if every model of ψ is also a model of φ.*

[1] Here $n \in \{1, 2\}$.
[2] 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.

$$
\begin{array}{llll}
(C \sqcap D)^{\mathcal{M}} & = C^{\mathcal{M}} \cap D^{\mathcal{M}} & (r \sqcap s)^{\mathcal{M}} & = r^{\mathcal{M}} \cap s^{\mathcal{M}} \\
(C \sqcup D)^{\mathcal{M}} & = C^{\mathcal{M}} \cup D^{\mathcal{M}} & (r \sqcup s)^{\mathcal{M}} & = r^{\mathcal{M}} \cup s^{\mathcal{M}} \\
(\neg C)^{\mathcal{M}} & = M \setminus C^{\mathcal{M}} & (r \setminus s)^{\mathcal{M}} & = r^{\mathcal{M}} \setminus s^{\mathcal{M}} \\
(C \times D)^{\mathcal{M}} & = C^{\mathcal{M}} \times D^{\mathcal{M}} & (r^{-})^{\mathcal{M}} & = \{(e,e') \mid (e',e) \in r^{\mathcal{M}})\} \\
(\exists r.C)^{\mathcal{M}} & = \{e \mid \exists e' : (e,e') \in r^{\mathcal{M}}\} & & \\
\mathcal{M} \models C \sqsubseteq D \text{ if } C^{\mathcal{M}} \subseteq D^{\mathcal{M}} & & \mathcal{M} \models r \sqsubseteq s \text{ if } r^{\mathcal{M}} \subseteq s^{\mathcal{M}} & \\
\mathcal{M} \models func(r) \text{ if} \{(e,e_1),(e,e_2)\} \subseteq r^{\mathcal{M}} \text{ implies } e_1 = e_2 & & &
\end{array}
$$

The closure of $\models$ under $\wedge$ $\vee$,$\neg$,$\rightarrow$ is defined in the natural way. We abbreviate: $\top = C \sqcup \neg C$, where C is an arbitrary atomic concept and $\bot = \neg\top$; $\alpha \equiv \beta$ for the formula $\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \alpha$; and $\exists r$ for the concept $\exists r.\top$; (o, o') for the role $o \times o'$. Note that $\top^{\mathcal{M}} = M$ and $\bot^{\mathcal{M}} = \emptyset$ for any structure $\mathcal{M} = (M, \tau, \cdot)$.

2.3 Running Example: Content Invariants in $\mathcal{L}$

Now we make the example from Section 1.1 more precise. The concepts $ELst$ and $PLst$ are interpreted as the sets of elements in the employee list resp. the project list. $mngBy$, $isMngr$ and $wrkFor$ are roles. o_{eHd} and o_{pHd} are the constants which correspond to the heads of the two lists. The invariants of the systems are:

The emploee and project lists are allocated:	$PLst \sqcup ELst$	$\sqsubseteq$	$Alloc$
Projects and employees are distinct:	$PLst \sqcap ELst$	$\sqsubseteq$	$\bot$
$wrkFor$ is set to null for projects:	$PLst$	$\sqsubseteq$	$\exists wrkFor.o_{\text{null}}$
$mngBy$ is set to null for employees:	$ELst$	$\sqsubseteq$	$\exists mngBy.o_{\text{null}}$
$wrkFor$ of employees in the list point to projects in the list or to null:	$\exists wrkFor^{-}.ELst$	$\sqsubseteq$	$PLst \sqcup o_{\text{null}}$
$isMngr$ is a Boolean field:	$\exists isMngr^{-}.ELst$	$\sqsubseteq$	$Boolean$
$mngBy$ of projects point to managers or null:	$\exists mngBy^{-}.PLst$	$\sqsubseteq$	$(ELst \sqcap \exists isMngr.o_{\mathbf{T}}) \sqcup o_{\text{null}}$
The manager of a project must work for the project:	$mngBy \cap (\top \times ELst)$	$\sqsubseteq$	$wrkFor^{-}$

Let the conjunction of the invariants be given by $\varphi_{invariants}$.

Consider S from Section 1. The states of the heap before and after the execution of S can be related by the following $\mathcal{L}$ formulae. $\varphi_{lists-updt}$ and $\varphi_{p-assgn}$. $\varphi_{lists-updt}$ states that the employee list at the end of the program ($ELst$) is equal to the employee list at the beginning of the program ($ELst_{gho}$), and that the project list at the end of the program ($PLst$) is the same as the project list at the beginning of the program ($PLst_{gho}$), except that $PLst$ also contains the new project o_{proj}. $ELst_{gho}$ and $wrkFor_{gho}$ are *ghost relation symbols*, whose interpretations hold the corresponding values at the beginning of S.

$$
\begin{aligned}
\varphi_{lists-updt} &= ELst_{gho} \equiv ELst \wedge PLst_{gho} \sqcup o_{proj} \equiv PLst \\
\varphi_{p-assgn} &= ELst_{gho} \sqcap \exists wrkFor_{gho}.o_{\text{null}} \equiv ELst \sqcap \exists wrkFor.o_{proj}
\end{aligned}
$$

Ghost Symbols. As discussed in Section 2.3, in order to allow invariants of the form $\varphi_{lists-updt} = ELst_{gho} \equiv ELst \wedge PLst_{gho} \sqcup o_{proj} \equiv PLst$ we need ghost symbols. We assume τ contains, for every symbol e.g. $s \in \tau$, the symbol s_{gho}. Therefore, memory structures actually contain *two* snapshots of the memory: one is the *current* snapshot, on which the program operates, and the other is a *ghost* snapshot, which is a snapshot of the memory at the beginning of the program, and which the program does not change or interact with. We denote the two underlying memory structures of $\mathcal{M}$ by $\mathcal{M}_{cur}$ and $\mathcal{M}_{gho}$. Since the interpretations of ghost symbols should not change throughout the run of a program, they will sometime require special treatment.

2.4 The Separation Logic Fragment SLls

The SL that we use is denoted **SLls**, and is the logic from [7] with lists and multiple pointer fields, but without trees. It can express that the heap is partitioned into lists and individual cells. For example, to express that the heap contains only the two lists $ELst$ and $PLst$ we can write the **SLls** formula $\mathrm{ls}(pHd, \mathrm{null}) * \mathrm{ls}(eHd, \mathrm{null})$.

We denote by $var_i \in Var$ and $f_i \in Fields$ the sets of variables respectively fields to be used in **SLls**-formulae. var_i are constant symbols. f_i are binary relation symbols always interpreted as functions. An **SLls**-formula $\Pi \mid \Sigma$ is the conjunction of a *pure part* Π and a *spatial part* Σ. Π is a conjunction of equalities and inequalities of variables and o_{null}. Σ is a *spatial conjunction* $\Sigma = \beta_1 * \cdots * \beta_r$ of formulae of the form $\mathrm{ls}(E_1, E_2)$ and $var \mapsto [f_1 : E_1, \ldots, f_k : E_k]$, where each E_i is a variable or o_{null}. Additionally, Σ can be *emp* and Π can be $\mathbf{T}$. When $\Pi = \mathbf{T}$ we write $\Pi \mid \Sigma$ simply as Σ.

The memory model of [7] is very similar to ours. We give the semantics of **SLls** in memory structures directly due to space constraints. See the full paper [19] for a discussion of the standard semantics of **SLls**. Π is interpreted in the natural way. Σ indicates that $Alloc^{\mathcal{M}}$ is the disjoint union of r parts $P_1^{\mathcal{M}}, \ldots, P_r^{\mathcal{M}}$. If β_i is of the form $var \mapsto [f_1 : E_1, \ldots, f_k : E_k]$ then $|P_i^{\mathcal{M}}| = 1$ and, denoting $v \in P_i^{\mathcal{M}}$, $f_j^{\mathcal{M}}(v) = E_j^{\mathcal{M}}$. If β_i is of the form $\mathrm{ls}(E_1, E_2)]$, then $|P_i^{\mathcal{M}}|$ is a list from $E_1^{\mathcal{M}}$ to $E_2^{\mathcal{M}}$. $E_2^{\mathcal{M}}$ might not belong to $P_i^{\mathcal{M}}$. If $\Sigma = emp$ then $Alloc^{\mathcal{M}} = \emptyset$.

2.5 The Two-Variable Fragment with Counting and Trees CT^2

C^2 is the subset of first-order logic whose formulae contain at most two variables, extended with counting quantifiers $\exists^{\leq k}$, $\exists^{\geq k}$ and $\exists^{=k}$ for all $k \in \mathbb{N}$. W. Charatonik and P. Witkowski [10] recently studied an extension of C^2 which trees which, as we will see, contains both our DL and our SL. CT^2 is the subset of second-order logic of the form $\exists F_1\, \varphi(F_1) \wedge \varphi_{forest}(F_1)$ where $\varphi \in C^2$ and $\varphi_{forest}(F_1)$ says that F_1 is a forest. Note that CT^2 is not closed under negation, conjunction or disjunction. However, CT^2 is closed under conjunction or disjunction with C^2-formulae.

A CT^2-formula φ is *satisfiable in a memory structure* if there is a memory structure $\mathcal{M}$ such that $\mathcal{M} \models \varphi$. We write $\psi \models_m \varphi$ if $\mathcal{M} \models \psi$ implies $\mathcal{M} \models \varphi$ for every memory structure $\mathcal{M}$. Lemma 1 states the crucial property of CT^2 that we use. It follows from [10], by reducing the memory structures to closely related finite structures.[3] (see full version [19]).

Lemma 1. *Satisfiability of CT^2 by memory structures is in NEXPTIME.*

2.6 Embedding $\mathcal{L}$ and SLls in CT^2

$\mathcal{L}$ has a fairly standard reduction (see e.g. [8]) to $\mathcal{C}^2$:

Lemma 2. *For every vocabulary, there exists $tr : \mathcal{L}(\tau) \to C^2(\tau)$ such that for every $\varphi \in \mathcal{L}(\tau)$, φ and $tr(\varphi)$ agree on the truth value of all τ-structures.*

E.g., $tr(C_1 \sqsubseteq C_2) = \forall x\, C_1(x) \to C_2(x)$. The details of tr are given in the full version [19].

The translation of **SLls** requires more work. Later we need the following related translations: α : **SLls** $\to \mathcal{L}$ extracts from the **SLls** properties whatever can be expressed in $\mathcal{L}$. β : **SLls** $\to CT^2$ captures **SLls** precisely.

Given a structure $\mathcal{M}$, $L^{\mathcal{M}}$ is a *singly linked list from $o^{\mathcal{M}}_{var_1}$ to $o^{\mathcal{M}}_{var_2}$ w.r.t. the field $next^{\mathcal{M}}$* if $\mathcal{M}$ satisfies the following five conditions, or it is empty. Except for (5), the conditions are expressed fully in $\mathcal{L}$ below:
(1) $o^{\mathcal{M}}_{var_1}$ belongs to $L^{\mathcal{M}}$; (2) $o^{\mathcal{M}}_{var_2}$ is pointed to by an $L^{\mathcal{M}}$ element; (3) $o^{\mathcal{M}}_{var_2}$ does not belong to $L^{\mathcal{M}}$; (4) Every $L^{\mathcal{M}}$ element is pointed to from an $L^{\mathcal{M}}$ element, except possibly for $o^{\mathcal{M}}_{var_1}$; (5) all elements of $L^{\mathcal{M}}$ are reachable from $o^{\mathcal{M}}_{var_1}$ via $next^{\mathcal{M}}$. Let

$$\begin{array}{ll}
\alpha^1(\text{ls}) = (o_{var_1} \sqsubseteq L) & \alpha^3(\text{ls}) = (o_{var_2} \sqsubseteq \neg L) \\
\alpha^2(\text{ls}) = (o_{var_2} \sqsubseteq \exists next^-.L) & \alpha^4(\text{ls}) = (L \sqsubseteq o_{var_1} \sqcup \exists next^-.L) \\
\alpha_{emp-ls}(\text{ls}) = (L \sqsubseteq \bot) \wedge (o_{var_1} = o_{var_2}) & \\
\alpha(\text{ls}) \qquad = \alpha^1(\text{ls}) \wedge \cdots \wedge \alpha^4(\text{ls}) \vee \alpha_{emp-ls}(\text{ls}) &
\end{array}$$

In memory structures $\mathcal{M}$ satisfying $\alpha(\text{ls})$, if $L^{\mathcal{M}}$ is not empty, then it contains a list segment from $o^{\mathcal{M}}_{var_1}$ to $o^{\mathcal{M}}_{var_2}$, but additionally $L^{\mathcal{M}}$ may contain additional simple $next^{\mathcal{M}}$-cycles, which are disjoint from the list segment. Here we use the finiteness of $Alloc^{\mathcal{M}}$ (which contains $L^{\mathcal{M}}$) and the functionality of $next^{\mathcal{M}}$. A connectivity condition is all that is lacking to express ls precisely. $\alpha(\text{ls})$ can be extended to α : **SLls** $\to \mathcal{L}$ in a natural way (see the full version [19]) such that:

Lemma 3. *For every $\varphi \in$ **SLls**, φ implies $\alpha(\varphi)$ over memory structures.*

To rule out the superfluous cycles we turn to CT^2. Let $\beta^5(\text{ls}) = \forall x \forall y \left[(L(x) \wedge L(y)) \to (F_1(x,y) \leftrightarrow next(x,y))\right] \wedge \forall x \left[\left(L(x) \wedge \forall y\, (L(y) \to \neg F_1(y,x))\right) \to (x \approx o_{var_1})\right]$. $\beta^5(\text{ls})$ states that the forest F_1 coincides with $next$ inside L and that the forest induced by F_1 on L is a tree. Let $\beta(\text{ls}) = \exists F_1\, tr(\alpha(\text{ls})) \wedge \beta^5(\text{ls}) \wedge \varphi_{forest}(F_1)$. $\beta(\text{ls}) \in CT^2$ and it expresses that $L^{\mathcal{M}}$ is a list. The extension of $\beta(\text{ls})$ to the translation function β : **SLls** $\to CT^2$ is natural and discussed in the full version [19]. The full version [19] also discusses the translation of *cyclic data structures* under β.

[3] In fact [10] allows existential quantification over two forests, but will only need one.

Lemma 4. *For every $\varphi \in \mathbf{SL}_{\mathbf{ls}}$: φ and $\beta(\varphi)$ agree on all memory structures.*

CT^2's flexibility allows to easily 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*, and analogue variants of trees.

2.7 Running Example: Shape Invariants

At the loop header of the program S from the introduction, the memory contains two distinct lists, namely *PLst* and *ELst*. *ELst* 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 $\mathbf{SL}_{\mathbf{ls}}$ by the formula: $\varphi_{\ell_l} = \mathbf{T} \mid \mathrm{ls}(eHd, e) * \mathrm{ls}(e, nil) * \mathrm{ls}(pHd, nil)$. The translation $\alpha(\varphi_{\ell_l})$ is given by $P_1 \sqcup P_2 \sqcup P_3 \equiv Alloc \wedge \alpha(\mathrm{ls}(eHd, e, next, P_1)) \wedge \alpha(\mathrm{ls}(e, \mathrm{null}, next, P_2)) \wedge \alpha(\mathrm{ls}(pHd, \mathrm{null}, next, P_3)) \wedge P_1 \sqcap P_2 \equiv \bot \wedge P_1 \sqcap P_3 \equiv \bot \wedge P_2 \sqcap P_3 \equiv \bot \wedge \alpha_{\mathbf{T}}$ The translation from SL assigns concepts P_i to each of the lists. $\alpha_{\mathbf{T}}$ which occurs in $\alpha(\varphi_{\ell_l})$ is the translation of $\Pi = \mathbf{T}$ in φ_{ℓ_l}. In order to clarify the meaning of $\alpha(\varphi_{\ell_l})$ we relate the P_i to the concept names from Section 2.3 and simplify the formula somewhat. Let $\psi_l = P_1 \sqcup P_2 \equiv ELst \wedge P_3 \equiv PLst$. P_1 contains the elements of *ELst* visited in the loop so far. $\alpha(\varphi_{\ell_l})$ is equivalent to: $\alpha'(\varphi_{\ell_l}) = \psi_l \wedge ELst \sqcup PLst \equiv Alloc \wedge ELst \sqcap PLst \equiv \bot \wedge \alpha(\mathrm{ls}(eHd, e, next, P_1)) \wedge \alpha(\mathrm{ls}(e, \mathrm{null}, next, ELst \sqcap \neg P_1)) \wedge \alpha(\mathrm{ls}(pHd, \mathrm{null}, next, PLst))$. We have $\beta^5(\Sigma) = \beta^5(\mathrm{ls}(eHd, e, next, P_1)) \wedge \beta^5(\mathrm{ls}(e, \mathrm{null}, next, ELst \sqcap \neg P_1)) \wedge \beta^5(\mathrm{ls}(pHd, \mathrm{null}, next, PLst))$ and $\beta(\varphi_{\ell_l}) = \exists F_1\, tr(\alpha(\varphi) \wedge \beta^5(\Sigma) \wedge \varphi_{forest}(F_1)$.

3 Content Analysis

3.1 Syntax and Semantics of the Programming Language

Loopless Programs are generated by the following syntax:

$$
\begin{array}{l}
e \;::\; var.f \mid var \mid \mathrm{null} \qquad\qquad (f \in \tau_{\mathrm{fields}},\ o_{var} \in \tau_{\mathrm{var}}) \\
b \;::\; (e_1 = e_2) \mid \sim b \mid (b_1\, and\, b_2) \mid (b_1\, or\, b_2) \mid \mathbf{T} \mid \mathbf{F} \\
S \;::\; var_1 := e_2 \mid var_1.f := e_2 \mid skip \mid S_1; S_2 \mid var := new \mid dispose(var) \mid \\
\qquad if\ b\ then\ S_1\ fi \mid if\ b\ then\ S_1\ else\ S_2\ fi \mid assume(b)
\end{array}
$$

Let *Exp* denote the set of expressions e and *Bool* denote the set of Boolean expressions b. To define the semantics of pointer and Boolean expressions, we extend $f^{\mathcal{M}}$ by $f^{\mathcal{M}}(\mathrm{err}) = \mathrm{err}$ for every $f \in \tau_{\mathrm{fields}}$. We define $\mathcal{E}_e(\mathcal{M}) : Exp \to Addresses^{\mathcal{M}} \cup \{\mathrm{null}, \mathrm{err}\}$ and $\mathcal{B}_b(\mathcal{M}) : Bool \to \{o_{\mathbf{T}}, o_{\mathbf{F}}, \mathrm{err}\}$ (with $\mathrm{err} \notin M$):

$$
\begin{array}{ll}
\mathcal{E}_{var}(\mathcal{M}) = o^{\mathcal{M}}_{var},\ \text{if } o^{\mathcal{M}}_{var} \in Alloc^{\mathcal{M}} & \mathcal{B}_{e_1=e_2}(\mathcal{M}) = \mathrm{err} \text{ if } \mathcal{E}_{e_i}(\mathcal{M}) = \mathrm{err}, i \in \{1,2\} \\
\mathcal{E}_{var}(\mathcal{M}) = \mathrm{err},\ \text{if } o^{\mathcal{M}}_{var} \notin Alloc^{\mathcal{M}} & \mathcal{B}_{e_1=e_2}(\mathcal{M}) = o_{\mathbf{T}},\ \text{if } \mathcal{E}_{e_1}(\mathcal{M}) = \mathcal{E}_{e_2}(\mathcal{M}) \\
\mathcal{E}_{ar.f}(\mathcal{M}) = f^{\mathcal{M}}(\mathcal{E}_{var}(\mathcal{M})) & \mathcal{B}_{e_1=e_2}(\mathcal{M}) = o_{\mathbf{F}},\ \text{if } \mathcal{E}_{e_1}(\mathcal{M}) \neq \mathcal{E}_{e_2}(\mathcal{M})
\end{array}
$$

$\mathcal{B}$ 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 $\langle S, \mathcal{M} \rangle \rightsquigarrow$ abort. Otherwise, the semantics is as listed below. First we assume that in the memory structures involved all relation symbols either belong to τ_{fields}, are ghost symbols or are the required symbols of memory structures (*Alloc*, *Aux*, etc.).

1. $\langle skip, \mathcal{M}\rangle \rightsquigarrow \mathcal{M}$.
2. $\langle var_1 := e_2, \mathcal{M}\rangle \rightsquigarrow [\mathcal{M} \mid o^{\mathcal{M}}_{var_1}$ is set to $\mathcal{E}_{e_2}(\mathcal{M})]$.
3. $\langle var := new, \mathcal{M}\rangle \rightsquigarrow [\mathcal{M} \mid$ For some $t \in MemPool^{\mathcal{M}}$, t is moved to $Alloc^{\mathcal{M}}$ and $o^{\mathcal{M}}_{var}$ is set to $t]$,
4. If $o^{\mathcal{M}}_{var} \notin Alloc^{\mathcal{M}}$, $\langle dispose(var), \mathcal{M}\rangle \rightsquigarrow$ abort; otherwise $\langle dispose(var), \mathcal{M}\rangle \rightsquigarrow [\mathcal{M} \mid o^{\mathcal{M}}_{var}$ is removed from $Alloc^{\mathcal{M}}]$.
5. $\langle S_1; S_2, \mathcal{M}\rangle \rightsquigarrow \langle S_2, \langle S_1, \mathcal{M}\rangle\rangle$
6. $\langle if\ b\ then\ S_{\mathbf{T}}\ else\ S_{\mathbf{F}}, \mathcal{M}\rangle \rightsquigarrow \langle S_{tv}, \mathcal{M}\rangle$ where $tv = \mathcal{B}_b(\mathcal{M})$.
7. $\langle if\ b\ then\ S\ , \mathcal{M}\rangle \rightsquigarrow \langle if\ b\ then\ S\ else\ skip\ fi, \mathcal{M}\rangle$.
8. If $\mathcal{B}_b(\mathcal{M}) = \mathbf{T}$, then $\langle assume(b), \mathcal{M}\rangle \rightsquigarrow \mathcal{M}$; otherwise $\langle assume(b), \mathcal{M}\rangle \rightsquigarrow$ abort.

If $\mathcal{M}$ is a memory structure and $\langle S, \mathcal{M}\rangle \rightsquigarrow \mathcal{M}'$, then $\mathcal{M}'$ is a memory structure.

Now consider a relation symbol e.g. *ELst*. If $\langle S, \mathcal{M}\rangle \rightsquigarrow \mathcal{M}'$, then we want to think of $ELst^{\mathcal{M}}$ and $ELst^{\mathcal{M}'}$ as the employee list before and after the execution of S. However, the constraints that $ELst^{\mathcal{M}}$ and $ELst^{\mathcal{M}'}$ are lists and that $ELst^{\mathcal{M}'}$ is indeed obtained from from $ELst^{\mathcal{M}}$ by running S will be expressed as formulae. In the $\rightsquigarrow$ relation, we allow any values for $ELst^{\mathcal{M}}$ and $ELst^{\mathcal{M}'}$.

For any tuple $\bar{R}$ of relation symbols which do not belong to τ_{fields}, are not ghost symbols and are not the required symbols of memory structures (*Alloc*, *Aux*, etc.), we extend $\rightsquigarrow$ as follows: if $\langle S, \mathcal{M}\rangle \rightsquigarrow \mathcal{M}'$, then $\left\langle S, \left\langle \mathcal{M}, \bar{R}^{\mathcal{M}}\right\rangle\right\rangle \rightsquigarrow \left\langle \mathcal{M}', \bar{R}^{\mathcal{M}'}\right\rangle$, for any tuples $\bar{R}^{\mathcal{M}}$ and $\bar{R}^{\mathcal{M}'}$.

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

Definition 3 (Program). *A* program *is* $G = \langle V, E, \ell_{init}, shp, cnt, \lambda\rangle$ *such that* $G = (V, E)$ *is a directed graph with no multiple edge but possibly containing self-loops,* $\ell_{init} \in V$ *has in-degree* 0*,* $shp : V \to \mathbf{SL}_{\mathbf{ls}}$*,* $cnt : V \to \mathcal{L}(\tau)$ *are functions, and* λ *is a function from* E *to the set of loopless programs.*

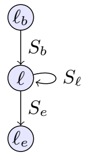
Here is the code S from the introduction:

$V = \{\ell_b, \ell_l, \ell_e\}$ $\qquad E = \{(\ell_b, \ell_l), (\ell_l, \ell_l), (\ell_l, \ell_e)\}$

$\lambda(\ell_b, \ell_l) = S_b$ $\qquad \ell_{init} = \ell_b$

$\lambda(\ell_l, \ell_l) = assume(\sim (e\ =\ \text{null})); S_{\ell_l}$

$\lambda(\ell_l, \ell_e) = assume(e\ =\ \text{null}); S_e$



S_b, S_{ℓ_l} 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 *shp* and *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 that 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.

Definition 4 ($\rightsquigarrow^*$ for paths). *Given a program G, a path P in G, and memory structures $\mathcal{M}_1$ and $\mathcal{M}_2$ we define whether $\langle P, \mathcal{M}_1\rangle \rightsquigarrow^* \mathcal{M}_2$ holds inductively. If P is empty, then $\langle P, \mathcal{M}_1\rangle \rightsquigarrow^* \mathcal{M}_2$ iff $\mathcal{M}_1 = \mathcal{M}_2$. If e_t is the last edge of P, then $\langle P, \mathcal{M}_1\rangle \rightsquigarrow^* \mathcal{M}_2$ iff there is $\mathcal{M}_3$ such that $\langle P\backslash\{e_t\}, \mathcal{M}_1\rangle \rightsquigarrow^* \mathcal{M}_3$ and $\langle \lambda(e_t), \mathcal{M}_1\rangle \rightsquigarrow^* \mathcal{M}_3$. $P\backslash\{e_t\}$ denotes the path obtained from P by removing the last edge e_t.*

3.2 Hoare-Style Proof System

Now we are ready to state our two-step verification methodology that we formulated in Section 1 precisely. Our methodology assumes a program P as in Definition 3 as input (ignoring the *shp* and *cnt* functions for the moment).

I. Shape Analysis. The user annotates the program locations with SL formulae from $\mathbf{SL_{ls}}$ (stored in the *shp* function of P). Then the user proves the validity of the $\mathbf{SL_{ls}}$ annotations, for example, by using techniques from [7].

II. Content Analysis. The user annotates the program locations with $\mathcal{L}$-formulae that she wants to verify (stored in the *cnt* function of P). We point out that an annotation $cnt(\ell)$ can use the concepts occurring in $\alpha(shp(\ell))$ (recall that $\alpha : \mathbf{SL_{ls}} \to \mathcal{L}$ maps SL formulae to $\mathcal{L}$-formulae).

In the rest of the paper we discuss how to verify the *cnt* annotations. In Section 3.3 we describe how to derive a verification condition for every program edge. The verification conditions rely on the backwards propagation function Θ for $\mathcal{L}$-formulae which we introduce in Section 3.5. The key point of our methodology is that the validity of the verification conditions can be discharged automatically by a satisfiability solver for CT^2-formulae. We show that all the verification conditions are valid if and only if *cnt* is inductive. Intuitively, *cnt* being inductive ensures that the annotations *cnt* can be used in an inductive proof to show that all reachable memory structures indeed satisfy the annotations $cnt(\ell)$ at every program location ℓ (see Definition 6 below).

3.3 Content Verification

We want to prove that, for every initial memory structure $\mathcal{M}_1$ from which the computation satisfies *shp* and which satisfies the content pre-condition $cnt(\ell_{init})$, the computation satisfies *cnt*. Here are the corresponding verification conditions, which annotate the vertices of G:

Definition 5 (Verification conditions). *Given a program G, VC is the function from E to $\mathcal{L}$ given for $e = (\ell_0, \ell)$ by $VC(e) = \neg\big[\beta(shp(\ell_0)) \wedge tr(cnt(\ell_0)) \wedge tr\left(\Theta_{\lambda(e)}\left(\alpha(shp(\ell)) \wedge \neg cnt(\ell)\right)\right)\big]$* $VC(e)$ holds *if $VC(e)$ is a tautology over memory structures ($\top \models_m VC(e)$).*

Θ is discussed in Section 3.5. As we will see, $VC(\ell_0, \ell)$ expresses that when running the loopless program $\lambda(e)$ when the memory satisfies the the annotations of ℓ_0, and when the shape annotation of ℓ is at least partly true (i.e., when $\alpha(shp(\ell)))$, the content annotation of ℓ holds.

Let J be a set of memory structures. For a formula in CT^2 or $\mathcal{L}$, we write $J \models \varphi$ if, for every $\mathcal{M} \in J$, $\mathcal{M} \models \varphi$. Let *Init* be a set of memory structures.

Definition 6 (Inductive program annotation). *Let $f : V \to CT^2$. We say f is* inductive *for Init if (i) Init $\models f(\ell_{init})$, and (ii) for every edge $e = (\ell_1, \ell_2) \in E$ and memory structures $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_1 \models f(\ell_1)$ and $\langle \lambda(e), \mathcal{M}_1 \rangle \rightsquigarrow \mathcal{M}_2$, we have $\mathcal{M}_2 \models f(\ell_2)$. We say shp is inductive for Init if the composition $shp \circ \beta : V \to CT^2$ is inductive for Init. We say cnt* is inductive for *Init* relative to *shp if shp is inductive for Init and $g : V \to CT^2$ is inductive for Init, where $g(\ell) = tr(cnt(\ell)) \wedge \beta(shp(\ell))$.*

Theorem 1 (Soundness and Completeness of the Verification Conditions). *Let G be a program such that shp is inductive for Init and Init $\models cnt(\ell_{init})$. The following statements are equivalent:*
(i) For all $e \in E$, $VC(e)$ holds. *(ii) cnt is inductive for Init relative to shp.*

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

In particular, $Reach(\ell_{init}) = Init$. The proof of Theorem 1 and its consequence Theorem 2 below are given in the full version [19].

Theorem 2 (Soundness of the Verification Methodology). *Let G be a program such that shp is inductive for Init and Init $\models cnt(\ell_{init})$. If for all $e \in E$, $VC(e)$ holds, then for $\ell \in V$, $Reach(\ell) \models cnt(\ell)$.*

3.4 Running Example: General Methodology

To verify the correctness of the code S, the *shp* and *cnt* annotations must be provided. The shape annotations of program S are: $shp(\ell_b) = \mathrm{ls}(eHd, \mathrm{null}) * \mathrm{ls}(pHd, \mathrm{null})$, $shp(\ell_e) = (proj = pHd) \,|\, \mathrm{ls}(eHd, \mathrm{null}) * \mathrm{ls}(pHd, \mathrm{null})$, and $shp(\ell_l) = \varphi_{\ell_l}$, where φ_{ℓ_l} is from Section 2.7.

The three content annotations require that the system invariants $\varphi_{invariants}$ from Section 2.3 hold. The post-condition additionally requires that $\varphi_{p-assgn}$ and $\varphi_{lists-updts}$ hold. Recall $\varphi_{p-assgn}$ states that every employee which was not assigned a project, is assigned to o_{proj}. $\varphi_{lists-updts}$ states that the content of the two lists remain unchanged, except that the project o_{proj} is inserted to $PLst$.

In order to interact with the translations $\alpha(shp(\cdots))$ of the shape annotations, we need to related the P_i to the concepts $ELst$ and $PLst$. In Section 2.7 we defined ψ_l, which relates the P_i generated by α on $shp(\ell_l)$. We have $\psi_{\ell_b} = \psi_{\ell_e} = P_1 \equiv ELst \wedge P_2 \equiv PLst$. Then $cnt(\ell_b) = \psi_{\ell_b} \wedge \varphi_{invariants}$, $cnt(\ell_l) = \psi_{\ell_l} \wedge \varphi_{invariants} \wedge \varphi_{lists-updt} \wedge \varphi_{p-as-\ell_l}$, and $cnt(\ell_e) = \psi_{\ell_e} \wedge \varphi_{invariants} \wedge \varphi_{lists-updt} \wedge \varphi_{p-assgn}$, where $\varphi_{p-as-\ell_l} = P_1 \sqcap \exists wrkFor_{gho}.o_{\mathrm{null}} \equiv P_1 \sqcap \exists wrkFor.o_{proj}$. $\varphi_{p-as-\ell_l}$ states that, in the part of $ELst$ 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 $wrkFor$) is assigned to the project $proj$. $\varphi_{p-as-\ell_l}$ makes no demands on elements of $ELst$ which have not been reach in the loop so far. The verification conditions of G are, for each $(\ell_1, \ell_2) \in E$, $VC(\ell_1, \ell_2) = \neg\big[\beta(shp(\ell_1)) \wedge tr(cnt(\ell_1)) \wedge tr\big(\Theta_{\lambda(l_1,l_2)}(\alpha(shp(\ell_2)) \wedge \neg cnt(\ell_2))\big)\big]$.

The verification conditions $VC(e)$ express that the loopless programs on the edges e of G satisfy their annotations. To prove the correctness of G w.r.t. $VC(e)$ using Theorem 2, we prove that $VC(e)$, $e \in E$, hold, in order to get as a conclusion that $Reach(\ell) \models cnt(\ell)$, for all $\ell \in V$.

3.5 Backwards Propagation and the Running Example

Here we shortly discuss the backwards propagation of a formula along a loopless program S. Let $\langle S, \mathcal{M}_1 \rangle \rightsquigarrow \mathcal{M}_2$ where $\mathcal{M}_1$ and $\mathcal{M}_2$ are memory structures over the same vocabulary τ. E.g., in our running example, for $i = 1, 2$, $\mathcal{M}_i$ is $\big\langle M, ELst^{\mathcal{M}_i}, next^{\mathcal{M}_i}, mngBy^{\mathcal{M}_i}, \cdots, ELst_{gho}^{\mathcal{M}_i}, next_{gho}^{\mathcal{M}_i}, \cdots, Alloc^{\mathcal{M}_i}, Aux^{\mathcal{M}_i} \cdots \big\rangle$. We will show how to translate a formula for $\mathcal{M}_2$ to a formula for an extended $\mathcal{M}_1$. Fields and variables in $\mathcal{M}_2$ will be translated by the backwards propagation into expressions involving elements of $\mathcal{M}_1$. For ghost symbols s_{gho}, $s_{gho}^{\mathcal{M}_1}$ will be used instead of $s_{gho}^{\mathcal{M}_2}$ since they do not change during the run of the program. Let $\tau^{rem} \subseteq \tau$ be the set of the remaining symbols, i.e. the symbols of $\tau \setminus (\{PossibleTargets, MemPool\} \cup \tau_{\text{fields}})$ which are not ghost symbols, for example $ELst$, but not $ELst_{gho}$, $next$ or $mngBy$. We need the result of the backwards propagation to refer to the interpretations of symbols in τ^{rem} from $\mathcal{M}_2$ rather than $\mathcal{M}_1$. Therefore, these interpretations are copied as they are from $\mathcal{M}_2$ and added to $\mathcal{M}_1$ as follows. For every $R \in \tau^{rem}$, we add a symbol R^{ext} for the copied relation. We denote by $(\bar{R}^{ext})^{\mathcal{M}_1}$ the tuple $\big((R^{ext})^{\mathcal{M}_1} : (R^{ext})^{\mathcal{M}_1} = R^{\mathcal{M}_2} \text{ and } R \in \tau^{rem}\big)$ Let τ^{ext} extend τ with R^{ext} for each $R \in \tau^{rem}$. The backwards propagation updates the fields and variables according to the loopless code. Afterwards, we substitute the symbols $R \in \tau^{rem}$ in φ with the corresponding R^{ext}. We present here a somewhat simplified version of the backwards propagation lemma. The precise version is similar in spirit and is in the full version [19]

Lemma 5 (Simplified). *Let S be a loopless program, let $\mathcal{M}_1$ and $\mathcal{M}_2$ be memory structures, and φ be an $\mathcal{L}$-formula over τ. (1) If $\langle S, \mathcal{M}_1 \rangle \rightsquigarrow \mathcal{M}_2$, then: $\mathcal{M}_2 \models \varphi$ iff $\big\langle \mathcal{M}_1, (\bar{R}^{ext})^{\mathcal{M}_1} \big\rangle \models \Theta_S(\varphi)$. (2) If $\langle S, \mathcal{M}_1 \rangle \rightsquigarrow$ abort, then $\big\langle \mathcal{M}_1, (\bar{R}^{ext})^{\mathcal{M}_1}, \big\rangle \not\models \Theta_S(\varphi)$.*

As an example of the backwards propagation process, we consider a formula from Section 3.4, which is part of the content annotation of ℓ_l and perform the backwards propagation on the loopless program inside the loop: $\varphi_{p-as-\ell_l} = P_1 \sqcap \exists wrkFor_{gho}.o_{\text{null}} \equiv P_1 \sqcap \exists wrkFor.o_{proj}$ Since $next$ does not occur in $\varphi_{p-as-\ell_l}$, backwards propagation of $\varphi_{p-as-\ell_l}$ over $e := e.next$ does not change the formula (however $\alpha(shp(\ell_l))$ by this command). The backwards propagation of the *if* command gives $\Psi_{S_{\ell_l}}(\varphi_{p-as-\ell_l}) = \big(\neg(\exists wrkFor^{-}.o_e \equiv o_{\text{null}}) \wedge \varphi_{p-as-\ell_l}\big) \vee \exists wrkFor^{-}.o_e \equiv o_{\text{null}} \wedge \Psi_{e.wrkFor:=proj}(\varphi_{p-as-\ell_l})\big)$ and $\Psi_{e.wrkFor:=proj}(\varphi_{p-as-\ell_l}) = P_1 \sqcap \exists wrkFor_{gho}.o_{\text{null}} \equiv P_1 \sqcap \exists((wrkFor \backslash (o_e \times \top)) \cup (o_e, o_{proj})).o_{proj}$. $\Psi_{e.wrkFor:=proj}(\varphi_{p-as-\ell_l})$ is obtained from $\varphi_{p-as-\ell_l}$ by substituting the $wrkFor$ role with the correction $((wrkFor \backslash (o_e \times \top)) \cup (o_e, o_{proj}))$ which updates the value of o_e in $wrkFor$ to $proj$. $\Phi_{S_{\ell_l}}(\varphi_{p-as-\ell_l})$ is obtained from

$\Psi_{S_{\ell_l}}(\varphi_{p-as-\ell_l})$ by subtituting P_1 with ${P_1}^{ext}$. Θ is differs from Φ from technical reasons related to aborting computations (see the full version [19]).

4 Related Work

Shape Analysis attracted considerable attention in the literature. The classical introductory paper to SL [24] presents an expressive SL which turned out to be undecidable. We have restricted our attention to the better behaved fragment in [7]. The work on SL focuses mostly on shape rather than content in our sense. SL has been extended to object oriented languages, cf. e.g. [23,11], where shape properties similar to those studied in the non objected 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. [25] use 3-valued logic to perform shape analysis. Regional logic is used to check correctness of program with shared dynamica memory areas [5]. [16] uses nested tree automata to represent the heap. [21] combines monadic second order logic with SMT solvers.

Description Logics have not been considerd for verification of programs with dynamically allocated memory, with the exception of [13] whose use (mostly undecidable) DLs to express shape-type invariants, ignoring content information. In [9] 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 [4,14] and their references). *Temporal Description Logics*, which combine classic DLs with classic temporal logics, have also received significant attention in the last decade (see [20] for a survey).

Related Ideas. Some recent papers have studied verification strategies which use information beyond the semantics of the source code. E.g., [18] is using diagrams from design documentation to support verification. [12,1] infer the intended use of program variables to guide a program analysis. Instead of starting from code and verifying its correctness, [15] explores how to declaratively specify data structures with sharing and how to automatically generate code from this specification. Given the importance of both DL as a formalism of content representation and of program verification, and given that both are widely studied, we were surprised to find little related work. However, we believe this stems from large differences between the research in the two communities, and from the interdisciplinary nature of the work involved.

References

1. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., von Rhein, A.: Domain types: Abstract-domain selection based on variable usage. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 262–278. Springer, Heidelberg (2013)
2. Artale, A., Calvanese, D., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Reasoning over extended ER models. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 277–292. Springer, Heidelberg (2007)

3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic handbook: theory, implementation, and applications. Cambridge University Press (2003)
4. Baader, F., Zarrieß, B.: Verification of golog programs over description logic actions. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) FroCoS 2013. LNCS, vol. 8152, pp. 181–196. Springer, Heidelberg (2013)
5. Banerjee, A., Naumann, D.A., Rosenberg, S.: Local reasoning for global invariants, part I: Region logic. J. ACM 60(3), 18 (2013)
6. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artificial Intelligence 168(1-2), 70–118 (2005)
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
8. Borgida, A.: On the relative expressiveness of description logics and predicate logics. Artif. Intell. 82(1-2), 353–367 (1996)
9. Calvanese, D., Ortiz, M., Šimkus, M.: Evolving graph databases under description logic constraints. In: Proc. of DL, pp. 120–131 (2013)
10. Charatonik, W., Witkowski, P.: Two-variable logic with counting and trees. In: LICS, pp. 73–82 (2013)
11. Chin, W., David, C., Nguyen, H.H., Qin, S.: Enhancing modular oo verification with separation logic. In: POPL, pp. 87–99. ACM (2008)
12. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. In: FMCAD, pp. 226–230 (2013)
13. Georgieva, L., Maier, P.: Description Logics for shape analysis. In: SEFM, pp. 321–331 (2005)
14. De Giacomo, G., Lespérance, Y., Patrizi, F.: Bounded situation calculus action theories and decidable verification. In: Proc. of KR (2012)
15. Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M.: Data structure fusion. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 204–221. Springer, Heidelberg (2010)
16. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 740–755. Springer, Heidelberg (2013)
17. Ishtiaq, S.S., O'Hearn, P.W.: Bi as an assertion language for mutable data structures. In: POPL, pp. 14–26. ACM (2001)
18. James, D., Leonard, T., O'Leary, J., Talupur, M., Tuttle, M.R.: Extracting models from design documents with mapster. In: PODC (2008)
19. Kotek, T., Simkus, M., Veith, H., Zuleger, F.: Extending alcqio with reachability. CoRR, abs/1402.6804 (2014)
20. Lutz, C., Wolter, F., Zakharyaschev, M.: Temporal description logics: A survey. In: Proc. of TIME. IEEE Computer Society (2008)
21. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622. ACM, USA (2011)
22. W3C OWL Working Group. OWL 2 Web Ontology Language: Document Overview. W3C Recommendation (October 27, 2009)
23. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. SIGPLAN Not. 43(1), 75–86 (2008)
24. Reynolds, J.C.: Separation Logic: A logic for shared mutable data structures. In: Proc. of LICS, pp. 55–74. IEEE Computer Society, Washington, DC (2002)
25. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 530–545. Springer, Heidelberg (2004)