Description Logics for Shape Analysis

Lilia Georgieva
School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK
lilia@macs.hw.ac.uk

Patrick Maier
Max-Planck-Institut für Informatik
Saarbrücken, Germany
maier@mpi-inf.mpg.de

Abstract

Verification of programs requires reasoning about sets of program states. In case of programs manipulating pointers, program states are pointer graphs. Verification of such programs involves reasoning about unbounded sets of graphs.

Three-valued shape analysis (Sagiv et. al.) is an approach based on explicit manipulation of 3-valued shape graphs, which abstract sets of pointer graphs. Other approaches use symbolic representations, e. g., by describing (sets of) graphs as logical formulas. Unfortunately, many resulting logics are either undecidable or cannot express crucial properties like reachability and separation.

In this paper, we investigate an alternative approach. We study well-known description logics as a framework for symbolic shape analysis. We propose a predicate abstraction based shape analysis, parameterized by description logics to represent the abstraction predicates. Depending on the particular logic chosen sharing, reachability and separation in pointer data structures are expressible.

1. Introduction

The goal of shape analysis [22] is to derive and re-establish data structure invariants and identify alias relationship between access paths in the heap. Shape analysis has been used for (partial) program verification. Its applications include static detection of memory and logical errors (dereferencing $NULL$ pointers, dereferencing dangling pointers, and memory leaks), establishing properties of dynamically allocated data structures, code optimizations. It is challenging because the aliasing relationships are complex and there is no bound on the size of run-time data structures.

Among the approaches that have been used to study shape analysis, prominent are the symbolic ones. For instance, symbolic shape analysis based on 3-valued logic uses an over-approximation and has been studied in [17, 22, 28]. The semantics of statements and the invariants are expressed in first-order logic with transitive closure. The logic is interpreted over 3-valued structures, according to Kleene's 3-valued interpretation of first-order logic, and is undecidable in general. Other symbolic approaches for shape analysis do not use 3-valued interpretations [13, 14, 16, 21, 27]. However, the logics employed are either undecidable or of severely restricted expressive power; for instance, when augmented with transitive closure or fixpoint operators for modeling reachability, most decidable first-order fragments become undecidable.

In this paper we focus on an alternative approach. We use description logics (DL) which are extensively studied and well-understood languages for graph structures. We study the applications of the languages for symbolic representation of sets of shape graphs and shape analysis. We use DLs to characterize the heap allocated data structures that a program manipulates. The state of the heap in a pointer program is modeled as a finite graph. For each pointer-valued field in a data structure the graph has a corresponding edge relation. Properties of such graphs can be described in DLs, where the edge relations are represented as roles.

We express the properties of recursively defined data structures such as lists (including singly and doubly linked lists, cyclic lists, and lists with a pointer to the last element), binary trees, and DAGs in various fragments of $\mu\mathcal{ALCQO}^{-1}$, which is an extension of the basic DL $\mathcal{ALC}$ with fixpoints, number restrictions, nominals, and inverse roles. Two main issues arise: decidability of the logics and expressive power.

The DL $\mu\mathcal{ALCQO}^{-1}$ is undecidable [3]. We consider weaker logics as sound approximations, e. g., the logics $\mu\mathcal{ALCO}^{-1}$ and $\mu\mathcal{ALCQ}^{-1}$ both of which are decidable [4, 23]. Reasoning for and complexity of description logics have been studied and well-understood. Tools for reasoning with expressive and even undecidable description logics exist, e. g., FaCT [12], Racer [10].

The structure of the paper is as follows. In Section 2 we define the syntax and semantics of description logics. We encode shape types in DL in Section 3. Shape analysis for description logics is presented in Section 4. We comment on related work in Section 5. Finally, Section 6 concludes.

2. Description Logics

In this section we briefly introduce the conventional syntax and semantics of description logics following [1].

For representation of shape graphs we use extensions of the basic description logic $\mathcal{ALC}$ [24] with nominals, number restrictions, fixpoint concepts, universal role, functional roles, and inverse roles. The syntax of such description logics defines concept expressions (commonly denoted by letters C and D) and role expressions (denoted by R and S). Concept and role expressions are constructed inductively from a finite set of atomic concepts and roles by applying the constructors as shown in table 1. Note that the formation of fixpoint concepts $\mu Z.C$ and $\nu Z.C$ is restricted to cases where the concept variable Z is positive in C (i. e., always occurs under an even number of negations). Besides the Boolean concept constructors, we will also use concept implication $\Rightarrow$ and equivalence $\Leftrightarrow$, which are defined in the usual way.

In the remainder of this paper, a (description logic) formula always means a concept expression. Given formulas C and D, we call $C \sqsubseteq D$ a concept inclusion, $C \doteq D$ a concept equality. A TBox (T for terminological) is a finite set of concept inclusions and equalities.

Semantically, concept and role expressions are interpreted as unary and binary relations in labeled graphs. Formally, an interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ and an interpretation function which assigns to every atomic concept A a subset $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended to concept and role expressions as shown in table 1. Note that this semantics is well-defined for fixpoint concepts due to the polarity restriction on concept variables.

Let $\mathcal{I}$ be an interpretation. We say that $\mathcal{I}$ satisfies a formula C if $C^{\mathcal{I}} \neq \emptyset$. $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$, written $\mathcal{I} \models C \sqsubseteq D$, if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Likewise, we define $\mathcal{I} \models C \doteq D$. $\mathcal{I}$ satisfies a TBox $\mathcal{T}$, written $\mathcal{I} \models \mathcal{T}$, if it satisfies every inclusion and every equality in $\mathcal{T}$.

Let $\mathcal{T}$ be a TBox. We call a formula C satisfiable w. r. t. $\mathcal{T}$ if there is an interpretation $\mathcal{I}$ satisfying both $\mathcal{T}$ and C. We say that $\mathcal{T}$ implies an inclusion $C \sqsubseteq D$, denoted by $\mathcal{T} \models C \sqsubseteq D$, if for all interpretations $\mathcal{I}$, $\mathcal{I} \models \mathcal{T}$ implies $\mathcal{I} \models C \sqsubseteq D$. We write $\mathcal{T} \models_{\mathrm{fin}} C \sqsubseteq D$, if $\mathcal{I} \models \mathcal{T}$ implies $\mathcal{I} \models C \sqsubseteq D$ for all finite interpretations $\mathcal{I}$. We define $\mathcal{T} \models C \doteq D$ resp. $\mathcal{T} \models_{\mathrm{fin}} C \doteq D$ analogously.

Description logics are distinguished by the available constructors. Extending the basic logic $\mathcal{ALC}$, which features the Boolean concept constructors, value restriction, and existential quantification, we obtain more expressive languages, see table 2. Note that we call a role R functional in a given interpretation I if $R^{\mathcal{I}}$ is a functional relation, i. e., if $\mathcal{I} \models \top \sqsubseteq \leq 1\, R.\top$.

Desc. logic	Concept/Role constructors beyond $\mathcal{ALC}$
$\mathcal{ALCO}_f^U$	nominals, universal role, functional roles
$\mathcal{ALCO}_f^{-1,U}$	+ inverse roles
$\mathcal{ALCQO}^{-1,U}$	+ qualifi ed number restrictions
$\mu\mathcal{ALCO}_f$	fi xpoints, nominals, functional roles
$\mu\mathcal{ALCO}_f^{-1}$	+ inverse roles

Table 2. Names of description logics.

3. Shape Types

In this section we introduce the notion of shape, defined in terms of description logics and show that description logics allow for the encoding of useful data structures.

Extensions of imperative languages, such as Shape-C [6], in practice use only a subset of possible shape graphs, corresponding to rooted pointer structures. The graphs are characterized by the following properties: (i) Relations are either unary or binary. (ii) Each unary relation is satisfied by exactly one node in the graph. (iii) Binary relations are partial functions. (iv) The whole graph can be traversed starting from its roots. These conditions correspond to properties of rooted data structures expressible in description logics; in particular, binary relations are represented by functional roles due to (iii), whereas program variables are represented by nominals due to (ii). Examples of shape types such as singly linked list, doubly linked list, linked list with pointer to the last element, cyclic list, and binary tree are given in figure 1.

In this section we study DLs with fixpoints and DLs with transitive closure. More precisely, we consider extensions of the description logic $\mathcal{ALCO}_f$ with fixpoint operators (which can be used to express transitive closure). These logics are related to the modal μ-calculus [15, 25], which was originally introduced to describe program behavior.

Singly linked list. We view a singly linked list as a set of heap cells in which each node is the terminating $NULL$ node or a structure with a pointer field $next$ pointing to the next node. For the purpose of shape analysis, we abstract from data fields in the structure.

In terms of description logics, we can represent a singly linked list as the concept of all nodes that can reach the $NULL$ concept I_{NULL} via the reflexive-transitive closure of the functional role R_{next}. A TBox expressing that program variable x (modeled by the nominal I_x) points to a singly linked list is

$$\mathcal{T}_1 = \{List \doteq \exists R_{next}^{*}.I_{NULL},\ \ I_x \sqsubseteq List\}$$

where $\exists R_{next}^{*}.I_{NULL}$ is a shorthand for the fixpoint concept $\mu Z.(I_{NULL} \sqcup \exists R_{next}.Z)$. The property that the variables x and y point to separated lists (i. e., there is no heap cell

Concept/Role constructor	Syntax	Semantics
Atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Nominal	I	$I^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ with $\lvert I^{\mathcal{I}}\rvert = 1$
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Complement	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Value restriction	$\forall R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \Rightarrow b \in C^{\mathcal{I}}\}$
Existential quantifi cation	$\exists R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$
Qualifi ed number restriction	$\geq n\, R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\rvert \geq n\}$
	$\leq n\, R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\rvert \leq n\}$
	$=n\, R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \lvert\{b \in \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\rvert = n\}$
Fixpoint	$\mu Z.C$	least fi xpoint of $\lambda\mathcal{E} \subseteq \Delta^{\mathcal{I}}.C[\mathcal{E}/Z]^{\mathcal{I}}$
	$\mu Z.C$	greatest fi xpoint of $\lambda\mathcal{E} \subseteq \Delta^{\mathcal{I}}.C[\mathcal{E}/Z]^{\mathcal{I}}$
Atomic role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Universal role	U	$\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Inverse role	R^{-}	$\{(b,a) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}$
Role union	$R \sqcup S$	$R^{\mathcal{I}} \cup S^{\mathcal{I}}$
Transitive closure	$R^{*},\ R^{+}$	$\bigcup_{n\geq 0} (R^{\mathcal{I}})^{n},\ \bigcup_{n\geq 1} (R^{\mathcal{I}})^{n}$

Table 1. Syntax and semantics of description logic constructs.

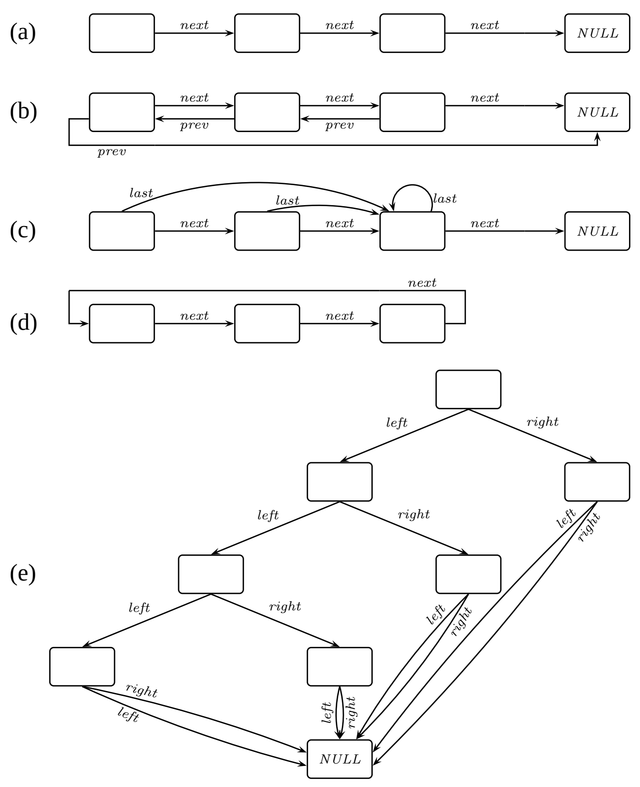

Figure 1. Examples of shape types: (a) singly linked list, (b) doubly linked list, (c) singly linked list with pointer to the last element, (d) cylic singly linked list, (e) binary tree.

except $NULL$ that can reach both x and y by traversing $next$ pointers backwards) can be expressed by the TBox:

$$\mathcal{T}_2 = \{List \doteq \exists R^{*}_{next}.I_{NULL},\ \ I_x \sqsubseteq List,\ \ I_y \sqsubseteq List,$$
$$\exists (R^{-}_{next})^{*}.I_x \sqcap \exists (R^{-}_{next})^{*}.I_y \doteq I_{NULL}\}$$

Doubly linked list. We model that the program variable x points to a doubly linked list with $next$ and $prev$ pointer fields by the TBox:

$$\mathcal{T}_3 = \{List \doteq \exists R^{*}_{next}.I_{NULL},\ \ I_x \sqsubseteq List,$$
$$I_x \sqsubseteq \exists R_{prev}.I_{NULL},\ \ I_{NULL} \sqsubseteq \exists R^{-}_{next}.\exists R^{*}_{prev}.I_x\}$$

The first axiom states that $NULL$ is reachable from x by following $next$ pointers. The second axiom states that the $prev$ pointer from x points to $NULL$. The third axiom states that x is reachable from $NULL$ by following the $next$ pointer backward once (yielding the last element of the list) and then following the $prev$ pointers.

Singly linked list with pointer to the last element. A TBox expressing that x points to a singly linked list with pointer to the last element contains, in addition to the axioms for singly linked lists, two more axioms:

$$\mathcal{T}_4 = \mathcal{T}_1 \cup \{Last \doteq \exists R_{next}.I_{NULL},\ \ \exists R_{last}.Last \doteq \neg I_{NULL}\}$$

The first defines the last element of the list (i. e., the predecessor of $NULL$), the second states that for all nodes except $NULL$, there is a $last$ pointer to the last element.

Cyclic singly linked list. The property that x points to a cyclic singly linked list is succinctly expressed by the TBox

$$\mathcal{T}_5 = \{I_x \sqsubseteq \exists R^+_{next}.I_x\}$$

where $\exists R^+_{next}.I_x$ is a shorthand for $\exists R_{next}.\exists R^*_{next}.I_x$. Alternatively, we can express that y is on a cyclic list pointed by x by the TBox:

$$\begin{aligned}\mathcal{T}_6 = \{&\mathit{Reach}_x \doteq \exists R^*_{next}.I_x,\\ &\mathit{List} \doteq \nu Z.\exists R^-_{next}.\mathit{Reach}_x \sqcap Z,\ \ I_y \sqsubseteq \mathit{List}\}\end{aligned}$$

Trees and DAGs. The property that a program variable x points to a binary DAG with *left* and *right* pointers is captured by the TBox:

$$\begin{aligned}\mathcal{T}_7 = \{&DAG \doteq \forall(R_{\mathit{left}} \sqcup R_{\mathit{right}})^*.I_{NULL},\ \ I_x \sqsubseteq DAG,\\ &\exists R_{\mathit{left}}.\top \sqcap \exists R_{\mathit{right}}.\top \doteq \neg I_{NULL}\}\end{aligned}$$

Thereby, the defined concept DAG is the set of all heap cells from which $NULL$ is reachable regardless of the followed path. Note that $\forall(R_{\mathit{left}} \sqcup R_{\mathit{right}})^*.I_{NULL}$ is again expressible using fixpoints because

$$\begin{aligned}\forall(R \sqcup S)^*.C &\equiv \mu Z.C \sqcup \forall(R \sqcup S).Z\\ &\equiv \mu Z.C \sqcup \neg\exists(R \sqcup S).\neg Z\\ &\equiv \mu Z.C \sqcup \neg((\exists R.\neg Z) \sqcup (\exists S.\neg Z)).\end{aligned}$$

To express that x points to a binary tree, we augment the above TBox for DAGs with axioms that prohibit sharing:

$$\begin{aligned}\mathcal{T}_8 = \mathcal{T}_7 \cup \{&\leq\!1\, R^-_{\mathit{left}}.\top \doteq \neg I_{NULL},\ \ \leq\!1\, R^-_{\mathit{right}}.\top \doteq \neg I_{NULL},\\ &(\exists R^-_{\mathit{left}}.\top) \sqcap (\exists R^-_{\mathit{right}}.\top) \doteq I_{NULL}\}\end{aligned}$$

Note that balanced tree data structures such as AVL trees, 2-3 trees, B trees, red-black trees cannot be expressed in description logics, since their invariants involve comparing properties (typically the height) of different subtrees.

4. Symbolic Shape Analysis

In this section, we sketch a symbolic shape analysis based on *predicate abstraction* [9]. In terms of the abstract interpretation framework [5], the essential ingredients for a program analysis are the Galois connection between the concrete and the abstract domain and the state transformer (program semantics) on the concrete domain. Our concrete domain are graphs that describe pointer structures in the heap, and the semantics of programs that manipulate such pointer graphs locally (in each computation step) is straightforward.

In predicate abstraction, the Galois connection is uniquely determined by the abstract domain, which is a finite lattice of formulas constructed from a finite set of *abstraction predicates*. In our case, the abstraction predicates are concept formulas in a description logic that can express reachability, and the abstract domain is the Boolean algebra of abstraction predicates. Instead of computing the best abstract transformer in this domain, we follow the suggestion of [2, 19, 27] and apply an additional Cartesian abstraction, leading to a less precise transformer. Yet, this transformer is efficiently computable provided that entailment between conjunctions of abstraction predicates and weakest preconditions of abstraction predicates is decidable efficiently.

4.1. Programs

We introduce the pointer programs that we want to analyze and give their semantics. For simplicity, we abstract away from data and we do not handle dynamic memory allocation and deallocation. I. e., our programs are control flow graphs, whose edges are labeled by the following seven types of statements.

$\mathit{Stmt} ::=$	$y = e?$	*equality test*
$\mid$	$y \neq e?$	*disequality test*
$\mid$	$x \neq NULL!$	*non-NULL assertion*
$\mid$	$x.f!$	*non-dangling assertion*
$\mid$	$y := e$	*assignment*
$\mid$	$y := x.f$	*pointer field dereference*
$\mid$	$x.f := e$	*pointer field update*

Thereby, x and y denote pointer variables, e denotes a pointer variable or $NULL$, and $x.f$ means dereferencing x and selecting field f of the pointed-to heap cell. To keep the semantics simple, we assume that the statements $x.f := y$ and $y := x.f$ are never executed when x is $NULL$ or dangling, respectively; this can be ensured by preceding assertions $x \neq NULL!$ and $x.f!$. Figure 5 shows a simple example program (a piece of straight line code of 5 statements) for inserting a node into a cyclic list.

Let P be a program given as a control flow graph. Informally, a state of P is the contents of the program variables and the contents of the heap. We represent both by means of a relational first-order structure, which interprets program variables (and the special value $NULL$) by singleton sets and the heap by an edge-labeled graph with functional edge relations. Formally, P induces a relational vocabulary $\bar{I} \cup \bar{R}$ consisting of sets $\bar{I}$ resp. $\bar{R}$ of unary predicates I_e, where e is a program variable or $NULL$, and binary predicates R_f, where f is a pointer field. A *state* of P is a relational structure $\mathbf{A}$ over $\bar{I} \cup \bar{R}$ subject to the following restrictions:

1. The universe of $\mathbf{A}$ is finite.
2. For all $I_e \in \bar{I}$, the set $I_e^{\mathbf{A}}$ is a singleton.
3. For all $R_f \in \bar{R}$, the relation $R_f^{\mathbf{A}}$ is functional.
4. $NULL$ has no successors in $\mathbf{A}$, i. e., for all $R_f \in \bar{R}$, $\mathbf{A} \models \forall u, v(I_{NULL}(u) \Rightarrow \neg R_f(u, v))$.

Statements transform program states, so their semantics is a binary relation on the state space, in our case a binary relation on first-order structures. We extend the vocabulary with a primed copy $\bar{I}' \cup \bar{R}'$, where $\bar{I}' = \{I'_e \mid I_e \in \bar{I}\}$ and $\bar{R}' = \{R'_f \mid R_f \in \bar{R}\}$. The semantics of statements s can now be expressed by first-order formulas $[\![s]\!]$ over $\bar{I} \cup \bar{I}' \cup \bar{R} \cup \bar{R}'$. Note that we consider a relational structure $\mathbf{A}$ over $\bar{I} \cup \bar{I}' \cup \bar{R} \cup \bar{R}'$ to be a model of $[\![s]\!]$ only if it satisfies the above restrictions for program states both for the unprimed and for the primed vocabulary. Figure 2 shows the semantics of statements, where for subsets $X \subseteq \bar{I}$ resp. $Y \subseteq \bar{R}$ of unary and binary predicates, the conjuncts $\mathit{nochange}(X)$ and $\mathit{nochange}(Y)$ specify that the predicates in X resp. Y do not change, formally:

$$\mathit{nochange}(X) \equiv \bigwedge_{I_e \in X} \forall u \big(I'_e(u) \Leftrightarrow I_e(u)\big)$$
$$\mathit{nochange}(Y) \equiv \bigwedge_{R_f \in Y} \forall u, v \big(R'_f(u, v) \Leftrightarrow R_f(u, v)\big)$$

Note that $[\![x \neq \mathit{NULL}!]\!] = [\![x \neq \mathit{NULL}?]\!]$, so we may view non-NULL assertions as special cases of disequality tests.

4.2. Weakest Preconditions

For computing abstractions to Boolean programs, it is essential to logically express the weakest (liberal) precondition $\mathit{wlp}_s(S)$ of a set of states S w. r. t. a statement s, i. e., the set of program states that are transformed to states in S upon execution of the (deterministic) statement s. Figure 3 presents a characterization of $\mathit{wlp}_s(\varphi)$ in first-order logic, where φ is a formula (which may have free variables) representing the set of states S. This characterization, which is straightforward to derive from the semantics in figure 2, uses quantification over a unary predicate I as well as substitutions[1] $\varphi[\psi(\$)/I_e(\$)]$ and $\varphi[\xi(\$_1, \$_2)/R_f(\$_1, \$_2)]$ of unary and binary predicates by formulas; the counting quantifier $\exists^{=1}$ (exists exactly one) is just an abbreviation.

Note the three preconditions for dereference statements. The first one is derived from the first-order semantics in the standard way, whereas the other two are tailored to deriving weakest preconditions in description logics. For proving the three preconditions equivalent one has to assume that x is not dangling (which can be ensured by a preceding assertion $x.f!$) and take into account the semantic restrictions on states saying that I_x is a singleton and R_f functional. Large parts of the equivalence proofs can be carried out automatically by modern first-order theorem provers like SPASS [26].

Recall that we want to employ DL systems for reasoning about weakest preconditions of certain concept formulas in a DL. Therefore, we have to express the weakest precondition operator as a concept formula transformer. Note that due to the semantic restrictions on states, we restrict to DL formulas where all atomic concepts are nominals (modeling program variables or NULL) and all atomic roles are functional (modeling pointer fields).

Figure 4 presents a characterization of the weakest preconditions $\mathit{wlp}_s(C)$ of a concept expression C w. r. t. a statement s in a DL with nominals, functional roles, and a universal role[2] U. Additionally the DL may use fixpoint concepts, inverse roles, and certain number restrictions.

For tests, assertions and assignments, it is easy to see that the DL preconditions in figure 4 correspond to the FO preconditions in figure 3 under the standard translation of concept formulas into first-order logic. The same is true for the preconditions of dereference statements, because the second order quantification $\forall I (\exists^{=1} u\, I(u) \Rightarrow \dots)$ is equivalent to introducing a new nominal I. Note that the two alternative preconditions differ in that the first may introduce inverse roles whereas the second does not. For update statements, the weakest preconditions are computed by structural recursion over the concept formula C, modifying only those subformulas of C that contain the role R_f. Note that we only allow number restrictions on inverse roles (since all atomic roles are functional anyway) of the form $\geq 2\, R_f^-.D$, $\leq 1\, R_f^-.D$ and $=1\, R_f^-.D$; for higher numbers the weakest preconditions of update statements are no longer expressible as concept formulas in a simple way. Again, large parts of the proof of correspondence between the weakest preconditions in FO and DL can be done automatically by the theorem prover SPASS. Figure 5 shows some examples of weakest preconditions of non-trivial concept formulas.

Note that $\mathit{wlp}_s(C) \in \mathcal{ALCO}_f^U$ if $C \in \mathcal{ALCO}_f^U$. The same preservation holds for some more expressive DLs including the ones listed in table 2.

4.3. Abstract State Transformer

We fix a set $\mathcal{C} = \{C_1, \dots, C_n\}$ of concept formulas, which we call abstraction predicates. The abstract domain of our shape analysis is the free Boolean algebra generated by the formulas in $\mathcal{C}$. That is, an abstract state is a Boolean combination of concept formulas, which is again a concept formula. Similar to the Boolean heaps of [19], an abstract state thus represents a set of heap graphs. More formally, the concretization of an abstract state C is the set of all interpretations $\mathcal{I}$ such that $\mathcal{I} \models \top \sqsubseteq C$.

To represent abstract states succinctly, we use sets of bitvectors, where a (3-valued) bitvector $(b_1, \dots, b_n)$ is a vector of values $b_i \in \{0, 1, *\}$. Such a bitvector $(b_1, \dots, b_n)$ can be interpreted as a conjunction

[1] $\varphi[\psi(\$)/I_e(\$)]$ is obtained from φ by replacing all subformulas $I_e(u)$ with $\psi(u)$. $\varphi[\xi(\$_1, \$_2)/R_f(\$_1, \$_2)]$ is defined analogously.

[2] Some DLs implicitly possess a universal role; e. g., in some fixpoint logics it is expressible as a greatest fixpoint.

$$
\begin{aligned}
[\![y = e?]\!] &\equiv \forall u\big(I_y(u) \Leftrightarrow I_e(u)\big) \wedge \mathit{nochange}(\bar{I}) \wedge \mathit{nochange}(\bar{R}) \\
[\![y \neq e?]\!] &\equiv \neg\forall u\big(I_y(u) \Leftrightarrow I_e(u)\big) \wedge \mathit{nochange}(\bar{I}) \wedge \mathit{nochange}(\bar{R}) \\
[\![x \neq \mathit{NULL}!]\!] &\equiv \neg\forall u\big(I_x(u) \Leftrightarrow I_{\mathit{NULL}}(u)\big) \wedge \mathit{nochange}(\bar{I}) \wedge \mathit{nochange}(\bar{R}) \\
[\![x.f!]\!] &\equiv \forall u\big(I_x(u) \Rightarrow \exists v\, R_f(u,v)\big) \wedge \mathit{nochange}(\bar{I}) \wedge \mathit{nochange}(\bar{R}) \\
[\![y := e]\!] &\equiv \forall u\big(I'_y(u) \Leftrightarrow I_e(u)\big) \wedge \mathit{nochange}(\bar{I} \setminus \{I_y\}) \wedge \mathit{nochange}(\bar{R}) \\
[\![y := x.f]\!] &\equiv \forall v\big(I'_y(v) \Leftrightarrow \exists u(I_x(u) \wedge R_f(u,v))\big) \wedge \mathit{nochange}(\bar{I} \setminus \{I_y\}) \wedge \mathit{nochange}(\bar{R}) \\
[\![x.f := e]\!] &\equiv \forall u,v\big(R'_f(u,v) \Leftrightarrow I_x(u) \wedge I_e(v) \vee \neg I_x(u) \wedge R_f(u,v)\big) \wedge \mathit{nochange}(\bar{I}) \wedge \mathit{nochange}(\bar{R} \setminus \{R_f\})
\end{aligned}
$$

Figure 2. First-order logic semantics of statements.

$$
\begin{aligned}
\mathit{wlp}_{y=e?}(\varphi) &\equiv \forall u\big(I_y(u) \Leftrightarrow I_e(u)\big) \wedge \varphi \\
\mathit{wlp}_{y\neq e?}(\varphi) &\equiv \neg\forall u\big(I_y(u) \Leftrightarrow I_e(u)\big) \wedge \varphi \\
\mathit{wlp}_{x.f!}(\varphi) &\equiv \forall u\big(I_x(u) \Rightarrow \exists v\, R_f(u,v)\big) \wedge \varphi \\
\mathit{wlp}_{y:=e}(\varphi) &\equiv \varphi[I_e(\$)/I_y(\$)] \\
\mathit{wlp}_{y:=x.f}(\varphi) &\equiv \exists I\big(\forall v\big(I(v) \Leftrightarrow \exists u(I_x(u) \wedge R_f(u,v))\big) \wedge \varphi[I(\$)/I_y(\$)]\big) \\
&\equiv \forall I\big(\exists^{=1} u\, I(u) \Rightarrow \forall v\big(I(v) \Leftrightarrow \exists u(R_f(u,v) \wedge I_x(u))\big) \Rightarrow \varphi[I(\$)/I_y(\$)]\big) \\
&\equiv \forall I\big(\exists^{=1} u\, I(u) \Rightarrow \forall u\big(I_x(u) \Leftrightarrow \exists v(R_f(u,v) \wedge I(v))\big) \Rightarrow \varphi[I(\$)/I_y(\$)]\big) \\
\mathit{wlp}_{x.f:=e}(\varphi) &\equiv \varphi[I_x(\$_1) \wedge I_e(\$_2) \vee \neg I_x(\$_1) \wedge R_f(\$_1,\$_2)/R_f(\$_1,\$_2)]
\end{aligned}
$$

Figure 3. Weakest preconditions in first-order logic.

$$
\begin{aligned}
\mathit{wlp}_{y=e?}(C) &\equiv \forall U.(I_y \Leftrightarrow I_e) \sqcap C \\
\mathit{wlp}_{y\neq e?}(C) &\equiv \neg\forall U.(I_y \Leftrightarrow I_e) \sqcap C \\
\mathit{wlp}_{x.f!}(C) &\equiv \forall U.(I_x \Rightarrow \exists R_f.\top) \sqcap C \\
\mathit{wlp}_{y:=e}(C) &\equiv C[I_e/I_y] \\
\mathit{wlp}_{y:=x.f}(C) &\equiv \forall U.(I \Leftrightarrow \exists R_f^-.I_x) \Rightarrow C[I/I_y] \qquad \textit{(I new nominal)} \\
&\equiv \forall U.(I_x \Leftrightarrow \exists R_f.I) \Rightarrow C[I/I_y]
\end{aligned}
$$

$$
\mathit{wlp}_{x.f:=e}(C) \equiv
\begin{cases}
\big(I_x \sqcap \exists U.(I_e \sqcap \mathit{wlp}_{x.f:=e}(D))\big) \sqcup \big(\neg I_x \sqcap \exists R_f.\mathit{wlp}_{x.f:=e}(D)\big) & \text{if } C \equiv \exists R_f.D \\
\mathit{wlp}_{x.f:=e}(\neg\exists R_f.\neg D) & \text{if } C \equiv \forall R_f.D \\
\big(I_e \sqcap \exists U.(I_x \sqcap \mathit{wlp}_{x.f:=e}(D))\big) \sqcup \exists R_f^-.(\neg I_x \sqcap \mathit{wlp}_{x.f:=e}(D)) & \text{if } C \equiv \exists R_f^-.D \\
\mathit{wlp}_{x.f:=e}(\neg\exists R_f^-.\neg D) & \text{if } C \equiv \forall R_f^-.D \\
\big(I_e \sqcap \exists U.(I_x \sqcap \mathit{wlp}_{x.f:=e}(D)) \sqcap \exists R_f^-.(\neg I_x \sqcap \mathit{wlp}_{x.f:=e}(D))\big) & \\
\qquad \sqcup \geq\!2\, R_f^-.(\neg I_x \sqcap \mathit{wlp}_{x.f:=e}(D)) & \text{if } C \equiv \geq\!2\, R_f^-.D \\
\mathit{wlp}_{x.f:=e}(\neg \geq\!2\, R_f^-.D) & \text{if } C \equiv \leq\!1\, R_f^-.D \\
\mathit{wlp}_{x.f:=e}((\exists R_f^-.D) \sqcap (\leq\!1\, R_f^-.D)) & \text{if } C \equiv =\!1\, R_f^-.D \\
\text{structural recursion over } C & \text{otherwise}
\end{cases}
$$

Figure 4. Weakest preconditions in description logic.

$\mathcal{C}(b_1, \ldots, b_n)$ of (possibly negated) predicates from $\mathcal{C}$. Formally, a predicate C_i

- occurs positively in $\mathcal{C}(b_1, \ldots, b_n)$ iff $b_i = 1$,
- occurs negatively in $\mathcal{C}(b_1, \ldots, b_n)$ iff $b_i = 0$ and
- does not occur in $\mathcal{C}(b_1, \ldots, b_n)$ iff $b_i = *$.

A set V of bitvectors can be interpreted as the disjunction of conjunctions $\mathcal{C}(b_1, \ldots, b_n)$, for all $(b_1, \ldots, b_n) \in V$. As abstract states are Boolean combinations of abstraction predicates, which can always be brought into disjunction normal form, every abstract state can be represented by a set of bitvectors. We end up with an abstract domain of size doubly exponential and height singly exponential in n.

To compute the abstract state transformer $post^\#_s(C)$ of an abstract state C w. r. t. a statement s, we assume that C is given as a set of bitvectors V. Since $post^\#$ distributes over disjunctions, it suffices to compute $post^\#_s(\mathcal{C}(b_1, \ldots, b_n))$ for each bitvector $(b_1, \ldots, b_n) \in V$. However, instead of computing the best $post^\#_s(\mathcal{C}(b_1, \ldots, b_n))$ we follow [2, 19, 27]. i. e., we express the state transformer in terms of the weakest preconditions (which is possible since the statements s are deterministic) and then apply an additional Cartesian abstraction. This leads to a less precise transformer, but can be computed by checking a linear number of entailments. More precisely, it ensures that the result of $post^\#_s(\mathcal{C}(b_1, \ldots, b_n))$ is a conjunction $\mathcal{C}(b'_1, \ldots, b'_n)$ represented by a bitvector $(b'_1, \ldots, b'_n)$, where

$$b'_i = \begin{cases} 1 & \text{if } \mathcal{C}(b_1, \ldots, b_n) \text{ entails } wlp_s(C_i) \\ 0 & \text{if } \mathcal{C}(b_1, \ldots, b_n) \text{ entails } wlp_s(\neg C_i) \\ * & \text{otherwise} \end{cases}$$

In our case, where abstraction predicates are concept formulas, checking entailment means checking concept inclusion w. r. t. to finite interpretations. Formally, the entailment checks are queries of the form

$$\mathcal{T} \models_{\text{fin}} \mathcal{C}(b_1, \ldots, b_n) \sqsubseteq wlp_s([\neg]C_i) \ .$$

Thereby, $\mathcal{T}$ is a TBox asserting (at least) axioms of the form $\top \sqsubseteq {\leq}1\, R_f.\top$, ensuring that R_f is a functional relation, and $NULL \sqsubseteq \forall R_f.\bot$, ensuring that $NULL$ does not have successors. Moreover, Wies [27] suggests to increase the precision of the analysis by checking entailment relative to the current abstract state C, which amounts to adding the inclusion $\top \sqsubseteq C$ to $\mathcal{T}$. Section 4.5 will show how the analysis works on an example.

4.4. Abstraction Predicates

The properties of a predicate abstraction based analysis, in terms of usefulness of the generated invariants, precision

DL	abs. predicates	purpose
$\mathcal{ALCO}_f^U$	$\top$	all nodes
	I_{NULL}, I_x	prog. var. nodes
	$\neg C$, $C_1 \sqcup C_2$	boolean comb.
	$\exists R_f.C$	direct predecessors
$\mathcal{ALCO}_f^{-1,U}$	$\exists R_f^-.C$	direct successors
$\mathcal{ALCQO}^{-1,U}$	${\geq}2\, R_f^-.C$	sharing nodes
$\mu\mathcal{ALCO}_f$	$\exists R_f^*.C$	predecessors
$\mu\mathcal{ALCO}_f^{-1}$	$\exists(R_f^-)^*.C$	successors
	$\nu Z.(\exists R_f^-.C \sqcap Z)$	cyclic nodes

Table 3. Abstraction predicates.

and cost, crucially depend on the class of abstraction predicates. If we stay within the logic $\mathcal{ALCO}_f^U$, only very simple abstraction predicates are expressible; more interesting predicates (in particular, for reachability) require more expressive logics. Table 3 shows which common abstraction predicates can be constructed in which logic. Note that the transitive closure operators on roles are abbreviations here, which can be eliminated in favor of fixpoints since $\exists R^*.C \equiv \mu Z.(C \sqcup \exists R.Z)$ and $\exists R^+.C \equiv \exists R.(\exists R^*.C)$. As the table shows, depending on the logic we can construct predicates expressing

- sharing like ${\geq}2\, R_f^-.\top$: all nodes with at least 2 incoming pointers,
- reachability and cyclicity like $\nu Z.(\exists R_f.Z)$: all nodes that can reach a cycle, and
- separation like $\neg(\exists(R_f^-)^*.I_x \sqcap \exists(R_f^-)^*.I_y)$: all nodes that are not jointly reachable from both x and y.

4.5. Example

Figure 5 depicts the results of the shape analysis on an example program for inserting a new node (pointed by e) into a singly-linked cyclic list (pointed by x). The example uses 9 abstraction predicates, C_1 for $NULL$, C_2 to C_4 for the cells pointed by the program variables e, x and t, C_5 for cells which have a successor, C_6 and C_7 for the successors of the cells pointed by x and e, C_8 for cells that can reach x, and C_9 for cells that are on a cycle and can reach x.

The initial set of bitvectors V_1 encodes a precondition for the program: x points to a cyclic list of at least two elements (the cell pointed by x and its successor), and e points to a cell whose successor is $NULL$; in particular e is not on the list. The analysis computes[3] the sets V_2 to V_4. From the final set V_4, we can read off the postcondition of the program: x points to a cyclic list, which includes e.

[3]By hand, for lack of a reasoner for the DL $\mu\mathcal{ALCO}_f^{-1}$.

set of abstraction predicates $\mathcal{C} = \{C_1, \ldots, C_9\}$								
C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
I_{NULL}	I_e	I_x	I_t	$\exists R_{next}.\top$	$\exists R^-_{next}.I_x$	$\exists R^-_{next}.I_e$	$\mu Y.I_x \sqcup \exists R_{next}.Y$	$\nu Z.\exists R^-_{next}.C_8 \sqcap Z$

statements	sets of bitvectors	vectors correspond to
	$V_1 = \{(1,0,0,*,0,0,1,0,0),$	$NULL$ (= successor of e)
	$(0,1,0,*,1,0,0,0,0),$	e
	$(0,0,1,*,1,0,0,1,1),$	x
	$(0,0,0,*,1,1,0,1,1),$	successor of x
	$(0,0,0,*,1,0,0,1,1),$	other nodes on the list
	$(0,0,0,*,*,0,0,*,0)\}$	other nodes in the heap
$x.next!$ $t := x.next$		
	$V_2 = \{(1,0,0,0,0,0,1,0,0),$	$NULL$ (= successor of e)
	$(0,1,0,0,1,0,0,0,0),$	e
	$(0,0,1,0,1,0,0,1,1),$	x
	$(0,0,0,1,1,1,0,1,1),$	t (= successor of x)
	$(0,0,0,0,1,0,0,1,1),$	other nodes on the list
	$(0,0,0,0,*,0,0,*,0)\}$	other nodes in the heap
$e \neq NULL!$ $e.next := t$		
	$V_3 = \{(1,0,0,0,0,0,0,0,0),$	$NULL$
	$(0,1,0,0,1,0,0,1,0),$	e
	$(0,0,1,0,1,0,0,1,1),$	x
	$(0,0,0,1,1,1,1,1,1),$	t (= successor of x and e)
	$(0,0,0,0,1,0,0,1,1),$	other nodes on the list
	$(0,0,0,0,*,0,0,*,0)\}$	other nodes in the heap
$x.next := e$		
	$V_4 = \{(1,0,0,0,0,0,0,0,0),$	$NULL$
	$(0,1,0,0,1,1,0,1,1),$	e (= successor of x)
	$(0,0,1,0,1,0,0,1,1),$	x
	$(0,0,0,1,1,0,1,1,1),$	t (= successor of e)
	$(0,0,0,0,1,0,0,1,1),$	other nodes on the list
	$(0,0,0,0,*,0,0,*,0)\}$	other nodes in the heap

Sample queries generated during the computation of the abstract post (TBox $\mathcal{T}_i$ corresponds to bitvector set V_i):

$$\mathcal{T}_1 \models \mathcal{C}(0,0,0,*,1,1,0,1,1) \sqsubseteq wlp_{x.next!;t:=x.next}(C_4)$$
$$\mathcal{T}_3 \models \mathcal{C}(0,1,0,0,1,0,0,1,0) \sqsubseteq wlp_{x.next:=e}(C_8)$$
$$\mathcal{T}_3 \models \mathcal{C}(0,1,0,0,1,0,0,1,0) \sqsubseteq wlp_{x.next:=e}(C_9)$$

Sample weakest preconditions:

$$\begin{aligned}
&wlp_{x.next!;x.next:=e}(C_4) \equiv wlp_{x.next!}(wlp_{x.next:=e}(I_t))\\
&\quad \equiv wlp_{x.next!}(\forall U.I \Leftrightarrow \exists R^-_{next}.I_x) \Rightarrow I)\\
&\quad \equiv (\forall U.I_x \Rightarrow \exists R_{next}.\top) \sqcap ((\forall U.I \Leftrightarrow \exists R^-_{next}.I_x) \Rightarrow I)\\
&wlp_{x.next:=e}(C_8) \equiv wlp_{x.next:=e}(\mu Y.I_x \sqcup \exists R_{next}.Y)\\
&\quad \equiv \mu Y.I_x \sqcup I_x \sqcap (\exists U.I_e \sqcap Y) \sqcup \neg I_x \sqcap \exists R_{next}.Y\\
&\quad \equiv \mu Y.I_x \sqcup \neg I_x \sqcap \exists R_{next}.Y\\
&wlp_{x.next:=e}(C_9) \equiv wlp_{x.next:=e}(\nu Z.\exists R^-_{next}.C_8 \sqcap Z)\\
&\quad \equiv \nu Z.I_e \sqcap (\exists U.I_x \sqcap (\mu Y.I_x \sqcup \neg I_x \sqcap \exists R_{next}.Y) \sqcap Z) \sqcup\\
&\qquad\quad (\exists R^-_{next}.\neg I_x \sqcap (\mu Y.I_x \sqcup \neg I_x \sqcap \exists R_{next}.Y) \sqcap Z)
\end{aligned}$$

Sample TBox: $\mathcal{T}_1 = \{I_{NULL} \sqsubseteq \forall R_{next}.\bot,$

$$\begin{aligned}
\top &\sqsubseteq {\leq}1\, R_{next}.\top,\\
\top &\sqsubseteq \mathcal{C}(1,0,0,*,0,0,1,0,0) \sqcup\\
&\quad\ \mathcal{C}(0,1,0,*,1,0,0,0,0) \sqcup\\
&\quad\ \mathcal{C}(0,0,1,*,1,0,0,1,1) \sqcup\\
&\quad\ \mathcal{C}(0,0,0,*,1,1,0,1,1) \sqcup\\
&\quad\ \mathcal{C}(0,0,0,*,1,0,0,1,1) \sqcup\\
&\quad\ \mathcal{C}(0,0,0,*,*,0,0,*,0)\}
\end{aligned}$$

Figure 5. Example: Inserting a new element into a singly-linked cyclic list.

4.6. Approximating Entailment

In the previous sections, we did advocate the use of very expressive description logics in shape analysis. There are a number of reasons why we may have to settle for less:

Decidability. Alternation-free $\mu\mathcal{ALCO}_f^{-1}$ was shown undecidable in [3]; decidability of $\mu\mathcal{ALCO}_f$ is not known.

Finite decidability. Actually, we want decidability in finite models, which does not necessarily coincide with decidability, as many expressive description logics (e. g., $\mu\mathcal{ALCO}_f^{-1}$) lack the finite model property. Finite decidability of $\mu\mathcal{ALCO}_f^{-1}$ and $\mu\mathcal{ALCO}_f$ is not known, however, $\mathcal{ALCO}_f^U$ and $\mathcal{ALCQO}^{-1,U}$ are finitely decidable via embedding into C^2, the 2-variable fragment of first-order logic with counting quantifiers, for which finite decidability is in NEXPTIME [20].

Availability. Currently, there are no systems that handle fixpoints or at least transitive closure, so the 'reachability' logics $\mu\mathcal{ALCO}_f$ and $\mu\mathcal{ALCO}_f^{-1}$ are beyond today's reach. $\mathcal{ALCO}_f^U$ and $\mathcal{ALCQO}^{-1,U}$ can be handled by existing systems (e. g., by FaCT [12] as they are fragments of $\mathcal{SHOQ}$ and $\mathcal{SHIOQ}$, respectively).

Performance. Computing abstractions in one shape analysis may trigger thousands of entailment tests. Even if they exist, systems for fixpoint logics may not be perform well enough to be of productive use.

There are different ways to relax these problems. If the system at hand implements a calculus for an undecidable logic, non-terminating computations may be aborted by timeouts, leading to a potential loss of precision in the analysis. The same technique may help to mitigate (sometimes) poor performance.

If the logic does not have the finite model property and no finite model reasoner is available, one may use an unrestricted model reasoner. The latter may prove less entailments, leading to a loss of precision, but all entailments proven are sound, i. e., continue to hold in finite models.

If systems for expressive logics are not available, not fast or not decidable enough, then one may also approximate using weaker logics, i. e., logics with less restrictions on the class of models. For instance, from the undecidable logic $\mu\mathcal{ALCO}_f^{-1}$ one can either drop the functional roles and obtain the EXPTIME-complete DL $\mu\mathcal{ALCO}^{-1}$ [23], or one can drop the nominals and obtain a fragment of the EXPTIME-complete DL $\mu\mathcal{ALCQ}^{-1}$ [4]. In both cases, the approximations are sound, i. e., if an entailment holds in the weaker logic it will continue to hold in the stronger one.

5. Related Work

In this section we consider symbolic languages for shape analysis among which are local shape logic [21], role logic [16], and 3-valued logic [22], and discuss their relationship with description logics.

3-valued logic. In the framework for parametric 3-valued shape analysis [22], concrete stores are represented by 2-valued logical structures. First-order logic with transitive closure, which is in general undecidable, is then used to express properties of stores such as reachability, acyclicity, sharing. The logic is interpreted over 3-valued structures, according to Kleene's 3-valued interpretation of first-order logic. The use of instrumentation predicates in 3-valued shape analysis is similar to our use of abstraction predicates (which is inspired by [19, 27]).

Local shape logic and role logic. Local shape logic (LSL) [21] is introduced in an attempt to describe shape graphs, e. g., linked lists, circular buffers. Similar to description logics, formulas in LSL are interpreted over labeled directed graphs, where nodes are locations of objects and edges represent references. LSL is a decidable fragment of the undecidable shape logic [21], which is a typed first-order logic. Decidability is gained by restricting the use of first-order variables; more precisely, LSL can be strictly embedded into C^2, the 2-variable fragment of first-order logic with counting quantifiers.

Role logic RL is variable free logic which is equally expressive as first-order logic with transitive closure and consequently undecidable. A subset of role logic, RL^2, is decidable and equally expressive as C^2. Thus, RL^2 is decidable and as expressive as the most expressive DLs without role composition, transitive closure and fixpoints.

A limitation of LSL and RL^2 is their expressive power; in contrast to DLs with fixpoints, neither logic can model crucial properties such as reachability and separation.

Decidable extensions of first-order fragments. An alternative approach to symbolic shape analysis [19, 27] uses decidable extensions of fragments of first-order logic, e. g., guarded fixpoint logic μGF [8]. In μGF, one can express reachability from specified points along specified paths, but full transitive closure (i. e., reachability between a pair of variables) is inexpressible. Moreover, μGF lacks the finite model property [8] and becomes undecidable when functionality restrictions are added [7].

Monadic second-order logic. An approach described in [18] uses a special pointer assertion logic for the verification of user-supplied data structure invariants. The logic is translated to formulae in monadic second order logic over trees, which are tested for satisfiability with the MONA system [11]. Note that this approach can only handle data structures which implicitly have an underlying tree structure; in particular, DAGs cannot be handled.

6. Conclusion

In this paper, we studied expressive description logics as a framework for shape analysis. We expressed properties of standard data structures, such as lists (including singly and doubly linked lists and cyclic lists) DAGs and trees, in DLs augmented with fixpoints. We sketched a symbolic shape analysis based on predicate abstraction, parameterized by description logics to represent the abstraction predicates. Depending on the chosen logic sharing, reachability and separation in pointer data structures are expressible.

Description logics have features which make them attractive for shape analysis. They are expressive languages which allow for the representation of complex descriptions of data structures. Formal properties of DLs are extensively studied and well-understood. Tools for reasoning with expressive and even undecidable description logics exist, e. g., FaCT [12], Racer [10].

To implement our analysis, we need reasoners that can handle DLs with fixpoints (or transitive closure), number restrictions, inverse roles and nominals. The existing tools already support some of these constructs efficiently, yet support for fixpoints and nominals is lacking. Comparing this situation to the state of the art in first-order theorem proving, neither transitive closure nor fixpoints are supported. In the light of the decidability results for some DLs with fixpoints, we believe that there is more hope for a well-behaved DL reasoner with fixpoints or transitive closure than for a first-order theorem prover with the corresponding features.

Acknowledgements. We thank Thomas Wies and the anonymous referees for discussions and suggestions.

References

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[2] T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proc. TACAS'01*, LNCS 2031, pages 268–283. Springer, 2001.

[3] P. Bonatti and A. Peron. On the undecidability of logics with converse, nominals, recursion and counting. *Artifi cial Intelligence*, 158:75–96, 2004.

[4] D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fi xpoints based on automata on infi nite trees. In *Proc. IJCAI'99*, pages 84–89. Morgan Kaufmann, 1999.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unifi ed lattice model for static analysis by construction of approximation of fi xed points. In *Proc. POPL'77*, pages 238–252. ACM Press, 1977.

[6] P. Fradet and D. Metayer. Shape types. In *Proc. POPL'97*, pages 27–39. ACM Press, 1997.

[7] E. Gr¨adel. On the restraining power of guards. *J. Symbolic Logic*, 64:1719–1742, 1999.

[8] E. Gr¨adel and I. Walukiewicz. Guarded fi xed point logic. In *Proc. LICS'99*, pages 45–54. IEEE Computer Society Press, 1999.

[9] S. Graf and H. Saidi. Construction of abstract shape graphs with PVS. In *Proc. CAV'97*, LNCS 1254, pages 72–83. Springer, 1997.

[10] V. Haarslev and R. M¨oller. RACER system description. In *Proc. IJCAR'01*, LNCS 2083, pages 29–44. Springer, 2001.

[11] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS'95*, LNCS 1019, pages 89–110, 1995.

[12] I. Horrocks. Using an expressive description logic: FaCT or fi ction? In *Proc. KR'98*, pages 636–647. Morgan Kaufman, 1998.

[13] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Proc. CSL'04*, LNCS 3210, pages 160–174. Springer, 2004.

[14] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verifi cation via structure simulation. In *Proc. CAV'04*, LNCS 3114, pages 281–294. Springer, 2004.

[15] D. Kozen. Results on the propositional μ-calculus. In *Proc. ICALP'82*, LNCS 140, pages 348–359. Springer, 1982.

[16] V. Kuncak and M. Rinard. On role logic. Technical Report 925, MIT Computer Science and Artifi cial Intelligence Laboratory, 2003.

[17] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Proc. SAS'00*, LNCS 1824, pages 280–301. Springer, 2000.

[18] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. PLDI'01*, pages 221–231, 2001.

[19] A. Podelski and T. Wies. Boolean heaps. In *Proc. SAS'05*, 2005. To appear.

[20] I. Pratt-Hartmann. Complexity of the two-variable fragment with counting quantifi ers. *J. Logic, Language and Information*. To appear.

[21] A. Rensink. Canonical graph shapes. In *Proc. ESOP'04*, LNCS 2986, pages 401–415. Springer, 2004.

[22] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24:217–298, 2002.

[23] U. Sattler and M. Y. Vardi. The hybrid μ-calculus. In *Proc. IJCAR'01*, LNAI 2083, pages 76–91. Springer, 2001.

[24] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *J. Artifi cial Intelligence*, 48:1–26, 1991.

[25] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional μ-calculus. *Information and Computation*, 81:249–264, 1989.

[26] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In *Proc. CADE'02*, LNCS 2392, pages 275–279. Springer, 2002.

[27] T. Wies. Symbolic shape analysis. Master's thesis, MPI Informatik, Saarbr¨ucken, Germany, 2004.

[28] G. Yorsh, T. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proc. TACAS'04*, LNCS 2988, pages 530–545. Springer, 2004.