Symbolic Shape Analysis

Diploma Thesis

Thomas Wies

September 2004

advised by
Prof. Dr. Andreas Podelski

Universität des Saarlandes
Fachbereich 6.2 - Informatik

Erklärung

Hiermit erkläre ich, Thomas Wies, an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, den 30.09.2004

Acknowledgments

First of all, I thank my advisor Prof. Andreas Podelski for the interesting and challenging thesis subject, the inspiring discussions, his support, and his patience. I would like to thank Viktor Kuncak for discussions that initiated the work on this thesis. I am also indebted to Dr. Mooly Sagiv and Greta Yorsh for giving helpful comments while on stay at Tel-Aviv in spring 2004. I have to thank everyone that contributed to the thesis by giving hints and suggestions for improvements. I am particularly grateful to Ina Schaefer and Andrey Rybalchenko for valuable debates and proofreading several draft versions of this thesis. Special thanks go to the members of AG2 at MPII for providing such a nice and productive working atmosphere. Finally, I thank my family for their constant support and encouragement that only made this thesis possible.

Abstract

Shape analysis deals with the synthesis of invariants for programs manipulating heap-allocated data structures. Explicit shape analysis algorithms do not scale very well. This work proposes a framework for symbolic shape analysis that addresses this problem. Our contribution is a framework that allows to abstract programs with heap-allocated data symbolically by Boolean programs. For this purpose, we combine abstraction techniques from shape analysis with ideas from predicate abstraction. Our framework is parameterized by a set of abstraction predicates. We propose a class of predicates that can be used to analyze reachability properties for linked data structures. This class may potentially be used for automated abstraction refinement.

Contents

Chapter 1

Introduction

Shape analysis deals with programs manipulating heap allocated data structures. Explicit shape analysis algorithms do not scale very well. One direction to address scalability is to use symbolic methods. This work proposes a framework for symbolic shape analysis.

1.1 Motivation

Invariants synthesized by program analysis are over-approximations of the set of reachable program states. They are useful in a variety of applications, ranging from instruction scheduling and code optimization in compilers to formal verification.

Most program analysis problems for infinite state systems are undecidable. Therefore, one needs techniques for approximation. *Abstract interpretation* [7] is a framework that formalizes approximation for program analysis. In order to compute invariants, the program is interpreted over an abstract domain of abstract states. The creative act in applying abstract interpretation lies in the identification of a suitable abstract domain.

Shape analysis aims for the synthesis of invariants for programs manipulating heap-allocated data structures. In order to model the state of a program with heap-allocated data, one has to model the state of the heap. The heap can be represented as a graph, where the nodes correspond to allocated objects and edge relations that reflect how objects are connected via pointers. Each pointer field of a stored data structure corresponds to one edge relation of the graph. Since *a priori* there is no bound on the number of allocated heap objects, these graphs can be arbitrary large. Consequently, the number of possible program states is unbounded and therefore we need abstraction.

Abstract domains used for shape analysis are based on several variants of *shape graphs* [17, 5, 29, 26, 27]. The common idea of these abstractions is to partition the set of nodes in a concrete graph into a finite set of equivalence classes. These equivalence classes are represented by abstract nodes in a shape graph. An equivalence class contains all nodes that are indistinguishable under a particular set of shape properties, e.g. expressing that a node is pointed to by a program variable, or that it is reachable from a program variable by following pointer fields in a data structure. In [27] a parametric framework for shape analysis is proposed that identifies shape properties as unary abstraction predicates that denote sets of nodes.

Although the use of abstract domains based on shape graphs gives approximating algorithms, these algorithms have high complexity, because the abstract domain is though finite, still very large. If one uses an explicit representation of abstract states, such as shape graphs, the analysis does not scale very well. One way to address this problem is to use symbolic methods.

In the following, we discuss one symbolic method in more details. *Predicate abstraction* (see e.g. [12]) is a technique used to symbolically abstract infinite state systems. It is successfully applied in software model checkers such as SLAM [3] and BLAST [13]. The abstract domain is given by a finite class of formulas built from a chosen set of abstraction predicates. These abstraction predicates denote sets of program states. The program is abstracted symbolically by a Boolean program whose program variables correspond to the abstraction predicates [1]. One of the merits of predicate abstraction is that the analysis of the obtained abstract Boolean program can be implemented efficiently using a symbolic representation of abstract states, e.g. based on BDDs.

The computed invariants can be used to verify whether a given property is satisfied by the program. If the invariants are too weak to entail the target property, the abstract domain has to be refined, in order to obtain stronger invariants. Abstraction refinement is a key technology for fully automated formal verification.

For abstract domains that are parameterized by a set of abstraction predicates, refinement amounts to adding additional abstraction predicates. For predicate abstraction there are successful automated abstraction refinement procedures; see e.g. [6, 14, 2]. However, in shape analysis there are, so far, only heuristics that help to find additional predicates; see e.g. *instrumentation predicates* proposed in [27]. The question whether these predicates can be derived automatically is still open.

This work proposes a framework for symbolic shape analysis. We give a symbolic abstract domain that allows to handle abstract programs symbolically. The framework enables the integration of automated abstraction refinement procedures. However, contrary to predicate abstraction, the abstract domain is parameterized by unary predicates that denote sets of nodes in the abstracted graphs. This conforms to the abstract domains for shape analysis that are based on shape graphs.

1.2 Contributions

Our contribution is a framework for symbolic shape analysis:

- We propose a symbolic abstract domain that allows to represent abstract programs symbolically. We call the obtained abstract programs *Boolean heap programs*.
- The construction of Boolean heap programs requires the identification of a suitable set of unary abstraction predicates. We propose *modal node predicates* that can be used for the analysis of reachability properties for linked data structures.
- We give preliminary results that may be useful for the development of abstraction refinement procedures for properties expressed by modal node predicates.

In the following, we give a more detailed discussion of these contributions.

Boolean Heap Programs. We propose a symbolic abstract domain for shape analysis. This abstract domain is parameterized by a set of unary abstraction predicates. In analogy to predicate abstraction, the concrete program is abstracted by a Boolean program. We call these abstract programs *Boolean heap programs*. A Boolean heap program is formally characterized in terms of an over-approximation of the best abstract post operator on the chosen abstract domain. This guarantees soundness of the resulting analysis. In addition, we examine under which conditions a Boolean heap program coincides with the best abstract post operator.

Modal Node Predicates. We identify the class of *modal node predicates*, which are unary abstraction predicates that express reachability properties for linked data structures. We give a first result regarding decidability of the satisfiability problem for modal node predicates. Moreover, we show that modal node predicates are closed under weakest liberal preconditions. These results may be useful for the development of an abstraction refinement procedure for properties expressed by modal node predicates.

1.3 Outline

The remainder of the thesis is organized as follows:

Chapter 2 gives the preliminaries. We first briefly introduce the basic notions of abstract interpretation (2.1). Afterwards, we revise the definition of first-order logic with transitive closure (2.2) which is then used to formally describe the concrete semantics of programs manipulating heap allocated data structures (2.3). This gives the formal foundation of our abstract interpretation based symbolic shape analysis.

In **Chapter 3**, we develop our abstraction framework. We give a symbolic abstract domain that is parameterized by a set of unary abstraction predicates (3.1). Thereafter, we show how the best abstract post operator on this abstract domain is approximated by an abstract post operator that corresponds to a Boolean program (3.2.1-3.2.4). Moreover, we analyze under which conditions the Boolean program coincides with the best abstract post (3.2.5). Finally, we propose a method that can be applied, in order to gain back precision in the case that the Boolean program gives imprecise results with respect to the best abstract post (3.2.6).

Chapter 4 introduces modal node predicates (4.1,4.2), a class of unary predicates that can be used to instantiate the framework presented in Chapter 3. We give first results regarding decidability of the satisfiability problem (4.3) and show that modal node predicates are closed under weakest liberal preconditions (4.4).

In **Chapter 5** the developed framework is illustrated in a case study. We verify a program that reverses singly-linked lists. The analysis uses modal node predicates. The steps of the corresponding fixed point iteration are given in **Appendix B**.

Chapter 6 compares the presented framework to related work. We discuss shape analysis algorithms that are based on shape graph abstraction (6.1), consider other approaches that apply symbolic methods in shape analysis (6.2), and discuss recent results on decidable logics for shape analysis (6.3).

In **Chapter 7** we conclude and discuss open problems for future work.

In **Appendix A** one can find the proofs of all statements made in this work.

Chapter 2

Preliminaries

This chapter gives a brief introduction into abstract interpretation based on Galois connections. Abstract interpretation is a formal, semantics-based framework for the systematic construction of sound static program analyses. Our approach presented in the later chapters conforms to this framework. We further revise the definition of first-order logic with transitive closure. This logic allows one to formally define the concrete representation of program stores and serves as a specification language for the properties we want to analyze.

2.1 Abstract Interpretation

The goal of a program analysis is to collect information over a program's runtime behavior for all possible input data. Running a program on all its inputs is usually either impossible, because the number of input values is infinite, or infeasible, because it is finite, but too large to be handled by exhaustive testing. Static analysis infers information over all possible program executions without executing the program explicitly.

Since nearly every interesting problem concerning a program's runtime behavior is undecidable, the use of approximation is inevitable to ensure termination of the analysis. The goal is thus to develop incomplete, but automatic methods that produce reliable results. Abstract interpretation gives a formal, semantics-based framework for the systematic construction of such approximations.

2.1.1 Transition Systems and Transformer Functions

In order to be able to formally reason about a system, it is crucial to have a formal semantics of the system behavior. Transition systems offer the most simple and unified approach to describe the semantics of a broad range of systems, in particular (concurrent) imperative programs. The common semantic properties of transition systems are well understood. In the following, we introduce the basic notions and summarize some of the more prominent properties that will become useful, later on. A detailed discussion of transition systems, transformer functions and their properties can be found for instance in [28].

Definition 2.1.1 (Transition System). *A **transition system** S is a tuple $\langle State, \mathsf{init}, R\rangle$ with:*

- *$State$: the (possibly infinite) set of states of the system,*
- *$\mathsf{init} \subseteq State$: the set of initial states,*

- *$R \subseteq State \times State$: the transition relation.*

Given a transition system $\mathcal{S}$, the properties of the transition relation R can be elegantly described in terms of the associated transformer functions. The *post operator* maps sets of states to the set of all successor states under R, the *pre operator* maps a set of states to the set of all predecessors under R, and the dual of the pre operator, maps a set of states to its *weakest liberal precondition* (wlp).

Definition 2.1.2 (Transformer Functions).

$$\begin{aligned}
\mathsf{post} &\overset{def}{=} \lambda\, S \,.\, \{\, s' \in State \mid \exists s \in S : (s, s') \in R \,\} \\
\mathsf{pre} &\overset{def}{=} \lambda\, S \,.\, \{\, s \in State \mid \exists s' \in S : (s, s') \in R \,\} \\
\widetilde{\mathsf{pre}} &\overset{def}{=} \lambda\, S \,.\, \{\, s \in State \mid \forall s \in State : (s, s') \in R \Rightarrow s' \in S \,\}
\end{aligned}$$

The operator pre is not much of interest in our setting. We concentrate on post and $\widetilde{\mathsf{pre}}$ and summarize some of their well-known properties. The following list is not to be meant complete, but it suffices for our further discussion.

Proposition 2.1.3. *Given a transition system $\mathcal{S} = \langle State, \mathsf{init}, R\rangle$, the following properties hold:*

(i) post *distributes over joins,*

(ii) $\widetilde{\mathsf{pre}}$ *distributes over meets,*

(iii) post *and* $\widetilde{\mathsf{pre}}$ *are monotone,*

(iv) $\mathsf{post} \circ \widetilde{\mathsf{pre}}$ *is reductive:* $\forall S \subseteq State : \mathsf{post}(\widetilde{\mathsf{pre}}(S)) \subseteq S$,

(v) $\widetilde{\mathsf{pre}} \circ \mathsf{post}$ *is extensive:* $\forall S \subseteq State : S \subseteq \widetilde{\mathsf{pre}}(\mathsf{post}(S))$,

(vi) $\forall S, S' \subseteq State : \mathsf{post}(S) \subseteq S' \iff S \subseteq \widetilde{\mathsf{pre}}(S')$.

All properties can be easily proven just using the definitions of post and $\widetilde{\mathsf{pre}}$, respectively. In particular, (iii) follows from properties (i) and (ii), and (vi) follows from properties (iii), (iv), and (v).

Proposition 2.1.4. *For a transition system $\mathcal{S}$ the following is equivalent:*

(i) R is total and deterministic,

(ii) $\widetilde{\mathsf{pre}}$ *is a homomorphism on the power set Boolean algebra of State.*

2.1.2 State Invariants and Reachability

A *state invariant* expresses a temporal property of a transition system. The expressiveness of state invariants is restricted to a particular subclass of temporal properties, so called *safety properties*. Safety properties are properties that guarantee the absence of abnormal system behavior in the sense that there is no possible execution that leads to a particular set of error states. We now formally investigate the connection between state invariants and reachability of system states.

Given a transition system $\mathcal{S}$ with

$$\mathcal{S} = \langle State, \mathsf{init}, R\rangle$$

the post operator post is extended to the operator F, as follows:

$$\begin{aligned}
F &\in 2^{State} \to 2^{State} \\
F &\overset{def}{=} \lambda\, S \,.\, \mathsf{init} \cup \mathsf{post}(S).
\end{aligned}$$

The set of reachable system states is now defined in terms of the operator F.

Definition 2.1.5 (Reachable States). *The set of reachable system states* reach *is the least fixed point of* post *under* init, *i.e.:*

$$\mathsf{reach} \overset{def}{=} \mathsf{lfp}(F).$$

The monotonicity of post guarantees the existence of the least fixed point of the operator F, thus the set of reachable states is well-defined.

Definition 2.1.6 (State Invariant). *A **state invariant** $I \subseteq State$ of transition system $\mathcal{S}$ is an over-approximation of the reachable states:* $\mathsf{reach} \subseteq I$. *A state invariant I is called **inductive**, if it is closed under* post: $\mathsf{post}(I) \subseteq I$.

Proposition 2.1.7. *$I \subseteq State$ is an inductive state invariant if and only if it is closed under the operator F, i.e. $F(I) \subseteq I$.*

Since any inductive state invariant is closed under the operator F, algorithms that synthesize state invariants rely on the iterative construction of the least fixed point of F, or to be more precise, the construction of approximations of $\mathsf{lfp}(F)$.

2.1.3 Abstract Interpretation and Galois Connections

The problem whether a given set of program states is reachable in an infinite state systems is in general undecidable. As a consequence, the construction of $\mathsf{lfp}(F)$ might not terminate. Even for systems with a finite state space it might be infeasible to construct $\mathsf{lfp}(F)$, because the state space is still to large to be managed by an exhaustive exploration of all reachable states. Hence, there is a need for approximation.

The idea of abstract interpretation is to come up with an abstraction of a concrete system $\mathcal{S}$ that mimics the behavior of $\mathcal{S}$. This abstraction has to be safe in the sense that every state invariant of the abstraction can be mapped to a state invariant of the concrete system. Abstract interpretation based on Galois connections [7, 8] is a formal framework for the construction of such abstractions.

The basic idea of this approach is that, instead of constructing $\mathsf{lfp}(F)$ on the concrete (complete) lattice $\langle 2^{State}, \subseteq, \emptyset, State, \cup, \cap\rangle$, the least fixed point of an abstraction of F is constructed on some (complete) abstract lattice $\langle D^{\#}, \sqsubseteq, \bot, \top, \sqcup, \sqcap\rangle$ of finite hight. The abstraction of F is defined in terms of abstraction function α, mapping sets of states to elements in the abstract domain, and concretisation function γ, mapping abstract values back to sets of states:

$$\begin{array}{rcl}\alpha & \in & 2^{State} \rightarrow D^{\#} \\ \gamma & \in & D^{\#} \rightarrow 2^{State}.\end{array}$$

The intuitive notion of a safe abstraction can now be formalized in terms of α and γ. The abstraction is safe, if every set of states is over-approximated by the consecutive application of α and γ:

$$\forall S \in 2^{State} : S \subseteq \gamma(\alpha(S)).$$

Finding some pair of safe abstraction and concretisation function is usually not a hard problem. However, we are not interested in any safe abstraction, we are interested in the best possible safe abstraction that can be defined for the chosen abstract lattice. The notion of a Galois connection formalizes this requirement.

Definition 2.1.8 (Galois Connection). *Given two partially ordered sets $\langle D, \subseteq\rangle$ and $\langle D^{\#}, \sqsubseteq\rangle$, the pair $\langle \alpha, \gamma\rangle$ is called a **Galois connection**, or a pair of **adjoint functions** iff:*

$$\forall c \in D, a \in D^{\#} : \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a).$$

Note that, without mentioning it explicitly, we have already seen an example for a Galois connection. As mentioned in Proposition 2.3.7, the operators post and $\widetilde{\mathsf{pre}}$ form a Galois connection on the power set lattice of $State$.

There are several equivalent characterizations of Galois connections. The following proposition captures some of them.

Proposition 2.1.9. *Let $\langle D, \subseteq, \bot, \top, \cup, \cap \rangle$ and $\langle D^\#, \sqsubseteq, \bot^\#, \top^\#, \sqcup, \sqcap \rangle$ be two complete lattices. For two functions $\alpha \in D \to D^\#$ and $\gamma \in D^\# \to D$ the following is equivalent:*

(i) $\langle \alpha, \gamma \rangle$ forms a Galois connection,

(ii) the following two conditions hold:

- *γ is a complete meet-morphism, i.e.*
  $\forall S^\# \subseteq D^\# : \gamma(\bigsqcup S^\#) = \bigcup\{ \gamma(a) \mid a \in S^\# \}$ and $\gamma(\top^\#) = \top$
- *$\alpha = \lambda\, c \in D\, .\, \bigsqcap\{ a \in D^\# \mid c \subseteq \gamma(a) \}$,*

(iii) the following two conditions hold:

- *α is a complete join-morphism, i.e.*
  $\forall S \subseteq D : \alpha(\bigcap S) = \bigsqcap\{ \alpha(c) \mid c \in S \}$ and $\alpha(\bot) = \bot^\#$
- *$\gamma = \lambda\, a \in D^\#\, .\, \bigcup\{ c \in D \mid \alpha(c) \sqsubseteq a \}$,*

(iv) the following three conditions hold:

- *α and γ are monotone,*
- *$\alpha \circ \gamma$ is reductive: $\forall a \in D^\# : \alpha(\gamma(a)) \sqsubseteq a$,*
- *$\gamma \circ \alpha$ is extensive: $\forall c \in D : c \subseteq \gamma(\alpha(c))$.*

For proof see [8].

The pair of abstraction and concretisation function $\langle \alpha, \gamma \rangle$ defines the best possible abstraction for D and $D^\#$ if and only if it forms a Galois connection. This is due to the fact that given that α and γ form a Galois connection, α maps a set of states S to the smallest abstract value that over-approximates S under γ. If $\sqcap$ is the meet-operation on the abstract lattice, according to Proposition 2.1.9, we have:

$$\alpha(S) = \bigsqcap\{ a \in D^\# \mid S \subseteq \gamma(a) \}.$$

We are now able to characterize the most precise abstraction $F^\#$ of the operator F. It is given by the composition of α, F, and γ:

$$\begin{array}{rcl} F^\# & \in & D^\# \to D^\# \\ F^\# & = & \alpha \circ F \circ \gamma. \end{array}$$

The monotonicity of F is preserved by the abstraction. This gives us a continuous operator $F^\#$ on the finite-hight abstract lattice. The computation of $\mathsf{lfp}(F^\#)$ is based on Kleene's fixed point characterization for continuous operators:

$$\mathsf{lfp}(F^\#) = \bigsqcup_{n \in \mathbb{N}} F^{\#n}(\bot).$$

We increasingly iterate $F^\#$, starting from the bottom element $\bot$ of the abstract domain, until we reach the least fixed point. The absence of infinite ascending chains in the abstract domain ensures termination.

2.2 First-Order Logic with Transitive Closure

In order to describe the memory state (program store) of a program manipulating heap-allocated data structures, we need an appropriate memory model, i.e. we have to formalize program stores in terms of mathematical objects.

We follow the setting in [27] and represent concrete program stores using first-order logical structures. First-order logic also serves as a specification language for the properties we want to analyze. However, it is known that first-order logic itself is too weak to express various interesting properties of graphs. For instance reachability or connectivity cannot be expressed. Extending first-order logic with transitive closure is one possible way to overcome this weakness. The resulting logic is strong enough to express the properties we are interested in and in addition allows us to relate the formal concrete semantics of program stores and their symbolic abstraction in a concise way.

We assume that V is a countable infinite set of variables with typical elements $v, v' \in V$. First-order formulas are defined with respect to a given signature of predicate and function symbols. Since functions can be axiomatized in first-order logic via predicates, we only consider a signature of predicate symbols.

Definition 2.2.1 (Syntax of $\mathsf{FO}^{\mathsf{TC}}$). *Given a signature Σ of predicate symbols p with arity $n \geq 0$ (written p/n), the set $\mathsf{FO}^{\mathsf{TC}}[\Sigma]$ of first-order formulas with transitive closure over Σ is defined as follows:*

$$\begin{array}{rll}\varphi, \psi \in \mathsf{FO}^{\mathsf{TC}}[\Sigma] ::= & \mathsf{true} \mid \mathsf{false} & \\ & \mid v \approx v' & \\ & \mid p(v_1, \ldots, v_n) & p/n \in \Sigma \\ & \mid \neg\varphi \mid \varphi \vee \psi & \\ & \mid \exists\, v.\varphi & \\ & \mid (\mathsf{TC}\, v_1, v_2.\varphi)(v, v') & \textit{(transitive closure)}\end{array}$$

We introduce the usual syntactic abbreviations for conjunction $\wedge$, implication $\rightarrow$, equivalence $\leftrightarrow$, and universal quantification $\forall$ (cf. Figure 2.1). Additionally, for a formula $\varphi(v, v')$, we introduce the abbreviations $\varphi^+(v, v')$ for the transitive closure, and $\varphi^*(v, v')$ for the reflexive transitive closure of φ. Parenthesis are omitted due to the following operator precedence $\succ_p$:

$$\neg \succ_p \wedge \succ_p \vee \succ_p \rightarrow \succ_p \leftrightarrow \succ_p \mathsf{TC} \succ_p \forall \succ_p \exists.$$

For a quantifier $Q \in \{\exists, \forall\}$ we abbreviate $Qv_1. \ldots Qv_n. \varphi$ by $Qv_1, \ldots, v_n. \varphi$. For $Q \in \{\exists, \forall, \mathsf{TC}\}$ and formula $Qv_1, \ldots, v_n. \varphi$, we call φ the *scope* of $Qv_1, \ldots, v_n$. An occurrence of a variable x is *bound* if it is inside the scope of $Qv_1, \ldots, v_n$ and $x \in \{v_1, \ldots, v_n\}$. Other occurrences are called *free*. A formula is *closed* if no free variables occur. We write $\varphi(v_1, \ldots, v_n)$ to indicate that at least the variables $v_1, \ldots, v_n$ occur free in the formula φ.

The semantics of formulas is given in the usual way, i.e. formulas are interpreted over first-order logical structures.

Definition 2.2.2 (Logical Structure). *A **logical structure** over Σ is a tuple $S = \langle U, \iota \rangle$, where*

- *U is a nonempty set, the **universe** of S, and*

$\varphi \wedge \psi \overset{def}{=} \neg(\neg\varphi \vee \neg\psi)$	(conjunction)
$\varphi \rightarrow \psi \overset{def}{=} \neg\varphi \vee \psi$	(implication)
$\varphi \leftrightarrow \psi \overset{def}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$	(equivalence)
$\forall v.\varphi \overset{def}{=} \neg\exists\, v.\neg\varphi$	(universal quantification)
$\varphi^+(v,v') \overset{def}{=} (\mathsf{TC}\, v, v'.\varphi(v,v'))(v,v')$	(transitive closure of $\varphi(v,v')$)
$\varphi^*(v,v') \overset{def}{=} \varphi^+(v,v') \vee v \approx v'$	(reflexive, transitive closure of $\varphi(v,v')$)

Figure 2.1: Additional syntactic abbreviations

- *ι is the **interpretation function** for predicate symbols in Σ, i.e. for an n-ary predicate symbol $p/n \in \Sigma$, the function $\iota\, p \in U^n \rightarrow \mathbb{B}$ assigns truth-values to n-tuples over U.*

*We write U^S for the universe of S, if it is not given an explicit name. The set Σ-*Struct *denotes all logical structures over Σ.*

Definition 2.2.3 (Semantics of $\mathsf{FO}^{\mathsf{TC}}$). *Let $S = \langle U, \iota\rangle$ be a logical structure and $\beta \in V \rightarrow U$ an assignment that maps variables to elements of the universe of S. The interpretation function $[\![\cdot]\!]^S \in \mathsf{FO}^{\mathsf{TC}}[\Sigma] \rightarrow (V \rightarrow U) \rightarrow \mathbb{B}$, assigning truth-values to $\mathsf{FO}^{\mathsf{TC}}$ formulas, is inductively defined as follows:*

$$\begin{array}{rcl}
[\![\mathsf{false}]\!]^S(\beta) & = & 0 \\
[\![\mathsf{true}]\!]^S(\beta) & = & 1 \\
[\![v_1 \approx v_2]\!]^S(\beta) & = & \textit{if}\ \beta\, v_1 = \beta\, v_2\ \textit{then}\ 1\ \textit{else}\ 0 \\
[\![p(v_1,\ldots,v_n)]\!]^S(\beta) & = & \iota\, p\,(\beta\, v_1, \ldots, \beta\, v_n) \\
[\![\neg\varphi]\!]^S(\beta) & = & 1 - [\![\varphi]\!]^S(\beta) \\
[\![\varphi \vee \psi]\!]^S(\beta) & = & \max\{[\![\varphi]\!]^S(\beta), [\![\psi]\!]^S(\beta)\} \\
[\![\exists\, x.\varphi]\!]^S(\beta) & = & \max\{\, [\![\varphi]\!]^S(\beta[x \mapsto u]) \mid u \in U\,\} \\
[\![(\mathsf{TC}\, v_1, v_2.\varphi)(v,v')]\!]^S(\beta) & = & 1 \iff \textit{there exists}\ u_1,\ldots,u_n \in U\ \textit{s.t.} \\
 & & u_1 = \beta\, v, u_n = \beta\, v'\ \textit{and} \\
 & & \min\limits_{1 \le i < n}\{[\![\varphi]\!]^S(\beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])\} = 1
\end{array}$$

We introduce the following notions of validity and entailment:
*S **satisfies** φ* under *β:* $\quad S,\beta \models \varphi \overset{def}{\iff} [\![\varphi]\!]^S(\beta) = 1.$
*φ **entails** ψ:* $\quad \varphi \models \psi \overset{def}{\iff}$ *$S,\beta \models \varphi$ implies $S,\beta \models \Psi$, for all S and β.*
*φ is valid in S (S is a **model** of φ):* $\quad S \models \varphi \overset{def}{\iff}$ *$S,\beta \models \varphi$, for all $\beta \in V \rightarrow U$.*
The set of all models of φ is denoted by:

$$\mathsf{Mod}(\varphi) \overset{def}{=} \{\, S \in \Sigma\text{-Struct} \mid S \models \varphi\,\}.$$

The set of all finite models of φ is denoted by:

$$\mathsf{Mod}_{\mathit{fin}}(\varphi) \overset{def}{=} \{\, S \in \Sigma\text{-Struct} \mid S \models \varphi\ \textit{and}\ U^S\ \textit{is finite}\,\}.$$

```
typedef struct node {
  struct node *n;
  int data;
} *List;
```

Figure 2.2: A simple type definition for singly-linked lists in C

2.3 System Description

2.3.1 Program Stores

In the analysis of programs without dynamic memory allocation a program store is described as a valuation of program variables to appropriate data objects, e.g. integers. That means, it can be modeled as an assignment from first-order variables to elements of the appropriate data domain.

In order to be able to represent the state of the heap, we need more than just first-order variable assignments. The state of the heap can be modeled as a graph. The nodes of the graph represent allocated heap objects and edge relations reflect how objects are connected via pointers. For each pointer-valued field in a data structure the graph has one corresponding edge relation. Graph-like structures can be equivalently described as logical structures, where edge relations are represented as binary predicates. This logical characterization of program stores conforms to the setting in [27].

In order to represent program stores that contain a particular data structure type, we consider a signature of predicate symbols Σ consisting of:

- a unary predicate symbol x for each pointer-valued program variable `x`,
- a unary predicate symbol $null$ for `NULL`,
- and a binary predicate symbol n for each pointer-valued structure field `n`.

In our concrete model we abstract away all non-pointer-valued program variables or structure fields. In addition, we restrict ourselves to stores that may just contain one single data structure type at any time. As an example, consider the *List* data type given in Figure 2.2. The signature for program stores containing singly-linked lists that are accessible by a program variable `x` is given by:

$$\Sigma_{List} = \{x/1, null/1, next/2\}.$$

A program store S is given by a logical structure $S = \langle U, \iota \rangle$ over Σ. U represents the finite set of allocated instances of the stored data structure type. We call the elements of *U nodes*. The value `NULL` is represented as a node of its own, we added the unary predicate $null$ to the signature, in order to distinguish `NULL` from all other nodes. The interpretation function ι interprets the predicates according to the program variables and pointer-valued data structure fields.

Not every logical structure represents a store. The number of objects that is allocated at a particular point in the program execution is always finite. Thus, we are only interested in finite structures. However, there are additional constraints that have to be satisfied. Since any structure field or program variable can only point to exactly one other node, the binary predicates should denote functional relations and the unary predicates should represent singleton sets. All necessary constraints can be modeled by an *integrity formula* [30]. The actual choice of this

U^S	$\{u_1, u_2, u_3\}$
ι^S	unary predicates; binary predicates

unary predicates

	u_1	u_2	u_3
x	1	0	0
$null$	0	0	1

binary predicates

n	u_1	u_2	u_3
u_1	0	1	0
u_2	0	0	1
u_3	0	0	0

$$\begin{tikzcd}
x \arrow[r] & u_1 \arrow[r, "n"] & u_2 \arrow[r, "n"] & u_3 \\
& & & null \arrow[u]
\end{tikzcd}$$

Figure 2.3: A store S containing a singly linked list accessible by variable `x`.

formula depends on the concrete model of program stores one has in mind. The minimal requirements on stores that we will refer to in this work are modeled by the following integrity formula F:

$$\begin{aligned}
F \overset{def}{=} \quad & \bigwedge_{x/1 \in \Sigma} \exists\, v.x(v) \land \forall v'.x(v') \to v \approx v' \\
\land & \bigwedge_{n/2 \in \Sigma} \forall v.\neg null(v) \to \exists\, v'.n(v,v') \land \forall v''.n(v,v'') \to v' \approx v'' \\
\land & \forall v, v'.n(v,v') \to \neg null(v).
\end{aligned}$$

Once we have chosen an appropriate integrity formula, we can define the set of all program stores *Store*.

Definition 2.3.1 (Program Stores). *Let F be an integrity formula. The set of program stores Store for F is the set of all finite models of F:*

$$Store \overset{def}{=} \mathsf{Mod}_{fin}(F).$$

In order to visualize logical structures we adopt the graphical representation used in [27]. The elements of U are represented as nodes in a graph. Binary predicates are represented as labeled edges and unary predicates as labeled arrows pointing to the nodes they are satisfied by. Figure 2.3 shows a program store both represented as a logical structure over Σ_{List} and using the graphical notation.

2.3.2 Program Semantics

In order to syntactically restrict the class of pointer programs that we have to consider, we permit nested dereferencing of program variables and structure fields. Any program can be translated into this restricted class, possibly by introducing fresh temporary program variables. This restriction leaves us with five kinds of atomic commands:

- commands assigning program variables or `NULL` to program variables:
  `x =` t, where t is a program variable, or `NULL`,
- commands assigning field values to program variables:
  `x = y->n`,

- commands assigning program variables or `NULL` to structure fields:
  `x->n = ` t, where t is a program variable, or `NULL`,
- allocation of a single fresh memory cell:
  `x = malloc(),`
- deallocation of a memory cell:
  `free(x).`

We explain the semantics of programs in terms of program stores, rather then program states. A program state can be considered as a tuple consisting of program store and program counter, hence it is always clear how to extend from stores to states.

Let c be some atomic command. The semantics of c is given by the transition relation $\xrightarrow{c}$ on stores. In [27] the transition relation is characterized in terms of *predicate-update formulas*. There is a relationship between predicate-update formulas and *weakest liberal preconditions* (wlp) that justifies this approach. In the following, we want to further investigate on this relationship.

The post and wlp operator on sets of stores are standard as follows:

$$\begin{aligned} \mathsf{post}_c &\overset{def}{=} \lambda \mathcal{M} \, . \, \{ S' \in \mathit{Store} \mid \exists S \in \mathcal{M} : S \xrightarrow{c} S' \} \\ \widetilde{\mathsf{pre}}_c &\overset{def}{=} \lambda \mathcal{M} \, . \, \{ S \in \mathit{Store} \mid \forall S' \in \mathit{Store} : S \xrightarrow{c} S' \Rightarrow S' \in \mathcal{M} \}. \end{aligned}$$

Since we will represent program stores symbolically using formulas, it is crucial to be able to handle the symbolic execution of commands. The effect of some command on the stores denoted by some formula should be expressible in terms of a predicate transformer on formulas. This predicate transformer should be computable as a purely syntactic operation, such that the denoted set of stores must not be considered explicitly.

In the setting of programs without heap-allocated data, where a store is represented as a first-order valuation of program variables, this causes no problems, because the predicate transformers are again expressible in first-order logic. Problems arise from the fact that, in our setting of programs with heap allocated data, stores are represented as first-order structures over a signature containing binary predicates. However, binary predicates correspond to second-order variables. Thus, it is in general not guaranteed that post_c or $\widetilde{\mathsf{pre}}_c$ are expressible as predicate transformers in first-order logic.

As we will see, the fact that all atomic commands are deterministic at least allows us to express the operator $\widetilde{\mathsf{pre}}_c$ as a purely syntactic operation on formulas. Although post_c is still not first-order expressible, due to the fact that post_c and $\widetilde{\mathsf{pre}}_c$ form a Galois connection on the power set lattice of stores, it suffices that one of the two adjoints can be expressed.

At first, we do not want to consider allocation and deallocation of heap cells. Since the remaining commands may only change pointer values, they simply correspond to updates of the interpretation of predicate symbols. The universe is unaffected by all these commands. Hence, we can fix a universe U that is shared by all logical structures that we consider for the rest of this section.

For a given logical structure S, a formula φ with n free variables denotes an n-ary relation over the universe U. We use an isomorphic representation and consider the denotation $[\![\varphi]\!]$ of some formula φ to be the set of all pairs of stores S and assignments β that satisfy φ:

$$[\![\varphi]\!] \overset{def}{=} \{ (S, \beta) \in \mathit{Store} \times (V \to U) \mid S, \beta \models \varphi \}.$$

c	$p(v_1,\dots,v_n)$	$p'_c(v_1,\dots,v_n)$
`x = NULL`	$x(v)$	$\mathit{null}(v)$
`x = y`	$x(v)$	$y(v)$
`x = y->n`	$x(v)$	$\exists\, v'.y(v') \wedge n(v',v)$
`x->n = NULL`	$n(v_1,v_2)$	$n(v,v') \wedge \neg x(v) \vee x(v') \wedge \mathit{null}(v')$
`x->n = y`	$n(v,v')$	$n(v,v') \wedge \neg x(v) \vee x(v) \wedge y(v')$

Table 2.1: Predicate-update formulas.

In order to give a predicate transformer on arbitrary $\mathsf{FO}^{\mathsf{TC}}$ formulas we have to extend the transformer functions post_c and $\widetilde{\mathsf{pre}}_c$ to functions on subsets of *extended stores*:

$$\mathit{ExtStore} \stackrel{\mathit{def}}{=} \mathit{Store} \times (V \to U).$$

We define the extended operators $\mathsf{ext\text{-}post}_c$ and $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$ simply by keeping the second component of an extended store untouched:

$$\begin{aligned}
\mathsf{ext\text{-}post}_c &\in 2^{\mathit{ExtStore}} \to 2^{\mathit{ExtStore}} \\
\mathsf{ext\text{-}post}_c &\stackrel{\mathit{def}}{=} \lambda\,\mathcal{M}\,.\,\{\,(S',\beta) \in \mathit{ExtStore} \mid \exists S \in \mathit{Store} : S \xrightarrow{c} S' \wedge (S,\beta) \in \mathcal{M}\,\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c &\in 2^{\mathit{ExtStore}} \to 2^{\mathit{ExtStore}} \\
\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c &\stackrel{\mathit{def}}{=} \lambda\,\mathcal{M}\,.\,\{\,(S,\beta) \in \mathit{ExtStore} \mid \forall S' \in \mathit{Store} : S \xrightarrow{c} S' \Rightarrow (S',\beta) \in \mathcal{M}\,\}.
\end{aligned}$$

Since $\widetilde{\mathsf{pre}}_c$ and post_c are just lifted from stores to extended stores, their characteristic properties are preserved. In particular, the following propositions hold.

Proposition 2.3.2. *The operators* $\mathsf{ext\text{-}post}_c$ *and* $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$ *form a Galois connection on the power set lattice of extended stores.*

Proposition 2.3.3. *The relation* $\xrightarrow{c}$ *is total and deterministic if and only if* $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$ *is a homomorphism on the power set Boolean algebra of extended stores.*

If the transition relation for atomic command c is total and deterministic then we interpret post_c as a total function on stores and $\mathsf{ext\text{-}post}_c$ as a total function on extended stores, respectively.

The predicate-update formulas that are used in [27] to describe the semantics of commands can now be formally defined in terms of the extended weakest liberal precondition $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$.

Definition 2.3.4 (Predicate-Update Formula). *A* ***predicate-update formula*** p'_c *for* n*-ary predicate symbol* p *and atomic command* c *is an* $\mathsf{FO}^{\mathsf{TC}}$ *formula whose denotation corresponds to the extended wlp of the denotation of* p*:*

$$\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![p(v_1,\dots,v_n)]\!]) = [\![p'_c(v_1,\dots,v_n)]\!].$$

Table 2.1 shows the predicate-update formulas for all atomic commands that we considered so far. Predicate symbols whose interpretation does not change under some command, i.e. where the predicate-update formula corresponds to the predicate symbol itself, are omitted. It is easy to see that these formulas indeed capture the semantics of the appropriate commands.

Proposition 2.3.5. *If the relation $\xrightarrow{c}$ is total and deterministic and if for every predicate symbol p there exists a predicate-update formula p'_c then the extended wlp of an $\mathsf{FO}^{\mathsf{TC}}$ formula φ can be computed by substituting syntactically all occurrences of predicate symbols in φ with their predicate-update formulas:*

$$\mathsf{ext}\text{-}\widetilde{\mathsf{pre}}_c([\![\varphi]\!]) = [\![\varphi[p'_c(v_1, \ldots, v_n)/p(v_1, \ldots, v_n)]]\!].$$

Proposition 2.3.5 gives rise to define weakest liberal preconditions as a predicate transformer on formulas that can be computed purely syntactically.

Definition 2.3.6 (Weakest Liberal Preconditions of Formulas). *The weakest liberal precondition of an $\mathsf{FO}^{\mathsf{TC}}$ formula φ for command c, written $\mathsf{wlp}_c(\varphi)$, is obtained by substituting syntactically every occurrence of a predicate symbol $p(v_1, \ldots, v_n)$ in φ with its predicate-update formula $p'_c(v_1, \ldots, v_n)$:*

$$\mathsf{wlp}_c \overset{def}{=} \lambda\, \varphi \in \mathsf{FO}^{\mathsf{TC}} .\ \varphi[p'_c(v_1, \ldots, v_n)/p(v_1, \ldots, v_n)].$$

The following proposition relates the symbolic weakest liberal precondition on formulas with the post operator on stores.

Proposition 2.3.7. *If $\xrightarrow{c}$ is total and deterministic then for a store S, $\mathsf{FO}^{\mathsf{TC}}$ formula φ and assignment β, we have:*

$$S, \beta \models \mathsf{wlp}_c(\varphi) \iff \mathsf{post}_c(S), \beta \models \varphi.$$

In the setting of pointer programs, all atomic commands are deterministic. Unfortunately, in general not all of them are total. Dereferencing pointers to `NULL` is a typical operation that is not permitted, i.e. in that case the post operator can not be defined as a total function on stores. This problem can be solved by applying the standard trick of introducing additional error states that turn the partial functional relation $\xrightarrow{c}$ on program stores into a total functional relation on program states.

Handling allocation and deallocation is a bit more tricky. As an example, consider the command c: `x = malloc()`. It is not possible to give a predicate-update formula for predicate symbol x, since the node on which program variable `x` points to after execution of c does not exist in the predecessor store. One possible solution is to add the new node and temporarily introduce a predicate symbol that distinguishes the added node from all others. This temporary predicate symbol can then be used to define the predicate-update formulas.

In order to keep the transition relation simple, we ignore allocation and deallocation in this work. However, this is not a real restriction, since it is possible to extend the presented framework in an appropriate way.

Chapter 3

Abstraction Framework

In this chapter we develop our abstraction framework. We first give a symbolic abstract domain that incorporates abstraction techniques from shape analysis. After that we apply methods from predicate abstraction, in order to abstract programs manipulating heap-allocated data structures by Boolean programs. The obtained Boolean program is called a *Boolean heap program*. We formally characterize a Boolean heap program as an over-approximation of the best abstract post operator on the chosen abstract domain and analyze its precision.

3.1 Node Predicate Abstraction

Finding a suitable abstract domain is considered to be one of the hardest parts in abstract interpretation. In the previous chapter we have seen how concrete program stores can be represented by logical structures. A formula is a symbolic representation of sets of logical structures, namely the set of its models. Hence, this shifts the problem of finding a suitable abstract domain to the problem of finding a suitable class of formulas.

3.1.1 Node Predicates

Graph-based abstract domains for shape analysis are induced by a chosen set of *shape properties*. In [27] these shape properties are identified as unary abstraction predicates that denote sets of nodes in the abstracted program stores. In the following, we are going to propose a symbolic abstract domain that is parameterized by unary abstraction predicates. We call these abstraction predicates *node predicates*.

Definition 3.1.1 (Node Predicates). *A **node predicate** $p(v)$ is an $\mathsf{FO}^{\mathsf{TC}}$ formula with a single dedicated free variable v.*

Figure 3.1 shows some typical node predicates in the context of singly-linked lists.

Definition 3.1.2. *Let Pred be a finite set of node predicates. A **literal** over Pred is a node predicate in Pred or its negation. A conjunction P of literals over Pred is called **complete** or a **monomial**, if for every node predicate $p \in Pred$, exactly one of its literals is a conjunct in P. The set $\mathcal{F}_{Pred}$ denotes the set of all Boolean combinations of node predicates in Pred. Let $\varphi(v) \in \mathcal{F}_{Pred}$. For a store S and a node $u \in U^S$ we write $S, u \models \varphi(v)$ as a short-notation for $S, [v \mapsto u] \models \varphi(v)$.*

$x(v)$	(v is pointed to by `x`)
$null(v)$	(v is `NULL`)
$r_x(v) \overset{def}{=} \exists\, v'.x(v') \wedge n^*(v', v)$	(v is reachable from `x`)
$r_x^+(v) \overset{def}{=} \exists\, v'.x(v') \wedge n^+(v', v)$	(v is reachable in at least one step)
$is(v) \overset{def}{=} \exists\, v', v''.n(v', v) \wedge n(v'', v) \wedge v' \not\approx v''$	(v is shared by at least two nodes)

Figure 3.1: Typical node predicates in the context of singly-linked lists.

3.1.2 Abstract Domain

For the rest of this chapter we fix a particular finite set of node predicates *Pred*. For notational convenience we consider *Pred* to be closed under negation.

The idea behind the abstraction we are going to propose is that a formula in the abstract domain is valid in a store, if it represents all nodes in the store. This leads to the following definition of an *abstract store*.

Definition 3.1.3 (Abstract Store $AbsStore[Pred]$**).** *An* ***abstract store*** Ψ *over Pred is a formula* Ψ *of the form:*

$$\Psi = \forall v.\psi(v)$$

where $\psi(v) \in \mathcal{F}_{Pred}$ *is a Boolean combination of node predicates. With* $AbsStore[Pred]$ *we denote the set of all abstract stores over Pred.*

In order to treat joins in the control flow in an adequate way, the abstract domain should be closed under disjunctions. We take the disjunctive completion over abstract stores as our abstract domain.

Definition 3.1.4 (Abstract Domain $AbsDom[Pred]$**).** *The abstract domain over Pred is given by the set* $AbsDom[Pred]$ *of all disjunctions of abstract stores:*

$$AbsDom[Pred] \overset{def}{=} \{ \bigvee_{i \in I} \Psi_i \mid \forall i \in I :\ \Psi_i \in AbsStore[Pred] \}.$$

The elements of $AbsDom[Pred]$ *are partially ordered by the entailment relation* $\models$ *on formulas.*

Since an abstract store can be represented as a Boolean function over *Pred*, the abstract domain $AbsDom[Pred]$ is isomorphic to the power set of Boolean functions over *Pred*. Thus, it is isomorphic to a finite domain.

We omit the set of node predicates *Pred* as the parameter for *AbsStore* and *AbsDom* whenever it is clear which set of node predicates we refer to. We will follow the same convention for all functions that we will define on these domains in the following sections.

3.1.3 Best Abstraction

We need to give the best possible mapping of elements from the concrete domain $\langle 2^{Store}, \subseteq \rangle$ to elements of the abstract domain $\langle AbsDom, \models \rangle$ and *vice versa*. More precisely, following [8] we have to provide abstraction and meaning functions:

$$\alpha \quad \in \quad 2^{Store} \to AbsDom$$

$$\gamma \quad \in \quad AbsDom \to 2^{Store}$$

such that α and γ form a Galois connection. Since the concrete domain is given by sets of program stores that are represented as logical structures, the meaning of a formula Ψ in $AbsDom$ is given by the set of all its models, restricted to program stores.

Definition 3.1.5 (Meaning Function). *The meaning function γ that maps formulas in the abstract domain to sets of program stores is defined by:*

$$\begin{array}{rcl} \gamma & \in & AbsDom \to 2^{Store} \\ \gamma & \stackrel{def}{=} & \lambda\,\Psi\,.\,\{\, S \in Store \mid S \models \Psi \,\}. \end{array}$$

Proposition 3.1.6. *The function γ is a complete meet-morphism and a complete join-morphism.*

For a pair of adjoint functions $\langle \alpha, \gamma \rangle$, one adjoint determines the other. Given the definition of γ, α maps a set of stores to its smallest over-approximation with respect to γ.

Definition 3.1.7 (Abstraction Function). *The abstraction function α that maps abstract sets of program stores to abstract values is defined by:*

$$\begin{array}{rcl} \alpha & \in & 2^{Store} \to AbsDom \\ \alpha & \stackrel{def}{=} & \lambda\,\mathcal{M}\,.\,\bigwedge\{\, \Psi \in AbsDom \mid \mathcal{M} \subseteq \gamma(\Psi) \,\}. \end{array}$$

We write $\alpha(S)$ instead of $\alpha(\{S\})$ whenever α is applied to a single store S.

Proposition 3.1.8. *Abstraction function α and concretisation function γ form a Galois connection between the posets $\langle 2^{Store}, \subseteq \rangle$ and $\langle AbsDom, \models \rangle$.*

Although we defined the abstraction function α in terms of γ, what we need is a constructive characterization of α. We now give such a characterization.

If we consider a single store S, the abstraction function α maps S to the smallest abstract store that is valid in S. An abstract store is a universally quantified formula in $\mathcal{F}_{Pred}$. Hence, in order to construct $\alpha(S)$, we need to construct the smallest formula in $\mathcal{F}_{Pred}$ that covers all nodes in the universe of S.

Given a node u in the universe of S, we can assign an *abstract node* $P_{S,u}$ to u and S that is given by a conjunction of node predicates. The abstract node $P_{S,u}$ represents the equivalence class of all nodes in the universe of S that satisfy the same node predicates as u.

Definition 3.1.9 (Abstract Nodes). *An* ***abstract node*** *is a conjunction P of literals over Pred. Let S be a store and $u \in U^S$. The abstract node $P_{S,u}$ is the complete conjunction of node predicates that are satisfied by u in S:*

$$P_{S,u}(v) \stackrel{def}{=} \bigwedge\{\, p \in Pred \mid S, u \models p \,\}.$$

As illustrated in Figure 3.2, a store S is abstracted by the smallest covering of nodes in U^S by abstract nodes over $Pred$. The smallest covering is given by the disjunction of all abstract nodes $P_{S,u}$ for nodes $u \in U^S$. Formally, we obtain the following characterization of α.

Theorem 3.1.10 (Characterization of Best Abstraction). *Let $\mathcal{M}$ be a set of program stores. The image of $\mathcal{M}$ under α is characterized as follows:*

$$\alpha(\mathcal{M}) \models\!\!\!\dashv \bigvee_{S \in \mathcal{M}} \forall v.\ \bigvee_{u \in U^S} P_{S,u}(v).$$

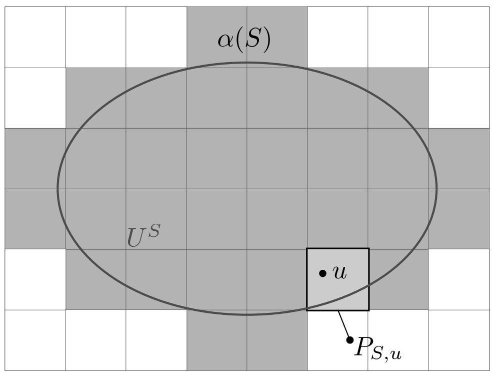


Figure 3.2: The universe U^S of a store S covered by abstract nodes over Pred.

Let us now illustrate the abstraction by an example.

Example. Consider again the list data type given in Figure 2.2. The set of abstraction predicates is given by literals over the node predicates

$$x(v), \mathit{null}(v), r_x(v)$$

that are defined in Figure 3.1. Figure 3.3 shows the abstraction of a store S containing a singly-linked list with three elements whose head is pointed to by program variable `x`.

Applying γ again to the abstraction of S does not only result in S itself, but in the set of all stores containing a singly-linked list with at least one element. The abstraction $\alpha(S)$ entails that `NULL` is reachable from program variable `x`. Since we deal with lists, this information is sufficient to guarantee that all lists in $\gamma(\alpha(S))$ are acyclic.

3.1.4 Expressiveness

Now when the decision for a particular abstract domain has been made, we briefly consider some aspects with respect to expressiveness. The goal of this section is not to give an exhaustive formal comparison to graph-based abstract domains for shape analysis, we focus on the aspect of how presence or absence of edges between abstract nodes are expressible.

In the following, we refer to the framework of parametric shape analysis via 3-valued logic [27]; see Section 6.1 for a detailed discussion. This approach uses three-value logical structures as a generalization of shape graphs.

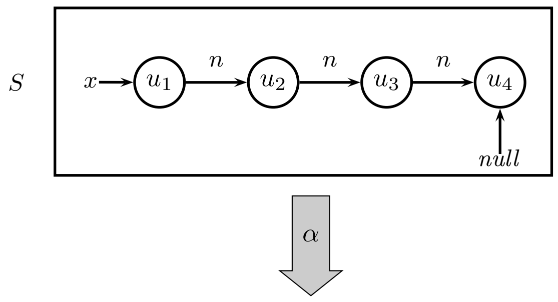
In [30] three-valued logical structures are translated into two-valued first-order formulas that represent the same set of concrete stores. Given two nodes in a three-valued logical structure that correspond to abstract nodes P and P', an n-edge between P and P' is translated to the constraint:

$$\forall v, v'. P(v) \land P'(v') \rightarrow n(v, v'). \tag{1}$$

The absence of an n-edge is translated accordingly:

$$\forall v, v'. P(v) \land P'(v') \rightarrow \neg n(v, v'). \tag{2}$$

The decision to use only unary node predicates in formulas of our abstract domain seems to be rather restrictive at first sight, because we cannot talk about



$$\begin{aligned}\alpha(S) &= \forall v. P_{S,u_1}(v) \vee P_{S,u_2}(v) \vee P_{S,u_3}(v) \vee P_{S,u_4}(v)\\ &= \forall v. \underbrace{[x(v) \wedge \neg null(v) \wedge r_x(v)]}_{P_{S,u_1}} \vee \underbrace{[\neg x(v) \wedge \neg null(v) \wedge r_x(v)]}_{P_{S,u_2}, P_{S,u_3}}\\ &\quad \vee \underbrace{[\neg x(v) \wedge null(v) \wedge r_x(v)]}_{P_{S,u_4}}\end{aligned}$$

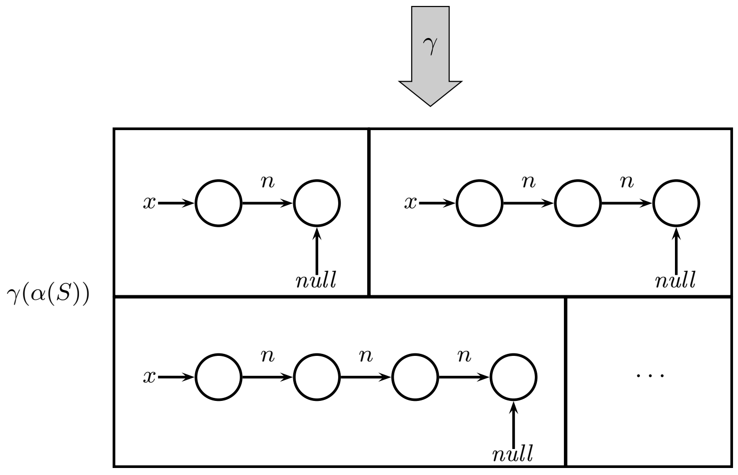


Figure 3.3: Abstraction and concretisation of a store S containing a singly-linked list.

edges between abstract nodes explicitly using the binary predicate symbols. Nevertheless, since we allow unary node predicates to be arbitrary $\mathsf{FO}^{\mathsf{TC}}$ formulas, it is possible to express the presence or absence of an edge by adding additional unary node predicates to the set *Pred*. We illustrate this with the constraints given above. Consider the following equivalence transformations:

$$\begin{array}{rcl}
\forall v, v'.P(v) \land P'(v') \to n(v, v') & \models\!\mid & \forall v.\neg P(v) \lor \underbrace{\forall v'.P'(v') \to n(v, v')}_{p_1(v)} \\
\forall v, v'.P(v) \land P'(v') \to \neg n(v, v') & \models\!\mid & \forall v.\neg P(v) \lor \neg \underbrace{\exists\, v'.P'(v') \land n(v, v')}_{p_2(v)}.
\end{array}$$

The right-hand sides are already in the right format with respect to our syntactic class of formulas, i.e. both are abstract stores, if we add the two additional abstraction predicates p_1 and p_2.

The abstract store $\alpha(S)$ for any store S that satisfies constraint (1) is guaranteed to imply constraint (1), because $\alpha(S)$ is the smallest abstract store that is valid in S. Hence, the abstraction preserves the information of an n-edge between the abstract nodes. The same holds for constraint (2) and the absence of the n-edge.

Thus, it is possible to express absence or presence of edges between abstract nodes by adding additional node predicates. However, it turns out that, at least for the analysis of properties like reachability, it is possible to obtain precise results without expressing edges in the way explained above. In Chapter 4 we will discuss a class of node predicates that seems to be better suited for this purpose.

3.2 Abstract Transformer

An integral part of an abstract interpretation based analysis is the abstract transformer function. In the case of a forward analysis, it is given by an abstraction of the operator post. In this section, we develop an abstraction of the concrete post operator that can be easily implemented.

3.2.1 Best Abstract Transformer

For the rest of this chapter, let c be some fixed atomic command and let post be the post operator for command c. Remember that c is deterministic yet not necessarily total. However, by adding appropriate guards to c, it is always possible to restrict the domain of the transition relation $\xrightarrow{c}$ such that post becomes a total function on the restricted domain. For this reason, we consider $\mathsf{post}(S)$ to be well-defined whenever post is applied to a single store S.

Given a Galois connection $\langle\alpha, \gamma\rangle$ between concrete domain and abstract domain, as explained in Section 2.1.3, the best abstraction $F^{\#}$ of a function F on the concrete domain is given by the composition of α, F, and γ:

$$\begin{array}{rcl}
F^{\#} & \in & \mathit{AbsDom} \to \mathit{AbsDom} \\
F^{\#} & = & \alpha \circ F \circ \gamma.
\end{array}$$

If we apply the characterization of α that we gave in Theorem 3.1.10, the best abstract post operator $\mathsf{post}^{\#}$ is given by:

$$\begin{array}{rcl}
\mathsf{post}^{\#} & \in & \mathit{AbsDom} \to \mathit{AbsDom} \\
\mathsf{post}^{\#}(\Psi) & = & \alpha(\mathsf{post}(\gamma(\Psi))) = \bigvee_{S \in \gamma(\Psi)} \forall v. \bigvee_{u \in U^S} P_{\mathsf{post}(S),u}.
\end{array}$$

This characterization is by no means constructive. It relies on enumerating all concrete stores contained in the image of Ψ under γ, which is in general an infinite set. We need an algorithmic characterization of the best abstract post operator.

The naïve approach is to enumerate all abstract stores explicitly and check for each one whether it is contained in $\mathsf{post}^{\#}(\Psi)$. Given that n is the number of (unsigned) node predicates in *Pred*, considering all 2^{2^n} abstract stores explicitly, results in a doubly-exponential lower bound for the complexity of the computation of $\mathsf{post}^{\#}$. This is not feasible in practice. We need to restrict the number of possible abstract stores that may occur as a disjunct in $\mathsf{post}^{\#}(\Psi)$.

For this reason, our goal is to develop an approximation of the best abstract post operator that can be easily implemented. However, we require this operator to be formally characterized in terms of an abstraction of $\mathsf{post}^{\#}$, since we want to know exactly where we lose precision.

3.2.2 Context-Sensitive Abstract Operators

In order to find an implementable abstraction of the best abstract post operator, we will reduce the computation of the abstract post of an abstract store Ψ to the computation of an abstract post on abstract nodes that occur in Ψ. We start with some more general observations that will help us to accomplish this task. In the following, we discuss how an abstract post on abstract nodes can be defined and relate it to an appropriate abstract wlp operator.

As we have seen in Section 2.3.2, the weakest liberal precondition operator $\widetilde{\mathsf{pre}}$ can be extended from a function on sets of stores to a syntactic operation wlp on arbitrary $\mathsf{FO}^{\mathsf{TC}}$ formulas. Thus, we are in particular able to compute the weakest liberal precondition of any formula in $\mathcal{F}_{Pred}$[1]. However, since $\mathcal{F}_{Pred}$ is in general not closed under the operator wlp, we are interested in the best abstraction of wlp.

The best abstraction $\mathsf{wlp}^{\#}$ of wlp in $\mathcal{F}_{Pred}$ with respect to the entailment relation as the partial order on formulas is as expected:

$$\mathsf{wlp}^{\#} \overset{def}{=} \lambda\, \varphi \in \mathcal{F}_{Pred} \,.\, \bigvee\{\, \psi \in \mathcal{F}_{Pred} \mid \psi \models \mathsf{wlp}(\varphi) \,\}.$$

Respectively, the corresponding best abstract post operator on $\mathcal{F}_{Pred}$ is given by:

$$\mathsf{post}^{\#} \overset{def}{=} \lambda\, \varphi \in \mathcal{F}_{Pred} \,.\, \bigwedge\{\, \psi \in \mathcal{F}_{Pred} \mid \varphi \models \mathsf{wlp}(\psi) \,\}.$$

If we apply this abstraction in our setting, the result is more conservative than necessary. We want to use the abstract post operator on a formula in $\mathcal{F}_{Pred}$ that occurs as a sub-formula in an abstract store Ψ for the computation of the abstract post of Ψ itself. In order to guarantee soundness of the resulting abstract post operator on abstract stores, we only need to abstract post and wlp with respect to stores that are actually models of Ψ. We say we abstract post and wlp *in the context of* Ψ.

Formally, this can be accomplished by changing the partial order on the lattice $\mathcal{F}_{Pred}$. We replace entailment relation $\models$ by a relaxed entailment relation $\models_\Psi$ which is restricted to logical structures that are program stores and models of Ψ.

Definition 3.2.1. *Let $\mathcal{M}$ be a set of structures over Σ. For two $\mathsf{FO}^{\mathsf{TC}}$ formulas φ and ψ, we say φ entails ψ restricted to $\mathcal{M}$, written $\varphi \models_{\mathcal{M}} \psi$, if*

$$\forall S \in \mathcal{M}, \beta \in V \to U^S : \; S, \beta \models \varphi \Rightarrow S, \beta \models \psi$$

For a closed $\mathsf{FO}^{\mathsf{TC}}$ formula Γ, we write $\varphi \models_\Gamma \psi$, instead of $\varphi \models_{\gamma(\Gamma)} \psi$. That is, $\models_\Gamma$ is the entailment relation restricted to stores that are models of Γ. We call Γ the context.

[1]This includes abstract nodes.

Definition 3.2.2 (Context-sensitive Abstract Operators). *Let Γ be a closed* $\mathsf{FO}^{\mathsf{TC}}$ *formula. The* ***context-sensitive abstract operators*** *for Γ on $\mathcal{F}_{Pred}$ are given by the abstractions of* post *and* wlp *in the context of Γ:*

$$\begin{aligned}\mathsf{wlp}^{\#}_{\Gamma} &\overset{def}{=} \lambda\,\varphi \in \mathcal{F}_{Pred}\,.\ \bigvee\{\,\psi \in \mathcal{F}_{Pred} \mid \psi \models_{\Gamma} \mathsf{wlp}(\varphi)\,\}\\ \mathsf{post}^{\#}_{\Gamma} &\overset{def}{=} \lambda\,\varphi \in \mathcal{F}_{Pred}\,.\ \bigwedge\{\,\psi \in \mathcal{F}_{Pred} \mid \varphi \models_{\Gamma} \mathsf{wlp}(\psi)\,\}.\end{aligned}$$

The definition of the context-sensitive abstract operators and the general properties of Galois connections ensure that $\mathsf{wlp}^{\#}_{\Gamma}$ and $\mathsf{post}^{\#}_{\Gamma}$ form a Galois connection between the posets $\langle \mathcal{F}_{Pred}, \models_{\Gamma}\rangle$ and $\langle \mathcal{F}_{Pred}, \models_{\mathsf{post}(\gamma(\Gamma))}\rangle$.

Proposition 3.2.3. *For a given context Γ, the context-sensitive abstract operators have the following properties:*

(i) $\mathsf{post}^{\#}_{\Gamma}$ *and* $\mathsf{wlp}^{\#}_{\Gamma}$ *form a Galois connection between the two posets* $\langle \mathcal{F}_{Pred}, \models_{\Gamma}\rangle$ *and* $\langle \mathcal{F}_{Pred}, \models_{\mathsf{post}(\gamma(\Gamma))}\rangle$*, formally:*

$$\forall \varphi, \psi \in \mathcal{F}_{Pred} :\ \mathsf{post}^{\#}_{\Gamma}(\varphi) \models_{\mathsf{post}(\gamma(\Gamma))} \psi \iff \varphi \models_{\Gamma} \mathsf{wlp}^{\#}_{\Gamma}(\psi),$$

(ii) $\mathsf{post}^{\#}_{\Gamma}$ *and* $\mathsf{wlp}^{\#}_{\Gamma}$ *are monotone,*

(iii) $\mathsf{post}^{\#}_{\Gamma} \circ \mathsf{wlp}^{\#}_{\Gamma}$ *is reductive,*

(iv) $\mathsf{wlp}^{\#}_{\Gamma} \circ \mathsf{post}^{\#}_{\Gamma}$ *is extensive,*

(v) $\mathsf{post}^{\#}_{\Gamma}$ *distributes over disjunctions,*

(vi) $\mathsf{wlp}^{\#}_{\Gamma}$ *distributes over conjunctions.*

3.2.3 Cartesian Abstraction and Cartesian Post

In predicate abstraction [12], elements of the abstract domain are given by disjunctions of conjunctions of abstraction predicates that are isomorphic to sets of bit-vectors. For the computation of the precise abstract post operator on sets of bit-vectors one has to check exhaustively for each possible bit-vector whether it occurs in the post or not. This results in an exponential lower bound for the complexity of the best abstract post. Hence, there is an analogous problem to the one we have to solve in our setting.

In predicate abstraction the problem of an exponential lower bound for the complexity of $\mathsf{post}^{\#}$ is addressed by applying an additional *Cartesian abstraction* [1]. This approach is effectively used in predicate abstraction based software model checkers such as SLAM [3] and BLAST [13].

Cartesian abstraction is an abstraction for vector domains. Given a vector domain D with

$$D = D_1 \times \cdots \times D_n$$

it over-approximates a set of vectors $V \subseteq D$ by ignoring the dependencies between the components of each vector in V, i.e. the Cartesian approximation of V is described by the Cartesian product:

$$\gamma_{\mathsf{Cart}}(\alpha_{\mathsf{Cart}}(V)) = \Pi_1(V) \times \cdots \times \Pi_n(V)$$

where the projections $\Pi_i(V)$ are given by:

$$\Pi_i(V) = \{\, v_i \in D_i \mid (v_1, \ldots, v_i, \ldots, v_n) \in V \,\}.$$

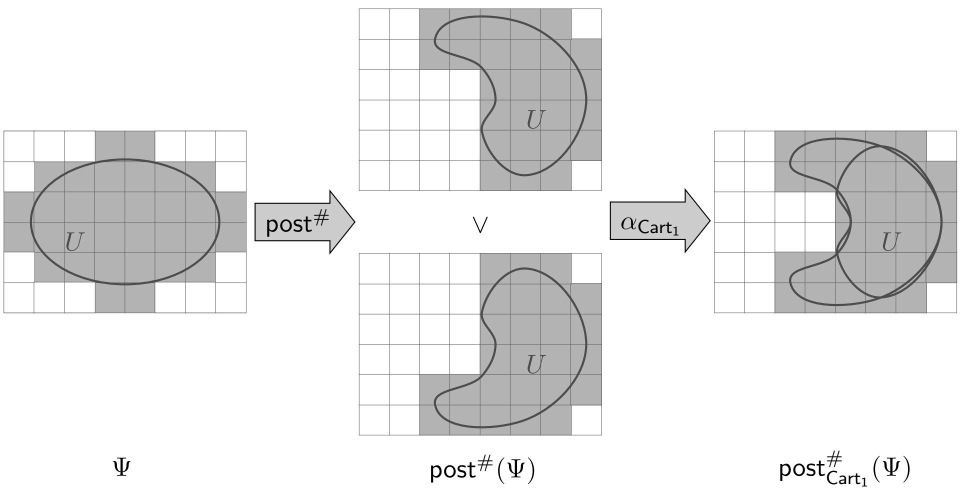


Figure 3.4: Application of $\mathsf{post}^{\#}$ to a single abstract store Ψ and the approximation under α_{Cart_1}.

For a set of bit-vectors represented by a formula ψ, its Cartesian abstraction $\alpha_{\mathsf{Cart}}(\psi)$ is given by the smallest conjunct over abstraction predicates that is implied by ψ. The operator that results from composing Cartesian abstraction with $\mathsf{post}^{\#}$ corresponds to a Boolean program over the abstraction predicates. This abstract post operator can be computed efficiently in practice.

In our setting, elements of the abstract domain are disjunctions of abstract stores, where abstract stores are Boolean combinations of node predicates. In other words, we are dealing with sets of sets of bit-vectors. We use a two-step Cartesian abstraction for the approximation of $\mathsf{post}^{\#}$. One abstraction step for each set hierarchy.

The best abstract post operator $\mathsf{post}^{\#}$ is a join-morphism[2], i.e. distributes over disjunctions. Hence, computing $\mathsf{post}^{\#}$ for a disjunction of abstract stores can be accomplished by computing $\mathsf{post}^{\#}$ for each abstract store individually. Thus, we just need to consider the case, where $\mathsf{post}^{\#}$ is applied to a single abstract store Ψ:

$$\Psi = \forall v.\psi.$$

Even if we apply $\mathsf{post}^{\#}$ to a single abstract store Ψ, its image under $\mathsf{post}^{\#}$ will in general be a disjunction of abstract stores rather then a single abstract store. The first step is to abstract a disjunction of abstract stores by a single abstract store. Formally, this corresponds to the application of the abstraction function α_{Cart_1}.

Definition 3.2.4 (First Cartesian Abstraction α_{Cart_1}). *The first Cartesian abstraction α_{Cart_1} that approximates a disjunction of abstract stores by a single abstract store is defined by:*

$$\begin{array}{lcl} \alpha_{\mathsf{Cart}_1} & \in & AbsDom \to AbsStore \\ \alpha_{\mathsf{Cart}_1} & \overset{def}{=} & \lambda\,\Psi\,.\,\forall v.\bigwedge\{\,\psi \in \mathcal{F}_{Pred} \mid \Psi \models \forall v.\psi\,\}. \end{array}$$

The additional abstraction function α_{Cart_1} is applied to the image of Ψ under $\mathsf{post}^{\#}$. As illustrated in Figure 3.4, the abstraction merges all abstract stores in the image into one single abstract store. The composition of α_{Cart_1} and $\mathsf{post}^{\#}$ gives us our first approximation of the best abstract post operator.

[2]The functions α, post and γ are all join-morphisms. Hence, their composition is one, too.

Definition 3.2.5. *The operator* $\mathsf{post}^{\#}_{\mathsf{Cart}_1}$ *is given by the composition of* α_{Cart_1} *and* $\mathsf{post}^{\#}$*:*

$$\begin{array}{rcl} \mathsf{post}^{\#}_{\mathsf{Cart}_1} & \in & \mathit{AbsStore} \to \mathit{AbsStore} \\ \mathsf{post}^{\#}_{\mathsf{Cart}_1} & = & \alpha_{\mathsf{Cart}_1} \circ \mathsf{post}^{\#} \,. \end{array}$$

The operator $\mathsf{post}^{\#}_{\mathsf{Cart}}$ can be characterized without referring to $\mathsf{post}^{\#}$ explicitly. Applying the first Cartesian abstraction can be seen as a kind of localization of the abstract post operator. We construct the abstract store that results from the abstraction of all disjuncts in the image of Ψ under $\mathsf{post}^{\#}$ by computing post locally for each abstract node in Ψ. That means, for each abstract node P in Ψ, we compute all abstract nodes in disjuncts of $\mathsf{post}^{\#}(\Psi)$ that cover concrete nodes represented by P. This operation exactly corresponds to the context-sensitive abstract post for Ψ applied to the abstract nodes in Ψ.

Proposition 3.2.6. *Let* $\Psi = \forall v.\psi$ *be an abstract store. The image of* Ψ *under* $\mathsf{post}^{\#}_{\mathsf{Cart}_1}$ *is obtained by applying the context-sensitive post operator for* Ψ *to* ψ*:*

$$\mathsf{post}^{\#}_{\mathsf{Cart}_1}(\Psi) \models\!\!\!\!\dashv \forall v.\, \mathsf{post}^{\#}_{\Psi}(\psi).$$

Now we express the image of an abstract store under $\mathsf{post}^{\#}_{\mathsf{Cart}_1}$ in terms of a post operator applied to the abstract nodes in the abstract store. However, this localized post operator still depends on its context Ψ. The information available in a single abstract node is not always sufficient to compute its covering in the post of Ψ precisely. Though, we will see later that the context information we actually need is of a very restricted kind. Usually, it is possible to express the additional constraints on the global state in terms of an abstract store over the node predicates that are already used to obtain the abstraction.

Let Ψ be given as a disjunction of conjunctions of node predicates in *Pred*:

$$\Psi = \forall v.\, \bigvee_i P_i.$$

According to Proposition 3.2.3, the context-sensitive abstract post distributes over disjunctions of formulas in $\mathcal{F}_{\mathit{Pred}}$. Hence, the first Cartesian abstraction of $\mathsf{post}^{\#}$ is obtained by applying the context-sensitive post for Ψ to each abstract node P_i.

$$\mathsf{post}^{\#}_{\mathsf{Cart}_1}(\Psi) = \forall v.\, \bigvee_i \mathsf{post}^{\#}_{\Psi}(P_i).$$

However, computing $\mathsf{post}^{\#}_{\mathsf{Cart}_1}$ is still an expensive operation. The result of the context-sensitive post operator applied to an abstract node will in general be a disjunction of abstract nodes. We face the same problem as before. We would have to look at all 2^n monomials over node predicates, in order to compute the precise image of an abstract node P_i under the context-sensitive post operator $\mathsf{post}^{\#}_{\Psi}$. Therefore, we introduce a second Cartesian abstraction that we compose with $\mathsf{post}^{\#}_{\Psi}$.

Definition 3.2.7 (Second Cartesian Abstraction α_{Cart_2}). *The second Cartesian abstraction* α_{Cart_2} *that approximates a disjunction of abstract nodes by a single abstract node is defined by:*

$$\begin{array}{rcl} \alpha_{\mathsf{Cart}_2} & \in & \mathcal{F}_{\mathit{Pred}} \to \mathcal{F}_{\mathit{Pred}} \\ \alpha_{\mathsf{Cart}_2} & \overset{\mathit{def}}{=} & \lambda\, \varphi \,.\, \bigwedge \{\, p \in \mathit{Pred} \mid \varphi \models_{\mathsf{post}(\gamma(\Psi))} p \,\}. \end{array}$$

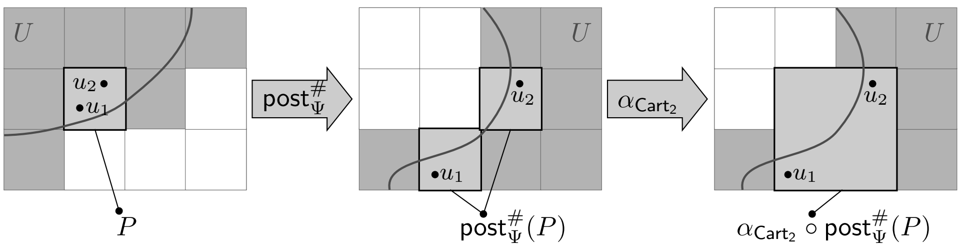

U
u_2
u_1
P
$\mathsf{post}^\#_\Psi$
U
u_2
u_1
$\mathsf{post}^\#_\Psi(P)$
α_{Cart_2}
U
u_2
u_1
$\alpha_{\mathsf{Cart}_2} \circ \mathsf{post}^\#_\Psi(P)$

Figure 3.5: Application of $\mathsf{post}^\#_\Psi$ to a single abstract node P and the approximation under α_{Cart_2}.

As illustrated in Figure 3.5, the function α_{Cart_2} approximates a disjunction of abstract nodes by one single abstract node. This second Cartesian abstraction allows us to define the final abstraction of $\mathsf{post}^\#$.

Definition 3.2.8 (Cartesian Post). *Let $\Psi = \forall v. \bigvee_i P_i(v)$ be an abstract store. The Cartesian post of Ψ is defined as follows:*

$$\mathsf{post}^\#_{\mathsf{Cart}}(\Psi) \quad \overset{def}{=} \quad \forall v. \bigvee_i \alpha_{\mathsf{Cart}_2} \circ \mathsf{post}^\#_\Psi(P_i).$$

We extend the Cartesian post to a function on $AbsDom$ in the canonical way by pushing it over disjunctions of abstract stores.

Theorem 3.2.9 (Soundness of Cartesian Post). *The operator $\mathsf{post}^\#_{\mathsf{Cart}}$ is an approximation of $\mathsf{post}^\#$:*

$$\forall \Psi \in AbsStore : \ \mathsf{post}^\#(\Psi) \models \mathsf{post}^\#_{\mathsf{Cart}}(\Psi).$$

Finally, the only remaining issue is the occurrence of the context-sensitive abstract post in the characterization of $\mathsf{post}^\#_{\mathsf{Cart}}$. Since the concrete post operator can be characterized in terms of predicate-update formulas, i.e. weakest liberal preconditions of predicate symbols, it is convenient to describe $\mathsf{post}^\#_{\mathsf{Cart}}$ using the context-sensitive abstract wlp. We use that $\mathsf{post}^\#_\Psi$ and $\mathsf{wlp}^\#_\Psi$ form a Galois connection, which gives us the final characterization of $\mathsf{post}^\#_{\mathsf{Cart}}$.

Theorem 3.2.10 (Characterization of Cartesian Post). *Let $\Psi = \forall v. \bigvee_i P_i$ be an abstract store. The Cartesian post of Ψ is characterized as follows:*

$$\mathsf{post}^\#_{\mathsf{Cart}}(\Psi) = \forall v. \bigvee_i \bigwedge \{\, p \in Pred \mid P_i \models_\Psi \mathsf{wlp}^\#_\Psi(p) \,\}.$$

The image of an abstract store Ψ under $\mathsf{post}^\#_{\mathsf{Cart}}$ is constructed by collecting, for each abstract node P_i in Ψ, those node predicates that are satisfied in the post of Ψ for the nodes covered by P_i. That means, if n is the number of unsigned node predicates in $Pred$ then for each abstract node P_i, we have to check $2n$ entailments of the form:

$$P_i \models_\Psi \mathsf{wlp}^\#_\Psi(p).$$

Since the number of abstract nodes in Ψ is at most exponential in the number of node predicates, we need $2n \cdot \mathcal{O}(2^n)$ entailment checks for the computation of $\mathsf{post}^\#_{\mathsf{Cart}}$. The complexity of computing the image of $\mathsf{post}^\#_{\mathsf{Cart}}$ for a single abstract store is therefore *only* at most exponential in the number of node predicates. This will be reasonably efficient, if one uses appropriate symbolic data structures for implementation.

Summarizing the above result, we have to provide a set of node predicates $Pred$ and the context-sensitive abstract wlp for each node predicates and atomic command, in order to instantiate the framework.

3.2.4 Boolean Heap Programs

In analogy to predicate abstraction, we can give a source-to-source transformation of the concrete program into a Boolean program, such that the post operator associated with this Boolean program corresponds to the Cartesian post. We call the resulting program a *Boolean heap program*.

The state of a Boolean heap program is given by an abstract store Ψ and a program location. Abstract stores are isomorphic to sets of bit-vectors, i.e. an abstract store Ψ with

$$\Psi = \forall v. \bigvee_i P_i$$

can be represented as a set of bit-vectors V_Ψ, where each of the bit-vectors $v_i \in V_\Psi$ corresponds to an abstract node P_i in Ψ.

The Boolean heap program is obtained from the concrete program by replacing each atomic command c with the predicate-updates of the components of all bit-vectors in V_Ψ:

```
                      for each bit-vector $v$ in $V_\Psi$ do
                        for each $p$ in $Pred$ do
                          if $v \models_\Psi \mathsf{wlp}^\#_{c,\Psi}(p)$ then
                            $v.p$ := true
c      $\leadsto$                else if $v \models_\Psi \mathsf{wlp}^\#_{c,\Psi}(\neg p)$ then
                            $v.p$ := false
                          else
                            $v.p$ := *
```

Since a single state of a Boolean heap program is given by a set of bit-vectors, corresponding to our abstract domain, we need data structures that canonically represent sets of sets of bit-vectors for implementation. A possible choice for such a data structure are nondeterministic BDDs [10], a generalization of BDDs [4].

3.2.5 Precision of Cartesian Post

In this section we characterize under which conditions the Cartesian post operator $\mathsf{post}^\#_{\mathsf{Cart}}$ does not lose precision with respect to the best abstract post operator $\mathsf{post}^\#$. This means, we want to analyze under which conditions the two operators coincide. Cartesian abstraction affects precision whenever a single abstract node is mapped to a set of more than one abstract node under the operator $\mathsf{post}^\#_\Psi$. Conversely, if any monomial over node predicates is again mapped to a single monomial, Cartesian abstraction does not introduce an additional loss of precision. In such a case we call $\mathsf{post}^\#$ deterministic.

Definition 3.2.11. *Let $\Psi = \forall v. \bigvee_i P_i$ be an abstract store, where the P_i are monomials. The operator $\mathsf{post}^\#$ is called **deterministic with respect to** Ψ if every monomial P_i is mapped to one monomial in the post of Ψ, i.e.:*

$$\mathsf{post}^\#(\forall v. \bigvee_i P_i) \models\!\mid \forall v. \bigvee_i \mathsf{post}^\#_\Psi(P_i) \quad \textit{where for all } i: \ \mathsf{post}^\#_\Psi(P_i) \textit{ is a monomial.}$$

*We call $\mathsf{post}^\#$ **deterministic** if for all stores S, $\mathsf{post}^\#$ is deterministic with respect to $\alpha(S)$.*

Note that the general notion of a deterministic $\mathsf{post}^\#$ only requires $\mathsf{post}^\#$ to be deterministic with respect to abstract stores that actually occur as the image of some concrete store under α and not with respect to arbitrary abstract stores.

Proposition 3.2.12. *Let Ψ be an abstract store. If* $\mathsf{post}^{\#}$ *is deterministic with respect to Ψ then* $\mathsf{post}^{\#}_{\mathsf{Cart}}$ *does not lose precision with respect to* $\mathsf{post}^{\#}$*, i.e.*

$$\mathsf{post}^{\#}(\Psi) \models\!\mid \mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi).$$

Whether $\mathsf{post}^{\#}$ is deterministic with respect to some abstract store $\alpha(S)$ depends on the node predicates that induce the abstraction. If the operator wlp is again precisely expressible in terms of the chosen node predicates, the deterministic behavior of the concrete transition system is preserved in its abstraction.

A fundamental observation is that $\mathsf{post}^{\#}$ and $\mathsf{post}^{\#}_{\mathsf{Cart}}$ coincide on abstract store $\alpha(S)$ if the context-sensitive abstract wlp and the precise wlp coincide with respect to the weakened entailment relation $\models_{\alpha(S)}$.

Proposition 3.2.13. *Let S be a store. The operator* $\mathsf{post}^{\#}$ *is deterministic with respect to $\alpha(S)$ if and only if for all node predicates p in Pred we have:*

$$\mathsf{wlp}(p) \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p).$$

Definition 3.2.14 (Closeness under Weakest Liberal Preconditions). *A set of node predicates $\mathcal{P}$ is said to be* ***closed*** *under* wlp *if for every node predicate p in $\mathcal{P}$ there is some finite subset Pred of $\mathcal{P}$ such that:*

$$\forall S \in \mathit{Store} : \mathsf{wlp}(p) \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p).$$

Proposition 3.2.15. $\mathsf{post}^{\#}$ *is deterministic if and only if the set of node predicates Pred is closed under* wlp.

Corollary 3.2.16. *If the set of node predicates Pred is closed under* wlp *then* $\mathsf{post}^{\#}$ *and* $\mathsf{post}^{\#}_{\mathsf{Cart}}$ *coincide.*

As we will see in Chapter 4, despite of trivial cases, finite sets of node predicates are not closed under wlp for all atomic commands. That means, for most finite sets of node predicates *Pred*, there will always be some command such that the Cartesian post will lose precision with respect to $\mathsf{post}^{\#}$. Nevertheless, the closeness property can be seen as a quality measure. Having some infinite class of node predicates with this property is a prerequisite for automated abstraction refinement, because it can guide the search for node predicates that increase the precision of the Cartesian post with respect to $\mathsf{post}^{\#}$.

3.2.6 Nondeterminism and Splitting Operators

As we have seen, Cartesian abstraction works well for commands with deterministic $\mathsf{post}^{\#}$. Unfortunately, for particular commands, $\mathsf{post}^{\#}$ is inherent nondeterministcal. The additional loss of precision caused by the Cartesian post with respect to a nondeterministic $\mathsf{post}^{\#}$ cannot always be tolerated. Each of the two Cartesian abstraction steps in the Cartesian post operator has its own potential loss of precision. In order to get a better understanding of this problem, we illustrate this at the following two examples.

Example. Recall the abstract store given by:

$$\begin{aligned}\Psi_1 = \forall v.\quad & [x(v) \wedge \neg null(v) \wedge r_x(v)] \vee [\neg p_x(v) \wedge \neg null(v) \wedge r_x(v)] \\ & \vee\ [\neg x(v) \wedge null(v) \wedge r_x(v)].\end{aligned}$$

As explained earlier, Ψ_1 represents the set of all stores containing an acyclic singly-linked list with at least one element and whose head is pointed to by program variable x. Now consider command c : x = x->n setting program variable x to the n-successor of the node that x points to. Since we know the concretisation of Ψ_1, we can compute its image under the best abstract post manually, which gives us the following disjunction of abstract stores:

$$\begin{aligned}
\Psi_1' &\overset{def}{=} \mathsf{post}_c^{\#}(\Psi_1)\\
&= \forall v.\ [\neg x(v) \wedge \neg null(v) \wedge \neg r_x(v)] \vee [x(v) \wedge null(v) \wedge r_x(v)] && (1)\\
&\quad \vee \forall v.\ [\neg x(v) \wedge \neg null(v) \wedge \neg r_x(v)] \vee [x(v) \wedge \neg null(v) \wedge r_x(v)] && (2)\\
&\qquad \vee [\neg x(v) \wedge null(v) \wedge r_x(v)]\\
&\quad \vee \forall v.\ [\neg x(v) \wedge \neg null(v) \wedge \neg r_x(v)] \vee [x(v) \wedge \neg null(v) \wedge r_x(v)] && (3)\\
&\qquad \vee [\neg x(v) \wedge \neg null(v) \wedge r_x(v)] \vee [\neg x(v) \wedge null(v) \wedge r_x(v)].
\end{aligned}$$

Abstract store (1) results from a store with a one element list, abstract store (2) from a store with a list of exactly two elements, and abstract store (3) from all stores containing lists with more than two elements. In contrast, the image of Ψ_1 under the Cartesian post is as follows:

$$\begin{aligned}
\Psi_1'' &\overset{def}{=} \mathsf{post}_{c,\mathsf{Cart}}^{\#}(\Psi_1)\\
&= \forall v.[\neg x(v) \wedge \neg null(v) \wedge \neg r_x(v)] \vee [\neg null(v) \wedge r_x(v)] \vee [null(v) \wedge r_x(v)].
\end{aligned}$$

The first Cartesian abstraction of the Cartesian post merges all abstract stores that occur in Ψ_1' into a single abstract store Ψ_1''. The main problem with this approximation is that Ψ_1'' now contains more than one monomial in which $x(v)$ occurs positively, namely the monomials:

$$x(v) \wedge \neg null(v) \wedge r_x(v) \;\text{ and }\; x(v) \wedge null(v) \wedge r_x(v).$$

In order to get reasonable precise results of the analysis, we need at least precise information about the nodes the program variables point to. This means in particular that each abstract store should only contain one monomial in which $x(v)$ occurs positively.

A slightly different problem is caused by the second step of Cartesian abstraction. In this step, the disjunction of all complete abstract nodes that precisely cover the post of a single abstract node are merged into just one abstract node.

Example. Consider the modified abstract store Ψ_2 where the node predicate r_x is replaced by the node predicate r_x^+, expressing that a node is reachable from x in at least one step:

$$\begin{aligned}
\Psi_2 = \forall v.\ &[x(v) \wedge \neg null(v) \wedge \neg r_x^+(v)] \vee [\neg x(v) \wedge \neg null(v) \wedge r_x^+(v)]\\
\vee\ &[\neg x(v) \wedge null(v) \wedge r_x^+(v)].
\end{aligned}$$

Applying $\mathsf{post}_{\mathsf{Cart}}^{\#}$ for command c to Ψ_2 results in the following abstract store:

$$\begin{aligned}
\Psi_2' \overset{def}{=} \mathsf{post}_{c,\mathsf{Cart}}^{\#}(\Psi_2) &= \forall v.[\neg x(v) \wedge \neg null(v) \wedge \neg r_x^+(v)] \vee \neg null(v) \vee null(v)\\
&\models\!\!\!\!\dashv \mathsf{true}.
\end{aligned}$$

The node predicate r_x^+ does not hold for nodes pointed to by program variable x. For the second and third disjunct in Ψ_2 there are nodes for which $x(v)$ may hold and others for which $x(v)$ may not hold in the post of stores satisfying Ψ. Since Cartesian abstraction approximates the disjunction of the corresponding abstract nodes by a single conjunction, we lose the precise information about whether r_x^+ holds or not.

In the above example the image of the precise context-sensitive abstract post for a single abstract node is a disjunction of abstract nodes. In the context of shape graphs this corresponds to the result of splitting a summary node in the shape graph into separated nodes. In [26] this splitting is also referred to as node materialization.

In both cases above, the loss of precision of the Cartesian post with respect to the best abstract post is connected to nondeterminism in the abstract transition system. In order to respect this nondeterminism, we want to relax Cartesian abstraction and allow splitting of abstract nodes and abstract stores in certain situations.

As we have seen in Proposition 3.2.13, determinism in the abstract system is closely connected to the precision of the context-sensitive abstract wlp. Whenever for some node predicate p the precise weakest liberal precondition $\mathsf{wlp}_c(p)$ is not expressible in the abstraction, a splitting may take place in the image under the best abstract post. In order to allow splitting on some node predicate p, we proceed as follows:

(1) temporarily add $\mathsf{wlp}_c(p)$ to the set of abstraction predicates Pred,

(2) translate Ψ from the original abstract domain to the abstract domain over the extended set of node predicates,

(3) compute the Cartesian post on the extended abstract domain,

(4) and translate the result back to the original abstract domain without node predicate $\mathsf{wlp}_c(p)$.

By translating Ψ from the original abstract domain to the extended abstract domain, it is guaranteed that each abstract node in the resulting abstract stores either entails $\mathsf{wlp}_c(p)$ or $\mathsf{wlp}_c(\neg p)$. Thus, the splitting of abstract nodes[3] according to the truth value of p in the successor stores already takes place before we actually compute the image under the Cartesian post.

The fact that $\mathsf{wlp}_c(p)$ is explicitly added to the set of abstraction predicates ensures that the context-sensitive abstract wlp exactly corresponds to $\mathsf{wlp}_c(p)$. Consequently, the splitting of abstract stores and abstract nodes is preserved under the Cartesian post on the extended abstract domain.

The splitting operation and the resulting abstract post operator that respects this splitting are formally captured by the following two definitions.

Definition 3.2.17 (Splitting Operators). *A* ***splitting operator*** $\mathsf{split}_c[\mathcal{P}]$ *for an atomic command c and node predicates $\mathcal{P} \subseteq \mathit{Pred}$ is an operator satisfying the following conditions:*

- $\mathsf{split}_c[\mathcal{P}] \in \mathit{AbsStore}[\mathit{Pred}] \to \mathit{AbsDom}[\mathit{Pred} \cup \mathsf{wlp}_c(\mathcal{P})]$
- $\forall \Psi \in \mathit{AbsStore}[\mathit{Pred}] : \mathsf{split}_c[\mathcal{P}]\ \Psi \models \Psi,$
- $\forall \Psi \in \mathit{AbsStore}[\mathit{Pred}] : \alpha[\mathit{Pred} \cup (\mathsf{wlp}_c\, \mathcal{P})](\gamma\ \Psi) \models \mathsf{split}_c[\mathcal{P}]\ \Psi,$

where: $\mathsf{wlp}_c(\mathcal{P}) = \{\, \mathsf{wlp}_c(p) \mid p \in \mathcal{P} \,\}$.

We call the splitting operator $\mathsf{split}^{\#}_c[\mathcal{P}]$ *satisfying:*

$$\mathsf{split}^{\#}_c[\mathcal{P}] = \alpha[\mathit{Pred} \cup \mathsf{wlp}_c(\mathcal{P})] \circ \gamma$$

the ***most precise splitting operator*** *for c and $\mathcal{P}$.*

[3]and the splitting of the abstract store itself

Definition 3.2.18 (Cartesian Post with Splitting). *The **Cartesian post operator with splitting** for splitting operator* $\mathsf{split}_c[\mathcal{P}]$ *is defined by:*

$$\begin{aligned}
\mathsf{post}^{\#}_{c,\mathsf{split}_c[\mathcal{P}]} &\in \mathit{AbsStore}[\mathit{Pred}] \rightarrow \mathit{AbsDom}[\mathit{Pred}] \\
\mathsf{post}^{\#}_{c,\mathsf{split}_c[\mathcal{P}]} &= \alpha[\mathit{Pred}] \circ \gamma \circ \mathsf{post}^{\#}_{c,\mathsf{Cart}}[\mathit{Pred} \cup \mathsf{wlp}_c(\mathcal{P})] \circ \mathsf{split}_c[\mathcal{P}].
\end{aligned}$$

Note that in contrast to the most precise splitting operator $\mathsf{split}^{\#}_c[\mathcal{P}]$ it is always simple to compute the function

$$\alpha[\mathit{Pred}] \circ \gamma \in \mathit{AbsDom}[\mathit{Pred} \cup \mathsf{wlp}_c(\mathcal{P})] \rightarrow \mathit{AbsDom}[\mathit{Pred}]$$

symbolically. Given some abstract value Ψ in $\mathit{AbsDom}[\mathit{Pred} \cup \mathsf{wlp}_c(\mathcal{P})]$, we just have to project the added node predicates in $\mathsf{wlp}_c(\mathcal{P})$, in order to obtain the image of Ψ under $\alpha[\mathit{Pred}] \circ \gamma$.

Definition 3.2.17 gives rise to a whole spectrum of possible splitting operators with different levels of precision. This spectrum ranges from the identity function, i.e. no splitting at all, to the most precise splitting operator. The possibility to choose among all these operators gives us the freedom to fine-tune the ratio between efficiency and precision of the resulting analysis. However, for any splitting operator soundness of the corresponding Cartesian post with splitting is guaranteed.

Proposition 3.2.19 (Soundness of Cartesian Post with Splitting). *Let* $\mathsf{split}_c[\mathcal{P}]$ *be a splitting operator for* $\mathcal{P}$ *and command c. The Cartesian post operator with splitting* $\mathsf{post}^{\#}_{c,\mathsf{split}_c[\mathcal{P}]}$ *is an approximation of* $\mathsf{post}^{\#}_c$ *on* $\mathit{AbsDom}[\mathit{Pred}]$*:*

$$\forall \Psi \in \mathit{AbsStore}[\mathit{Pred}] : \mathsf{post}^{\#}_c(\Psi) \models \mathsf{post}^{\#}_{c,\mathsf{split}_c[\mathcal{P}]}(\Psi).$$

In the following, we want to point out two observations about the two *border cases* of the spectrum of possible splitting operators. At first, consider the most precise splitting operator $\mathsf{split}^{\#}_c[\mathit{Pred}]$ that splits on the whole set of abstraction predicates Pred. It is not hard to see that this operator implements the best abstract post operator $\mathsf{post}^{\#}_c[\mathit{Pred}]$. This is simply due to the fact that the image of an abstract store Ψ under $\mathsf{split}^{\#}_c[\mathit{Pred}]$ represents at the same time both Ψ itself and its image under $\mathsf{post}^{\#}_c[\mathit{Pred}]$. Consequently, in this case the Cartesian post operator with splitting and the best abstract post operator coincide.

Proposition 3.2.20. *The Cartesian post with splitting for the most precise splitting operator* $\mathsf{split}^{\#}_c[\mathit{Pred}]$ *coincides with the best abstract post operator on the abstract domain* $\mathit{AbsDom}[\mathit{Pred}]$*.*

On the other hand, if $\mathsf{post}^{\#}_c$ is deterministic then for every possible splitting operator the corresponding Cartesian post with splitting will be most precise. This observation is a consequence of Proposition 3.2.12 and holds in particular, if the used splitting operator is just the identity function.

Proposition 3.2.21. *Let* $\mathsf{split}_c[\mathcal{P}]$ *be a splitting operator. If* $\mathsf{post}^{\#}_c$ *is deterministic then* $\mathsf{post}^{\#}_c$ *and the Cartesian post with splitting for* $\mathsf{split}_c[\mathcal{P}]$ *coincide.*

A concrete example for a splitting operator can be found in the case study that we discuss in Chapter 5.

Chapter 4

Modal Node Predicates

The construction of Boolean heap programs requires the identification of a suitable class of node predicates that induces the abstraction. There are some requirements that such a class of node predicates should ideally fulfil:

(1) It should be expressible enough to specify properties we are interested in: reachability, sharing, etc.

(2) It should not be too expressive, i.e. the satisfiability problem should be decidable.

(3) It should be closed under weakest liberal preconditions.

Requirements (2) and (3) are main prerequisites for automation. In Section 3.2.5 we already argued that (3) is needed for automated abstraction refinement. Requirement (2) allows checking entailments, which is needed for the computation of context-sensitive abstract weakest liberal preconditions and, hence, the construction of Boolean heap programs.

If we for instance consider all node predicates that are expressible in $\mathsf{FO}^{\mathsf{TC}}$ then this class satisfies (1) and (3), but as an extension of first-order logic, is undecidable. In the following, we restrict to node predicates that are sufficiently expressive to describe reachability properties for linked data structures. We propose *modal node predicates* and give first results regarding decidability and closeness under weakest liberal preconditions.

4.1 Motivation

For motivation, we consider again the reachability property r_x expressing that a node is reachable from program variable x by following any number of n pointer fields:

$$r_x \overset{def}{=} x(v) \lor \exists\, v'.x(v') \land n^+(v', v).$$

Now consider the command c : `z->n = y` that destructively updates the n-pointer field of the node pointed to by z. Assume that r_x did hold for some node u before c is executed. In order to check whether r_x still holds for u after executing c, we particularly need the information whether the node pointed to by z was lying on the n-path from x to the node u. If z was not lying on that path then surely r_x will still hold for u after c is executed.

Thus, when we deal with reachability, it is not sufficient to be able to express the existence of a path between two nodes in terms of the given node predicates. It

$$
\begin{array}{rcl}
p.\langle R\rangle(v) & \overset{def}{=} & \exists\, v'.p(v) \wedge r(v', v) \\
\langle R\rangle.p(v) & \overset{def}{=} & \exists\, v'.p(v) \wedge r(v, v') \\
p.\mathsf{U}_{\langle R\rangle}.q(v) & \overset{def}{=} & \exists\, v'.q(v') \wedge r^*[p(v)](v, v') \\
p.\mathsf{S}_{\langle R\rangle}.q(v) & \overset{def}{=} & \exists\, v'.q(v') \wedge r^*[p(v')](v', v) \\
p.\mathsf{U}_{[R]}.q(v) & \overset{def}{=} & \forall v'.\, r^*[\neg q(v)](v, v') \rightarrow \\
 & & \quad q(v') \vee p(v') \wedge \neg r^+[\neg q(v)](v', v') \wedge \exists\, v''.r^+(v', v'') \wedge q(v'') \\
p.\mathsf{S}_{[R]}.q(v) & \overset{def}{=} & \forall v'.\, r^*[\neg q(v)](v', v) \rightarrow \\
 & & \quad q(v') \vee p(v') \wedge \neg r^+[\neg q(v)](v', v') \wedge \exists\, v''.r^+(v'', v') \wedge q(v'')
\end{array}
$$

where

$$
\begin{array}{rcl}
r(v, v') & \overset{def}{=} & \bigvee_{n \in R} n(v, v') \\
r^+[\varphi(v)](v, v') & \overset{def}{=} & (\mathsf{TC}\, v, v'.r(v, v') \wedge \varphi(v))(v, v') \\
r^+[\varphi(v')](v, v') & \overset{def}{=} & (\mathsf{TC}\, v, v'.r(v, v') \wedge \varphi(v'))(v, v') \\
r^*[\varphi](v, v') & \overset{def}{=} & r^+[\varphi](v, v') \vee v \approx v'
\end{array}
$$

Figure 4.1: Semantics of modal node predicates.

should also be possible to express that additional constraints are satisfied on that particular path.

If we think of a store as a Kripke structure, we can express the missing properties in terms of modal operators. In particular, the fact that a node is reachable from x without passing a node pointed to by z can be expressed using the *since* operator from temporal logics with past. Because we construct our node predicate language using operators that have semantics corresponding to modal operators, we call them *modal node predicates*.

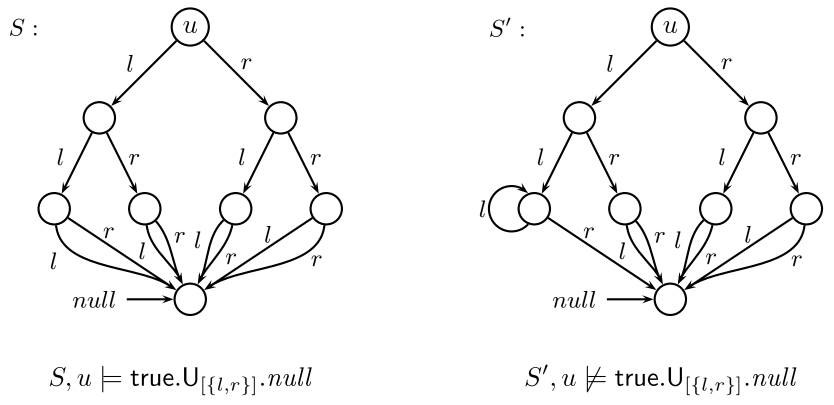
4.2 Syntax and Semantics

Definition 4.2.1 (Syntax of Modal Node Predicates). *Let Var be the set of program variables and let R be a nonempty subset of binary predicate symbols in Σ. The language* $\mathsf{MNP}[R]$ *of modal node predicates over R is defined by the following grammar:*

$$
\begin{array}{lll}
x \in \mathit{Var} \quad & p(v), q(v) \in \mathsf{MNP}[R] ::= & \mathit{null}(v) \mid x(v) \mid \neg p(v) \mid p(v) \wedge q(v) \\
 & & \mid \langle R\rangle.p(v) \mid p.\langle R\rangle(v) \\
 & & \mid p.\mathsf{U}_{\langle R\rangle}.q(v) \mid p.\mathsf{S}_{\langle R\rangle}.q(v) \\
 & & \mid p.\mathsf{U}_{[R]}.q(v) \mid p.\mathsf{S}_{[R]}.q(v)
\end{array}
$$

The modal operators have highest precedence and we skip the variable v whenever this causes no confusion. If R is a singleton containing just one binary predicate symbol n we write $\mathsf{MNP}[n]$ *instead of* $\mathsf{MNP}[\{n\}]$ *and similarly $\langle n\rangle.p$ instead of $\langle\{n\}\rangle.p$, etc. for the modalities.*

The formal semantics of modal node predicates is inductively defined over their structure, as shown in Figure 4.1. Node predicates built from Boolean connectives, as well as atomic node predicates such as x and *null*, are omitted since their semantics is clear.



$S:$
u
l
r
$null$
$S, u \models \mathsf{true}.\mathsf{U}_{[\{l,r\}]}.null$
$S':$
$S', u \not\models \mathsf{true}.\mathsf{U}_{[\{l,r\}]}.null$

Figure 4.2: Two stores S and S', the first containing an acyclic, the second a cyclic binary tree. The two trees can be distinguished using modal node predicates.

Intuitively one can explain the semantics of modal node predicates in the following way. If we think of a store as a Kripke structure, where the underlying transition relation is induced by the union of interpretations of the binary predicate symbols in R then the semantics of $\langle R\rangle.p$, $p.\mathsf{U}_{\langle R\rangle}.q$, and $p.\mathsf{U}_{[R]}.q$ is equivalent to the CTLoperators EX, EU, and AU. Respectively, the modal node predicates $p.\langle R\rangle$, $p.\mathsf{S}_{[R]}.q$, and $p.\mathsf{S}_{\langle R\rangle}.q$ correspond to appropriate past operators.

We want to make the above observation that there is a correspondence between modal node predicates and CTL more precise. In the following, we give a path-based characterization of the semantics of MNP$[R]$ modalities.

Definition 4.2.2. *Let R be nonempty subset of binary predicate symbols in Σ and let $S = \langle U, \iota\rangle$ be a structure over Σ. A sequence $\pi \in \mathbb{N} \to U$ of nodes in U is called an R-path in S, if every two succeeding nodes in the sequence are either connected via a pointer field in R, or they correspond and have no R-successors. Formally, for all $1 \leq i$:*

$$\begin{array}{l}\textit{there exists } n \in R \textit{ such that } (\iota\ n)(\pi(i), \pi(i+1)) = 1\\ \textit{or } \pi(i) = \pi(i+1) \textit{ and for all } n \in R, u \in U : (\iota\ n)(\pi(i), u) = 0.\end{array}$$

Respectively, π is called an R^{-1}-path in S, if for all $1 \leq i$:

$$\begin{array}{l}\textit{there exists } n \in R \textit{ such that } (\iota\ n)(\pi(i+1), \pi(i)) = 1\\ \textit{or } \pi(i) = \pi(i+1) \textit{ and for all } n \in R, u \in U : (\iota\ n)(u, \pi(i)) = 0.\end{array}$$

We say an R-path (R^{-1}-path) π starts in $u \in U$ if $\pi(1) = u$.

Proposition 4.2.3. *Let S be a finite structure over Σ and $u \in U^S$. The modal node predicates $p.\mathsf{U}_{\langle R\rangle}.q$, $p.\mathsf{S}_{\langle R\rangle}.q$, $p.\mathsf{U}_{[R]}.q$, and $p.\mathsf{S}_{[R]}.q$ are characterized as follows:*

- $S, u \models p.\mathsf{U}_{\langle R\rangle}.q(v) \iff$ *there is an R-path π in S starting in u such that:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$
- $S, u \models p.\mathsf{S}_{\langle R\rangle}.q(v) \iff$ *there is an R^{-1}-path π in S starting in u such that:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$
- $S, u \models p.\mathsf{U}_{[R]}.q(v) \iff$ *for all R-paths π in S starting in u:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$
- $S, u \models p.\mathsf{S}_{[R]}.q(v) \iff$ *for all R^{-1}-paths π in S starting in u:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$

$$\begin{array}{rcl@{\qquad}rcl}
[R].p & \overset{def}{=} & \neg\langle R\rangle.(\neg p) & p.[R] & \overset{def}{=} & \neg(\neg p).\langle R\rangle \\
\langle R^*\rangle.p & \overset{def}{=} & \mathsf{true}.\mathsf{U}_{\langle R\rangle}.p & p.\langle R^*\rangle & \overset{def}{=} & \mathsf{true}.\mathsf{S}_{\langle R\rangle}.p \\
[R^*].p & \overset{def}{=} & \mathsf{true}.\mathsf{U}_{[R]}.p & p.[R^*] & \overset{def}{=} & \mathsf{true}.\mathsf{S}_{[R]}.p \\
\langle R^+\rangle.p & \overset{def}{=} & \langle R\rangle.(\langle R^*\rangle.p) & p.\langle R^+\rangle & \overset{def}{=} & (p.\langle R^*\rangle).\langle R\rangle \\
[R^+].p & \overset{def}{=} & [R].([R^*].p) & p.[R^+] & \overset{def}{=} & (p.[R^*]).[R]
\end{array}$$

Figure 4.3: Syntactic abbreviations for modal node predicates.

The following example shows how modal node predicates can be used to express properties of linked data-structures that are related to reachability.

Example. Figure 4.2 shows two stores S and S' containing binary trees. The tree in S is acyclic, whereas the tree in S' contains a cycle. We can express acyclicity of linked data structures in terms of modal node predicates. The root node u in S satisfies the modal node predicate $\mathsf{true}.\mathsf{U}_{[\{l,r\}]}.null$, because all outgoing paths that follow r and l pointers eventually reach $null$. However, the root node in S' does not satisfy $\mathsf{true}.\mathsf{U}_{[\{l,r\}]}.null$, because there is a cyclic path that does not reach $null$.

In addition to the given modal node predicates we define some syntactic abbreviations. Figure 4.3 shows the complete list. For instance the abbreviation $p.\langle n^*\rangle$ expresses that a node is reachable from some node satisfying p following 0 or more n-fields. Thus, the node predicate r_x that we used in previous examples corresponds to the modal node predicate $x.\langle n^*\rangle$. For convenience we will also use Boolean connectives such as disjunction, implication, etc. as abbreviations in the syntax of modal node predicates.

We extend modal node predicates with guarded quantification. This extension allows us to express the context information that we need, in order to precisely characterize weakest liberal preconditions of modal node predicates.

Definition 4.2.4 (Modal Node Predicates with Guarded Quantification). *Let R be a nonempty subset of binary predicate symbols in Π. The set of* modal node predicates with guarded quantification $\mathsf{GMNP}[R]$ *is defined as follows:*

$$\begin{array}{lrcl}
x \in \mathit{Var} & p(v), q(v) \in \mathsf{GMNP}[R] & ::= & x(v) \mid null(v) \mid \neg p(v) \mid p(v) \wedge q(v) \\
& & \mid & \langle R\rangle.p(v) \mid p.\langle R\rangle(v) \\
& & \mid & p.\mathsf{U}_{\langle R\rangle}.q(v) \mid p.\mathsf{S}_{\langle R\rangle}.q(v) \\
& & \mid & p.\mathsf{U}_{[R]}.q(v) \mid p.\mathsf{S}_{[R]}.q(v) \\
& & \mid & \forall v.x(v) \rightarrow p(v)
\end{array}$$

Note that guards in quantified formulas are restricted to program variables.

4.3 Decidability

For the automatic construction of Boolean heap programs, we need to compute context-sensitive abstract wlps for the chosen abstraction predicates. This requires that it is possible to check validity of the context-dependent entailment relation $\models_\Psi$. Checking validity of $\models_\Psi$ can be reduced to the problem whether for a given node predicate p there is a program store S and a node $u \in U^S$, such that p is satisfied by u in S. Consequently, a decision procedure for the restricted satisfiability problem can be used to automate the construction of Boolean heap programs.

In the following, we give a first result concerning decidability of the satisfiability problem for $\mathsf{GMNP}[R]$ and thus $\mathsf{MNP}[R]$. We show that $\mathsf{GMNP}[R]$ can be translated into guarded fixed point logic $\mu\mathsf{GF}$. The general satisfiability problem for $\mu\mathsf{GF}$, i.e. the problem whether a given $\mu\mathsf{GF}$ formula has an arbitrary first-order model, is decidable [11].

Definition 4.3.1 (Guarded Fixed Point Logic $\mu\mathsf{GF}$). *The guarded fragment of first-order logic with fixed points $\mu\mathsf{GF}$ is defined by the grammar:*

$$\begin{aligned}\varphi, \psi \in \mu\mathsf{GF} ::= {} & a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \\ & \mid \forall \overline{v}.g \to \varphi \mid \exists\, \overline{v}.g \wedge \varphi \\ & \mid [\mu W \overline{v}.\, \varphi(W, \overline{v})](\overline{v}) \mid [\nu W \overline{v}.\, \varphi(W, \overline{v})](\overline{v})\end{aligned}$$

where $\overline{v} = v_1, \ldots, v_k$ is a k-tuple of first-order variables, a, g are atoms, g contains all free variables of φ. W is a k-ary relation variable that occurs only positive in $\varphi(W, \overline{v})$ and not in guards and moreover all free variables of $\varphi(W, \overline{v})$ belong to $\overline{v}$.

The semantics of formulas is standard. For fixed point formulas the interpretation of $\varphi(W, \overline{v})$ in a logical structure defines an operator on k-ary relations over the structure's universe. The fact that all uses of the variable W are positive ensures monotonicity of this operator and thereby the existence of its least and greatest fixed points. The formal semantics of fixed point formulas can be found in [11].

Theorem 4.3.2 (Grädel, Walukiewicz). *The satisfiability problem for $\mu\mathsf{GF}$ is decidable.*

The translation from $\mathsf{GMNP}[R]$ to guarded fixed point logic is straightforward. Since the modalities semantically correspond to CTL operators we make use of the standard translation of CTL formulas to the modal mu-calculus.

Definition 4.3.3 (Translation of GMNP to $\mu\mathsf{GF}$). *The function t mapping modal node predicates in $\mathsf{GMNP}[R]$ to $\mu\mathsf{GF}$ is inductively defined as follows:*

$$\begin{aligned}
t &\in \mathsf{GMNP}[R] \to \mu\mathsf{GF} \\
t(a) &= a, \quad \textit{if } a \textit{ is an atom} \\
t(\neg p) &= \neg t(p) \\
t(p \wedge q) &= t(p) \wedge t(q) \\
t(\langle R\rangle.p) &= \bigvee_{n \in R} \exists\, v'.n(v, v') \wedge t(p)(v') \\
t(p.\langle R\rangle) &= \bigvee_{n \in R} \exists\, v'.n(v', v) \wedge t(p)(v') \\
t(p.\mathsf{U}_{\langle R\rangle}.q) &= [\mu W v.\, t(q) \vee t(p) \wedge (\bigvee_{n \in R} \exists\, v'.n(v, v') \wedge W v')](v) \\
t(p.\mathsf{S}_{\langle R\rangle}.q) &= [\mu W v.\, t(q) \vee t(p) \wedge (\bigvee_{n \in R} \exists\, v'.n(v', v) \wedge W v')](v) \\
t(p.\mathsf{U}_{[R]}.q) &= [\mu W v.\, t(q) \vee t(p) \wedge (\bigvee_{n \in R} \exists\, v'.n(v, v')) \wedge (\bigwedge_{n \in R} \forall v'.n(v, v') \to W v')](v) \\
t(p.\mathsf{S}_{[R]}.q) &= [\mu W v.\, t(q) \vee t(p) \wedge (\bigvee_{n \in R} \exists\, v'.n(v, v')) \wedge (\bigwedge_{n \in R} \forall v'.n(v', v) \to W v')](v) \\
t(\forall v.x(v) \to p(v)) &= \forall v.x(v) \to t(p)(v).
\end{aligned}$$

Proposition 4.3.4 (Correctness of Translation). *Let $p \in \mathsf{GMNP}[R]$ then p and $t(p)$ are equivalent on finite structures, i.e. for every finite structure S and $u \in U^S$:*

$$S, u \models p \iff S, u \models t(p).$$

Theorem 4.3.2 establishes decidability of the satisfiability problem for μGF with respect to arbitrary first-order structures. However, we are interested in the satisfiability problem restricted to program stores, i.e. finite structures that satisfy certain integrity constraints. Hence, a decision procedure for μGF cannot directly be used for our needs. However, we have the conjecture that the fragment of μGF that results from the translation of $\mathsf{GMNP}[R]$ has the finite modal property[1]. As a first step, this would allow us to use a decision procedure for the general satisfiability problem as a decision procedure for the satisfiability problem restricted to finite first-order structures.

Unfortunately, the restriction from finite structures to program stores is more sophisticated. We need to translate the integrity constraints that define the set of stores to the logic μGF. It is not obvious whether this is possible for the constraints we are interested in. In particular functionality constraints for binary predicates, as they occur for instance in the integrity formula given in Section 2.3.1, cannot be expressed directly in μGF. Simply adding functionality constraints even to the guarded fragment GF of first-order logic without fixed points results in an undecidable logic. We leave these problems open for future work.

4.4 Closeness under Weakest Liberal Preconditions

In Section 3.2.5 we argued that being closed under weakest liberal preconditions is a prerequisite for the development of automated abstraction refinement procedures for a class of node predicates.

We now show that according to Definition 3.2.14 the class of modal node predicates $\mathsf{MNP}[n]$ for a single binary predicate symbol n is closed under weakest liberal preconditions. That means, for every command c and modal node predicate p, we can find a subset Pred of modal node predicates in $\mathsf{MNP}[n]$ such that the context-sensitive abstract wlp for p and Pred does not lose precision with respect to $\mathsf{wlp}_c(p)$. This guarantees that even if the set Pred does not contain $\mathsf{wlp}_c(p)$ explicitly, the Cartesian post is as precise as if $\mathsf{wlp}_c(p)$ would be contained in the set of abstraction predicates.

We have to consider every atomic command c that we discussed in Section 2.3.2. These commands can be divided into two groups:

- commands that update the values of program variables: `x = t`,
- commands that update the values of pointer fields: `x->n = t`.

We start with the first group of commands. Recall from Section 2.3.2 that the weakest liberal precondition operator wlp_c on formulas is defined as a syntactic substitution. This substitution replaces each predicate symbol p by its predicate update formula p'_c.

For all atomic commands that change the value of a program variable, only the update formula of the unary predicate symbol that corresponds to the updated program variable differs from the predicate symbol itself. Fortunately, as shown in Table 4.1, for those commands the predicate-update formulas correspond syntactically to modal node predicates. Thus, the result of the substitution is guaranteed to be a modal node predicate, too.

Proposition 4.4.1. *For commands c of the form* `x = y`*,* `x = NULL`*, and* `x = y->n`*, the class* $\mathsf{MNP}[n]$ *is closed under* wlp_c*. Formally, for any modal node predicate p there is a*

[1]The full logic μGF does not have finite modal property; see also [11].

c	$p(v)$	$p'_c(v)$
`x = NULL`	$x(v)$	$null(v)$
`x = y`	$x(v)$	$y(v)$
`x = y->n`	$x(v)$	$y.\langle n\rangle(v)$

Table 4.1: Predicate-update formulas expressed in terms of modal node predicates.

finite subset of modal node predicates Pred such that:

$$\forall S \in Store : \mathsf{wlp}^{\#}_{c,\alpha(S)}(p) \models\!\!\!\!\dashv \mathsf{wlp}_c(p).$$

Things get more complicated for the second class of commands. Consider again our example with command c: `z->n = y` and node predicate $x.\langle n^*\rangle$. The problem of expressing $\mathsf{wlp}_c(x.\langle n^*\rangle)$ is that the underlying relation n is changed by the command c. The operator wlp_c replaces the predicate symbol n by its predicate-update formula. Thus, if we compute the weakest liberal precondition according to wlp_c the result does not conform to the syntactic restrictions of modal node predicates, i.e. $\mathsf{wlp}_c(x.\langle n^*\rangle)$ is not contained in $\mathsf{MNP}[n]$. However, we can give an equivalent representation in the class of modal node predicates with guarded quantification $\mathsf{GMNP}[n]$.

The command c may effect the validity of $x.\langle n^*\rangle$ for some node u in two different ways:

- it may destroy an outgoing n-path from u that passed z and made $x.\langle n^*\rangle$ valid in the previous store, but false after executing c, or
- it may create a new outgoing n-path from u that passes z and was not present in the previous store, but makes $x.\langle n^*\rangle$ valid after executing c.

Thus, $x.\langle n^*\rangle$ may be valid for some node after executing c if and only if there already was an n-path from x not passing z and hence this path is not effected by c, or c constructs a new path that starts from x and passes both z and y. These two cases can be expressed in terms of $\mathsf{GMNP}[n]$. The context information about the node pointed to by z that is needed to express the second case precisely is covered by a guarded universal constraint:

$$\begin{aligned}\mathsf{wlp}_c(x.\langle n^*\rangle) \models\!\!\!\!\dashv\ & (\neg z).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg z)(v)\ \vee\ z \wedge x.\langle n^*\rangle \vee \\ & (\neg z).\mathsf{S}_{\langle n\rangle}.(y \wedge \neg z)(v)\ \wedge\ \forall v'.z(v') \rightarrow x.\langle n^*\rangle(v').\end{aligned}$$

We generalize the above example. In a first step we use the extended class $\mathsf{GMNP}[n]$ of modal node predicates with guarded quantification to express the precise wlp for each $\mathsf{MNP}[n]$ predicate. In a second step we eliminate the quantified formulas, in order to obtain again pure modal node predicates. Finally, this will allow us to show that $\mathsf{MNP}[n]$ is closed under wlp_c according to Definition 3.2.14.

We introduce a function f_c mapping modal node predicates to their weakest liberal precondition expressed in $\mathsf{GMNP}[n]$.

Definition 4.4.2. *Let c be a command of the form* `x->n = y`. *The function f_c is induc-*

tively defined on the structure of modal node predicates as follows:

$$\begin{array}{rcl}
f_c & \in & \mathsf{MNP}[n] \to \mathsf{GMNP}[n] \\
f_c(\mathit{null}) & = & \mathit{null} \\
f_c(x) & = & x \\
f_c(\neg p) & = & \neg(f_c\, p) \\
f_c(p \wedge q) & = & f_c\, p \wedge f_c\, q \\
f_c(\langle n\rangle.p) & = & \neg x\ \wedge\langle n\rangle.(f_c\, p) \\
 & & \vee\ x\ \wedge \forall v'.y(v') \to (f_c\, p)(v') \\
f_c(p.\langle n\rangle) & = & \quad (f_c\, p \wedge \neg x).\langle n\rangle \\
 & & \vee\ y \wedge \forall v'.x(v') \to (f_c\, p)(v') \\
f_c(p.\mathsf{U}_{\langle n\rangle}.q) & = & \quad (f_c\, p \wedge \neg x).\mathsf{U}_{\langle n\rangle}.(f_c\, q) \\
 & & \vee\ (f_c\, p).\mathsf{U}_{\langle n\rangle}.(f_c\, p \wedge x)\ \wedge \forall v'.y(v') \to (f_c\, p \wedge \neg x).\mathsf{U}_{\langle n\rangle}.(f_c\, q)(v') \\
f_c(p.\mathsf{S}_{\langle n\rangle}.q) & = & \quad (f_c\, p \wedge \neg x).\mathsf{S}_{\langle n\rangle}.(f_c\, q \wedge \neg x) \\
 & & \vee\ x \wedge (f_c\, p).\mathsf{S}_{\langle n\rangle}.(f_c\, q) \\
 & & \vee\ (f_c\, p \wedge \neg x).\mathsf{S}_{\langle n\rangle}.(f_c\, p \wedge y \wedge \neg x) \\
 & & \quad \wedge\ \forall v'.x(v') \to (f_c\, p).\mathsf{S}_{\langle n\rangle}.(f_c\, q)(v') \\
f_c(p.\mathsf{U}_{[n]}.q) & = & \quad (f_c\, p \wedge \neg x).\mathsf{U}_{[n]}.(f_c\, q) \\
 & & \vee\ (f_c\, p \wedge \neg x).\mathsf{U}_{[n]}.(f_c\, p \wedge x \vee f_c\, q) \\
 & & \quad \wedge\ \forall v'.y(v') \to (f_c\, p \wedge \neg x).\mathsf{U}_{[n]}.(f_c\, q)(v') \\
f_c(p.\mathsf{S}_{[n]}.q) & = & \quad \neg(\neg f_c\, q).\mathsf{S}_{\langle n\rangle}.(\neg f_c\, q \wedge \neg y \wedge x.[n]) \\
 & & \wedge \left(\begin{array}{l}
 \qquad x\ \wedge (f_c\, p \wedge \neg y).\mathsf{S}_{[n]}.(f_c\, q) \\
 \vee \neg x\ \wedge (f_c\, p \wedge \neg y).\mathsf{S}_{[n]}.(f_c\, q \vee x) \\
 \vee \neg x\ \wedge (f_c\, p).\mathsf{S}_{[n]}.(f_c\, p \wedge y \wedge \mathsf{false}.[n] \vee f_c\, q \vee x) \\
 \qquad\quad \wedge\ \forall v'.x(v') \to\ (f_c\, p \wedge \neg y).\mathsf{S}_{[n]}.(f_c\, q)(v') \\
 \qquad\qquad\qquad\qquad \wedge \neg(\neg f_c\, q).\mathsf{S}_{\langle n\rangle}.(\neg f_c\, q \wedge x.[n])(v')
 \end{array}\right)
\end{array}$$

Lemma 4.4.3. *For any program variable* x *and any* $\mathsf{FO}^{\mathsf{TC}}$ *formula* $\varphi(v)$*:*

$$\forall S \in \mathit{Store} :\ S \models \exists\, v.x(v) \wedge \varphi(v) \iff S \models \forall v.x(v) \to \varphi(v).$$

Proposition 4.4.4. *For commands c of the form* `x->n = y` *and any modal node predicate* p*,* $f_c(p)$ *and* $\mathsf{wlp}_c(p)$ *are equivalent on program stores, i.e.*

$$\forall S \in \mathit{Store}, u \in U :\ S, u \models f_c(p) \iff S, u \models \mathsf{wlp}_c(p).$$

From a $\mathsf{GMNP}[n]$ predicate p we can get back to a pure modal node predicate by evaluating the quantified sub-formulas that occur in p. This means, given a context Γ, for every quantified formula F occurring in p, we replace F by true or false depending on whether Γ entails F.

The $\mathsf{GMNP}[n]$ predicates that we evaluate express weakest liberal preconditions. In order to guarantee soundness, it is essential that the result of the evaluation is an under-approximation of the original predicate.

The process of evaluation is formalized by the function eval. The function eval respects the polarity of quantified formulas, in order to ensure that the evaluation indeed results in an under-approximation.

Definition 4.4.5. *The function* eval *that evaluates quantified sub-formulas in* $\mathsf{GMNP}[n]$ *predicates under a given context is defined on the structure of* $\mathsf{GMNP}[n]$ *as follows:*

$$\begin{array}{rcl}
\mathsf{eval} & \in & \{\mathsf{true}, \mathsf{false}\} \to \mathsf{FO}^{\mathsf{TC}} \to \mathsf{GMNP}[n] \to \mathsf{MNP}[n] \\
\mathsf{eval}\ t\ \Gamma\ a & = & a, \quad \textit{if } a \textit{ is an atom} \\
\mathsf{eval}\ t\ \Gamma\ (\neg p) & = & \neg(\mathsf{eval}\ (\neg t)\ \Gamma\ p) \\
\mathsf{eval}\ t\ \Gamma\ (p \wedge q) & = & (\mathsf{eval}\ t\ \Gamma\ p) \wedge (\mathsf{eval}\ t\ \Gamma\ q) \\
 & \dots &
\end{array}$$

$$\mathsf{eval}\ t\ \Gamma\ (\forall v.x(v) \to p(v)) \quad = \quad \begin{cases} t, & \textit{if}\ \Gamma \models \forall v.x(v) \to (t \leftrightarrow (\mathsf{eval}\ t\ \Gamma\ p)) \\ \neg t, & \textit{otherwise} \end{cases}$$

Proposition 4.4.6. *Let p be a $\mathsf{GMNP}[n]$ predicate and let Γ be a closed $\mathsf{FO}^{\mathsf{TC}}$ formula. The evaluation of p under* true *and Γ results in an under-approximation of p, i.e.*

$$\mathsf{eval\ true}\ \Gamma\ p \models_\Gamma p.$$

Although the evaluation of quantified sub-formulas in general results in an under-approximation of the original predicate, we can characterize sufficient conditions such that the evaluation does not lose precision.

First of all we are not interested in the evaluation under arbitrary contexts. According to Definition 3.2.14, we only consider contexts Γ which are abstract stores that result from the application of the abstraction function α to a single store S.

If the set of abstraction predicates *Pred* contains *enough* node predicates, there will be no loss of precision, because for quantified sub-formulas F, as they occur in $\mathsf{GMNP}[n]$, either all stores satisfying $\alpha(S)$ will also satisfy F, or all those stores will satisfy the negation of F. In the following, we make more precise what is meant by the term "enough".

Lemma 4.4.7. *Let S be some store and let Pred be the finite set of abstraction node predicates. For any program variable* x *with $x(v) \in$ Pred and formula $\varphi(v) \in \mathcal{F}_{Pred}$:*

$$\begin{aligned} \textit{either}\ \alpha(S) &\models \forall v.x(v) \to \varphi(v) \\ \textit{or}\ \alpha(S) &\models \forall v.x(v) \to \neg\varphi(v). \end{aligned}$$

We now show that for a given $\mathsf{GMNP}[n]$ predicate, we can construct a set of abstraction predicates *Pred* such that the evaluation under abstract stores over *Pred* does not lose precision. This allows us to prove closeness of $\mathsf{MNP}[n]$ under weakest liberal preconditions.

We construct the set *Pred* using a closure operator that collects all $\mathsf{MNP}[n]$ predicates that possibly occur as a sub-formula during the process of evaluation. This guarantees that Lemma 4.4.7 is always applicable when a quantified sub-formula is evaluated and ensures that we do not lose precision.

Definition 4.4.8. *The **closure** cl maps a $\mathsf{GMNP}[n]$ predicate to a set of modal node predicates as follows:*

$$\begin{aligned} \mathsf{cl} &\in \mathsf{GMNP}[n] \to 2^{\mathsf{MNP}[n]} \\ \mathsf{cl}(p) &= \mathsf{cl}_1(p) \cup \mathsf{cl}_2(p) \end{aligned}$$

where cl_1 and cl_2 are inductively defined on the structure of p by:

$$\begin{aligned} \mathsf{cl}_1(a) &= \{a\}, \quad \textit{if a is an atom} \\ \mathsf{cl}_1(\neg p_1) &= \{\, \neg p_1' \mid p_1' \in \mathsf{cl}_1(p_1)\,\} \\ \mathsf{cl}_1(p_1 \wedge p_2) &= \{\, p_1' \wedge p_2' \mid p_1' \in \mathsf{cl}_1(p_1)\ \textit{and}\ p_2' \in \mathsf{cl}_1(p_2)\,\} \\ &\dots \\ \mathsf{cl}_1(\forall v.x(v) \to p_1(v)) &= \{\mathsf{true}, \mathsf{false}\} \end{aligned}$$

$$\begin{aligned} \mathsf{cl}_2(a) &= \emptyset, \quad \textit{if a is an atom} \\ \mathsf{cl}_2(\neg p_1) &= \mathsf{cl}_2(p_1) \\ \mathsf{cl}_2(p_1 \wedge p_2) &= \mathsf{cl}_2(p_1) \cup \mathsf{cl}_2(p_2) \\ &\dots \\ \mathsf{cl}_2(\forall v.x(v) \to p_1(v)) &= \{x(v)\} \cup \mathsf{cl}_2(p_1) \cup \mathsf{cl}_1(p_1). \end{aligned}$$

Lemma 4.4.9. *Let p be a $\mathsf{GMNP}[n]$ predicate. The closure of p contains all possible modal node predicates that may occur in the image of* eval*, i.e. for every closed $\mathsf{FO}^{\mathsf{TC}}$ formula Γ and truth value t we have:*

$$\mathsf{eval}\ t\ \Gamma\ p \in \mathsf{cl}_1(p) \textit{ and } \mathsf{eval}\ t\ \Gamma\ p \in \mathsf{cl}(p).$$

Lemma 4.4.10. *Let p be a $\mathsf{GMNP}[n]$ predicate and let*

$$\forall v.x(v) \to q(v)$$

be some sub-formula of p then $x(v) \in \mathsf{cl}(p)$ and for every closed $\mathsf{FO}^{\mathsf{TC}}$ formula Γ and truth value t:

$$\mathsf{eval}\ t\ \Gamma\ q \in \mathsf{cl}(p).$$

Proposition 4.4.11. *Let p be a $\mathsf{GMNP}[n]$ predicate. If $\mathsf{cl}(p) \subseteq \mathit{Pred}$ then:*

$$\forall S \in \mathit{Store} : p \models\!\!\mid_{\alpha(S)} \mathsf{eval}\ \mathsf{true}\ (\alpha\ S)\ p.$$

Corollary 4.4.12. *Let c be a command of the form* `x->n = y` *and let p be an $\mathsf{MNP}[n]$ predicate. If $\mathsf{cl}(f_c\ p) \subseteq \mathit{Pred}$ then:*

$$\forall S \in \mathit{Store} : \mathsf{wlp}_c(p) \models\!\!\mid_{\alpha(S)} \mathsf{eval}\ \mathsf{true}\ (\alpha\ S)\ (f_c\ p).$$

Proposition 4.4.13. *For commands c of the form* `x->n = y` *and any $\mathsf{MNP}[n]$ predicate p there is a finite subset Pred of $\mathsf{MNP}[n]$ predicates such that:*

$$\forall S \in \mathit{Store} : \mathsf{wlp}^{\#}_{c,\alpha(S)}(p) \models\!\!\mid_{\alpha(S)} \mathsf{wlp}_c(p).$$

From Propositions 4.4.1 and 4.4.13 we can finally conclude that $\mathsf{MNP}[n]$ is closed under weakest liberal preconditions.

Theorem 4.4.14. *For any atomic command c, the class of modal node predicates $\mathsf{MNP}[n]$ is closed under wlp_c.*

The above result is a first step towards the development of an automated abstraction refinement procedure for modal node predicates. These procedures are essential components of modern tools for automated formal verification. If the abstract domain is not precise enough to verify a given property, it is refined by generating new abstraction predicates from spurious counterexamples in the abstract system. The refinement continuous until the analysis succeeds in either proving or disproving the property that is verified. In the case of a forward analysis, the generation of new predicates typically relies on the computation of weakest liberal preconditions.

Due to undecidability of the verification problem, termination of the refinement loop is not guaranteed in general. However, we can give at least a termination argument for a restricted class of programs. If we compute weakest liberal preconditions according to the algorithm given above, it is easy to see that only for commands of the form:

c : `x = y->n`.

the function f_c may introduce unbounded nesting of modal operators. Hence, if we consider programs built up from atomic commands that are not of this form then we only need a finite set of modal node predicates, in order to ensure that the Cartesian post is precise with respect to the best abstract post. This is due to the fact that for the remaining atomic commands the number of modal node predicates that is needed to express wlp_c is bounded by the number of program variables.

Chapter 5

Case Study: Reversing Singly-linked Lists

In this chapter we apply the techniques developed in the previous chapters to a standard example in shape analysis: a program reversing a singly-linked list.

5.1 The Program Reverse

The program REVERSE is declared in Figure 5.1. We assume that initially the program store contains one singly-linked list whose head is pointed to by program variable x. The program reverses the list by following its n-pointer fields starting from x and reversing each of them one by one. After the program has terminated program variable y points to the head of the reversed list.

Our goal is to show that under the assumption that the input list is acyclic the resulting reversed list will be acyclic, too. Although the acyclicity of the input list is never violated during the program execution, for automated methods this is a nontrivial property to check. The presence of destructive updates like c_{l5}[1] in program REVERSE makes it difficult to reason about reachability properties such as acyclicity.

[1]We make the notational convention that for some program location l we write c_l for the command that is associated with l.

```
List x,y,t;

l0: y = NULL;
l1: while(x != NULL){
l2:   t = y;
l3:   y = x;
l4:   x = x->n;
l5:   y->n = t;
    }
l6:
```

Figure 5.1: Program REVERSE

5.2 Choosing Abstraction Predicates

For a singly-linked list pointed to by program variable y we express acyclicity in terms of modal node predicates as follows:

$$\mathsf{acyclic}_y \overset{def}{=} \forall v.y(v) \rightarrow \langle n^* \rangle.\mathit{null}(v).$$

The formula expresses that there is a path from the node pointed to by program variable y that ends in NULL. Since we are dealing with lists, we can use the modal node predicate $\langle n^* \rangle.\mathit{null}(v)$ instead of $[n^*].\mathit{null}(v)$ to express acyclicity.

In order to come up with a suitable set of abstraction predicates, we compute weakest liberal preconditions of the node predicates that occur in the formula $\mathsf{acyclic}_y$. Starting from program location l6 we compute wlp_c for the modal node predicates $y(v)$ and $\langle n^* \rangle.\mathit{null}$ following the commands in the program backwards. Thereby, we make use of the fact that wlp_c can be expressed in terms of modal node predicates, as it is shown in Section 4.4. Every newly observed node predicate that is needed to express wlp_c is added to the set of abstraction predicates.

If we compute weakest liberal preconditions for one iteration on the commands in the body of the while-loop, we generate the following modal node predicates:

$$x,\ y,\ t,\ \mathit{null},\ (\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null},\ (\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null},\ \langle n^* \rangle.y,\ \langle n^* \rangle.x,\ \langle n^* \rangle.\mathit{null}.$$

If we continued naïvely refining the above set of node predicates, the command c_{14} would cause trouble. The operator $\mathsf{wlp}_{c_{14}}$ causes every occurrence of x in one of the node predicates to be replaced by $x.\langle n \rangle$. This would generate modal node predicates with unbounded nested modal operators and result in a non-terminating refinement loop. In order to circumvent this problem, we add the node predicate $x.\langle n^* \rangle$ as a widening of $x.\langle n \rangle$. We will use a splitting operator (cf. Section 3.2.6) in order to handle nondeterminism caused by imprecise abstract wlps.

Computing $\mathsf{wlp}_{c_{15}}$ of the node predicate $x.\langle n^* \rangle$ generates two additional modal node predicates:

$$(\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y) \text{ and } (\neg y).\mathsf{S}_{\langle n \rangle}.x.$$

Our experiments showed that if we omit the node predicates $\langle n^* \rangle.y$, $\langle n^* \rangle.x$, and $(\neg y).\mathsf{S}_{\langle n \rangle}.x$, the remaining set of node predicates still gives us a sufficiently precise abstraction for proving our target property. Since we want to focus on the essential aspects of the example, we keep the set of abstraction predicates as tight as possible. Thus, our choice of abstraction predicates is as follows:

$$\begin{aligned}\mathit{Pred} \overset{def}{=} \{\, & x,\ y,\ t,\ \mathit{null},\ (\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null},\ (\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null},\ x.\langle n^* \rangle, \\ & (\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y),\ \langle n^* \rangle.\mathit{null} \,\}.\end{aligned}$$

5.3 Splitting Operator

In this section we discuss the splitting operator that we will use to handle the nondeterminism in the abstract transition system that is caused by command c_{14}.

We want to ensure that each abstract store that occurs during the analysis has precise information about the nodes on which the program variables point to. More precisely, we want to ensure that for each program variable x all abstract stores contain exactly one monomial in which the node predicate x occurs positively.

```
$\mathsf{splitSingleton}_c[p]$ ($\Psi$ as $\forall v.\psi(v)$: $AbsStore[Pred]$): $AbsDom[Pred \cup \{\mathsf{wlp}_c(p)\}]$
  $\Psi'$:= $\mathsf{false}$
  for each monomial $P$ over $Pred$ with $P \models \psi$ do
    if $P \not\models_\Psi \mathsf{wlp}^\#_{c,\Psi}[Pred](\neg p)$ then
      if $P \models_\Psi \mathsf{wlp}^\#_{c,\Psi}[Pred](p)$ or $\mathsf{isSingleton}(P)$ then
        $\Psi'$ := $\Psi'$ $\vee$ $\forall v.(P \wedge \mathsf{wlp}_c(p)) \vee (\psi \wedge \neg P \wedge \neg\,\mathsf{wlp}_c(p))$
      else
        $\Psi'$ := $\Psi'$ $\vee$ $\forall v.(P \wedge \mathsf{wlp}_c(p)) \vee (\psi \wedge \neg\,\mathsf{wlp}_c(p))$
  return $\Psi'$
```

Figure 5.2: A simple splitting operator for splitting on node predicates denoting singleton sets.

The weakest liberal precondition for command c_{14} and node predicate x is given by:

$$\mathsf{wlp}_{c_{14}}(x) = x.\langle n\rangle.$$

Unfortunately, $x.\langle n\rangle$ can not be precisely expressed in terms of node predicates in $Pred$. Thus, computing the Cartesian post for command c_{14} will result in abstract stores that do not possess precise information about program variable x. In order to address this problem, we introduce a splitting operator for command c_{14} that splits on node predicate x.

The pseudo code describing the splitting operator is given in Figure 5.2. It can be applied for splitting on node predicates p that are guaranteed to denote singleton sets. This is in particular the case for node predicates corresponding to program variables.

The splitting operator $\mathsf{splitSingleton}_c[p]$ takes as argument an abstract store Ψ over $Pred$. It looks at each abstract node P in Ψ and checks whether some concrete node represented by P may satisfy p after execution of c. If there are both nodes in P that satisfy p and nodes that do not satisfy p after c is executed then P is split into two separate abstract nodes, one entailing $\mathsf{wlp}_c(p)$ and one entailing $\mathsf{wlp}_c(\neg p)$.

If there is more than one abstract node P in Ψ that represents nodes satisfying p after c, we use the fact that p denotes a singleton and split the complete abstract store. Each of the resulting abstract stores contains exactly one of the corresponding abstract nodes.

The function $\mathsf{isSingleton}$ checks whether some other node predicate denoting a singleton occurs in P. If this is the case, it indicates that P need not be split. In our example node predicates denoting singletons are: x, y, t, and $null$.

It is not hard to see that $\mathsf{splitSingleton}_c[p]$ is indeed a splitting operator according to Definition 3.2.17. However, it is not the most precise splitting operator. This is due to the fact that we do not check consistency of the remaining abstract nodes in Ψ with respect to abstract nodes that result from splitting.

In order to instantiate $\mathsf{splitSingleton}_c[p]$ for command c_{14}, we only need to provide the context-sensitive abstract wlps for c_{14} and x. This is done in the following section.

c	p	$\mathsf{wlp}^{\#}_c[\mathit{Pred}](p)$	$\mathsf{wlp}^{\#}_c[\mathit{Pred}](\neg p)$
c_{10}	y	null	$\neg \mathit{null}$
	$(\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}$	$\langle n^* \rangle.\mathit{null}$	$\neg\langle n^* \rangle.\mathit{null}$
	$(\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y)$	$x.\langle n^* \rangle \wedge \neg(\mathit{null} \wedge x)$	$\neg x.\langle n^* \rangle$ $\vee\ \mathit{null} \wedge x$
c_{12}	t	y	$\neg y$
c_{13}	y	x	$\neg x$
	$(\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}$	$(\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null}$	$\neg(\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null}$
	$(\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y)$	false	true
c_{14}	x	false	$\neg x.\langle n^* \rangle$ $\vee\ x \wedge \langle n^* \rangle.\mathit{null}$
c_{15}	$x.\langle n^* \rangle$	x $\vee\ (\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y)$	$\neg x.\langle n^* \rangle \wedge y$ $\vee\ \underline{\forall v.y \rightarrow \neg x.\langle n^* \rangle}$
	$(\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y)$	$\neg x.\langle n^* \rangle$
	$\langle n^* \rangle.\mathit{null}$	null $\vee\ (\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}$ $\vee\ y \wedge \underline{\forall v.t \rightarrow (\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}}$	$y \wedge t$ $\vee\ \underline{\forall v.y \rightarrow t}$
	$(\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}$	$(\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}$	$\neg\, \mathsf{wlp}^{\#}_c(p)$
	$(\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null}$	$(\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null} \wedge (\neg y).\mathsf{U}_{\langle n \rangle}.\mathit{null}$ $\vee\ y \wedge \neg x \wedge \underline{\forall v.t \rightarrow (\neg x).\mathsf{U}_{\langle n \rangle}.\mathit{null}}$	$x \wedge \neg \mathit{null}$

Table 5.1: Abstract context-sensitive wlps of node predicates in *Pred* for atomic commands in REVERSE.

5.4 Context-sensitive Abstract Weakest Liberal Preconditions

We need to give the context-sensitive abstract wlp for each of the chosen abstraction predicates. In order to obtain a representation that is independent of the context, we express them in terms of $\mathsf{GMNP}[n]$. Evaluating the quantified formulas under a given context, as it is formalized in Section 4.4, results in the appropriate context-sensitive version.

Table 5.1 lists the context-sensitive abstract wlps for node predicates in *Pred*. Node predicates that are not explicitly listed in the table do not change under the appropriate command. Since for command c_{14} we split on the node predicate x, we only give the context-sensitive abstract wlp for x itself. This is the only one that is needed in order to compute the operator $\mathsf{splitSingleton}_{c_{14}}[x]$.

For the computation of the Cartesian post with splitting that is used for command c_{14}, we extend *Pred* by node predicate $x.\langle n \rangle$. The context-sensitive abstract wlps over the refined set of abstraction predicates are given in Table 5.2.

5.5 Computing the Least Fixed Point

Before we can eventually compute the least fixed point of program REVERSE, we first need to model the data flow equations and give the abstraction of the initial set of stores under *Pred*.

The abstract domain $\mathit{AbsDom}[\mathit{Pred}]$ only describes the abstraction of the program stores, but contains no information regarding the program locations. We

c	p	$\mathsf{wlp}^{\#}_{c}[\mathit{Pred} \cup \{x.\langle n\rangle\}](p)$	$\mathsf{wlp}^{\#}_{c}[\mathit{Pred} \cup \{x.\langle n\rangle\}](\neg p)$
c_{14}	x	$x.\langle n\rangle$	$\neg x.\langle n\rangle$
	$x.\langle n\rangle$	$x \wedge x.\langle n\rangle$	$\neg x.\langle n^*\rangle$ $\vee\, x.\langle n\rangle \wedge \langle n^*\rangle.\mathit{null}$ $\vee\, x \wedge \neg x.\langle n\rangle$
	$x.\langle n^*\rangle$	$x.\langle n\rangle$ $\vee\, x.\langle n^*\rangle \wedge \neg x$ $\vee\, x.\langle n^*\rangle \wedge \neg\langle n^*\rangle.\mathit{null}$	$\neg x.\langle n^*\rangle$ $\vee\, x \wedge \neg\langle n^*\rangle.\mathit{null}$
	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$x.\langle n\rangle \wedge \neg y$ $\vee\, x.\langle n^*\rangle \wedge \neg x$ $\underline{\wedge\, \forall v.y \rightarrow (x \wedge \neg x.\langle n\rangle)}$ $\vee\, x.\langle n^*\rangle \wedge \neg\langle n^*\rangle.\mathit{null}$ $\underline{\wedge\, \forall v.y \rightarrow (x \wedge \neg x.\langle n\rangle)}$	y $\vee\, \neg x.\langle n^*\rangle$ $\vee\, x \wedge \neg\langle n^*\rangle.\mathit{null}$
	$(\neg x).\mathsf{U}_{\langle n\rangle}.\mathit{null}$	null $\vee\, \neg x.\langle n\rangle \wedge (\neg x).\mathsf{U}_{\langle n\rangle}.\mathit{null}$	$\neg\, \mathsf{wlp}^{\#}_{c}(p)$

Table 5.2: Abstract context-sensitive wlps for command c_{14} over the refined set of node predicates.

represent abstract states as tuples over $\mathit{AbsDom}[\mathit{Pred}]$, where the i-th component corresponds to the abstract stores at the i-th program location. The partial order on those tuples as well, as meat and join operations can be defined as the point-wise extensions of the corresponding operations on $\mathit{AbsDom}[\mathit{Pred}]$.

The following abstract post operator on tuples over $\mathit{AbsDom}[\mathit{Pred}]$ models the data flow equations in program REVERSE and gives us our final abstract post operator that we will use for the analysis:

$$\begin{aligned}
\mathsf{post}^{\#} &\in (\mathit{AbsDom}[\mathit{Pred}])^7 \rightarrow (\mathit{AbsDom}[\mathit{Pred}])^7 \\
\mathsf{post}^{\#}\langle \Psi_0, \dots, \Psi_6\rangle &= \langle \mathtt{l0} : \mathsf{false}, \\
&\quad\ \ \mathtt{l1} : \mathsf{post}^{\#}_{c_{10},\mathsf{Cart}}(\Psi_0) \vee \mathsf{post}^{\#}_{c_{15},\mathsf{Cart}}(\Psi_5), \\
&\quad\ \ \mathtt{l2} : (\forall v.x(v) \rightarrow \neg \mathit{null}(v)) \wedge \Psi_1, \\
&\quad\ \ \mathtt{l3} : \mathsf{post}^{\#}_{c_{12},\mathsf{Cart}}(\Psi_2), \\
&\quad\ \ \mathtt{l4} : \mathsf{post}^{\#}_{c_{13},\mathsf{Cart}}(\Psi_3), \\
&\quad\ \ \mathtt{l5} : \mathsf{post}^{\#}_{c_{14},\mathsf{splitSingleton}_{c_{14}}[x]}(\Psi_4), \\
&\quad\ \ \mathtt{l6} : (\forall v.x(v) \rightarrow \mathit{null}(v)) \wedge \Psi_1\rangle.
\end{aligned}$$

The initial abstract state $\mathsf{init}^{\#}$ is given by the tuple:

$$\mathsf{init}^{\#} \overset{def}{=} \langle \Psi_{\mathsf{init}}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}\rangle$$

where Ψ_{init} is the abstraction of the initial set of stores init. The set init contains all singly-linked lists that have at least one element and whose head is pointed to by program variable `x`. The abstraction of init under Pred, is described by the following table:

	x	y	t	null	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.\mathit{null}$	$(\neg x).\mathsf{U}_{\langle n\rangle}.\mathit{null}$	$\langle n^*\rangle.\mathit{null}$
Ψ_{init}	1	$*$	$*$	0	1	$*$	$*$	0	1
	0	$*$	$*$	0	1	$*$	$*$	1	1
	0	$*$	$*$	1	1	$*$	1	1	1

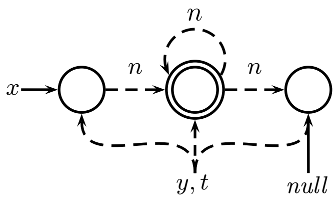
Each row in the table represents an abstract node in Ψ_{init}. If a cell in a row contains the value 1, the corresponding predicate occurs positively in the abstract node, the value 0 means it occurs negated, and $*$ means it does not occur at all. The first row represents the head of the list pointed to by `x`, the third row represents `NULL`, and the second row represents all remaining nodes in the list.

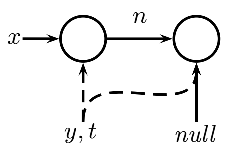
In order to provide a better intuition for the set of concrete stores that is represented by some abstract store, we use the graph notation from [27] which they use for the representation of three-valued logical structures.

This graph notation is similar to the one we introduced in Section 2.3.1, in order to represent two-valued logical structures. The nodes in such a graph give the partitioning of the universe of the represented logical structures into disjoint nonempty subsets. Singly-lined nodes represent singleton sets, whereas doubly-lined summary nodes may represent arbitrary large subsets of concrete nodes.

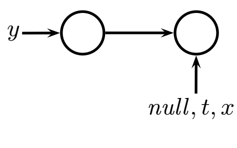
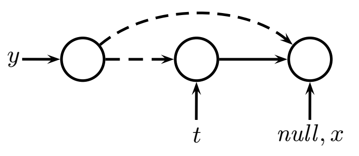
Predicate symbols over the nodes in the graph may be interpreted indefinitely, denoted by a dashed line between the corresponding nodes. The meaning of such a dashed line is that each logical structure represented by the graph can interpret the corresponding predicate symbol arbitrarily on nodes that are represented by the appropriate abstract nodes.

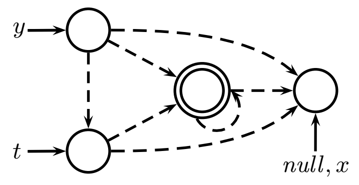
For each abstract store Ψ, we give a set of such graphs that represents the same set of concrete stores as Ψ. The initial abstract store Ψ_{init} represents the same set of concrete stores as the following two graphs:





We are now ready to compute the least fixed point of $\mathsf{post}^\#$ under $\mathsf{init}^\#$. The individual steps of the fixed point iteration are given in Appendix B. If we project $\mathsf{lfp}(\mathsf{post}^\#(\mathsf{init}^\#))$ on the 7th component, i.e. the one corresponding to program location `l6`, we get the following abstract stores:

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$
	0	1	0	0	0	0	0	1	1
`l6` :	0	0	0	0	1	1	1	1	1
	1	0	1	1	1	1	1	1	1
									
	0	0	1	0	0	0	1	1	1
	0	1	0	0	0	0	0	1	1
	0	0	0	0	1	1	1	1	1
	1	0	1	1	1	1	1	1	1
									

	0	0	0	0	0	0	1	1	1
	0	0	1	0	0	0	1	1	1
	0	1	0	0	0	0	0	1	1
	0	0	0	0	1	1	1	1	1
	1	0	1	1	1	1	1	1	1
									

One can see that each of the abstract stores contains an abstract node which is inconsistent with respect to the other abstract nodes. The corresponding rows in the table are marked gray. An abstract node P is inconsistent in some abstract store Ψ, if for all concrete stores represented by Ψ there is no node satisfying P. In the concrete stores represented by the abstract stores above it is always the case that program variable `x` points to `NULL`. Thus, `NULL` is the only possible node that is reachable by program variable `x`. This makes the marked abstract nodes inconsistent, because they are supposed to be reachable by `x` but are not `NULL`.

The inconsistent abstract nodes appear due to the use of the splitting operator $\mathsf{splitSingleton}_{c_{14}}[x]$. Replacing this splitting operator with the most precise splitting operator for command c_{14} and node predicate x would eliminate all inconsistencies.

It may appear disturbing that the information regarding the n predicate for the nodes pointed to by program variables `y` and `t` is rather imprecise. This is due to the fact that the appropriate reachability information is not expressible in terms of abstraction predicates in $Pred$.

However, the set of abstraction predicates was chosen in order to synthesize a specific invariant, namely that the output list of program REVERSE is always acyclic. This invariant is indeed entailed by the least fixed point. All abstract nodes in which the node predicate y occurs positively entail the node predicate $\langle n^* \rangle.null$. Thus, in all represented concrete stores it is possible to reach `NULL` from program variable y. This implies that all corresponding output lists are indeed acyclic.

Chapter 6

Related Work

There is a lot of recent work on shape analysis. In the following we compare some of these works to our own results.

6.1 Abstract Domains for Shape Analysis

Abstract domains designed for shape analysis [17, 5, 29, 26] are based on several variants of shape graphs. The use of three-valued logical structures as abstract domain is a generalization of this idea. In the following, we discuss this particular approach more thoroughly.

A shape analysis algorithm based on three-valued logic is given in [27]. The TVLA tool [21] implements this algorithm. The abstract domain used in this approach is given by sets of three-valued logical structures. Three-valued logical structures are used to finitely abstract infinite sets of two-valued logical structures that correspond to sets of program stores.

Their notion of *canonical abstraction* relates sets of two-valued logical structures with finite sets of three-valued logical structure. The abstraction is parameterized by a finite set of unary abstraction predicates on nodes. It collapses all subsets of nodes in an abstracted structure whose elements are indistinguishable under the abstraction predicates to single abstract nodes in the universe of the resulting three-valued logical structure.

The interpretation of a predicate symbol in the three-valued logical structure is given by a conservative abstraction of its interpretation on the represented nodes in the abstracted two-valued structures. This is accomplished by using the *indefinite* truth value in the three-valued logic. The interpretation of some predicate symbol p on abstract nodes has the *indefinite* value, if the interpretation of p on the represented nodes in the abstracted two-valued structure is ambiguous.

The computation of the abstract post operator is based on predicate-update formulas, i.e. abstract weakest liberal preconditions of abstraction predicates. The precision of the abstract post operator can be improved by adding *instrumentation predicates*. Instrumentation predicates preserve certain information about the concrete structures in the abstraction, e.g. acyclicity or sharing. The additional predicates can be used to increase the precision of predicate-update formulas. So called focus and coerce operations are used to obtain more precise results in the presence of nondeterminism in the abstract transition system.

We used several ideas from this work. In particular, we adapted the representation of concrete program stores as two-valued logical structures. However, we do not need three-valued logic, because we symbolically abstract sets of stores using formulas. *Canonical abstraction* is the basis of our own node predicate abstraction. Our framework does not provide the ability to define additional instrumentation predicates. We want to use automated abstraction refinement instead. If the two operations focus and coerce are combined into a single operation on formulas, the result corresponds to a splitting operator as given in Section 3.2.6.

A translation from three-valued logical structures, as they arise in [27], into an isomorphic representation in two-valued first-order logic is first given in [30]. Shape analysis constraints [19] generalize this result to a Boolean algebra of first-order formulas that is isomorphic to the same class of three-valued logical structures. The translation of [30] relates each three-valued structure to a conjunction of three constraints: a totality constraint expressing that every node is represented by some abstract node; an existential constraint expressing the non-emptiness of each abstract node; and a constraint expressing the values of binary predicate symbols on abstract nodes.

Our notion of an abstract store exactly corresponds to the totality constraint in the translation above. Consequently, for the same set of abstraction predicates our abstract domain is weaker as the one used in TVLA. However, we indicated in Section 3.1.4 that constraints for the values of binary predicate symbols can be translated to additional unary abstraction predicates. It may also be possible to strengthen our abstract stores by adding existential constraints, but existential constraints complicate the application of Cartesian abstraction.

6.2 Symbolic Methods in Shape Analysis

In [31] an algorithm for the symbolical computation of most precise operators for shape analysis is given. The paper describes an algorithm implementing an *assume* operation. The operator $assume[\varphi](a)$ takes a closed formula φ and a set of three-valued logical structures a and computes the best under-approximation of a satisfying the formula φ, provided that a decision procedure for the underlying logic exists. The $assume$ operation allows inter-procedural shape analysis based on assume-guarantee reasoning. Moreover, $assume$ can be instantiated to compute best abstraction functions, most-precise post operators, and the meet operation for abstract domains of three-valued logical structures.

Although this algorithm is primarily designed to be applied in the context of abstract domains of three-valued logical structures, a modified version could be used to compute most precise splitting operators.

Dams and Namjoshi [9] use predicate abstraction and model checking for shape analysis. They abstract pointer programs by Boolean programs using *classical* predicate abstraction [12] based on state predicates. After the predicate abstraction step they apply standard finite state model checking techniques to analyze the obtained Boolean program. Their abstraction predicates describe properties of the shape of the program stores, including reachability properties. They give a weakest precondition calculus, in order to compute predicate-updates and provide heuristics for abstraction refinement. The calculus makes use of generalized reachability predicates. These predicates are parameterized by sets of addresses that have to be avoided on a path. The calculus allows to express weakest preconditions of reachability properties in a similar way as it is done in this work using modal node predicates.

Although both their and our work use techniques from predicate abstraction, our abstract domain is induced by unary node predicates instead of *nullary* state predicates. The size of our abstract domain is triple exponential in the number of abstraction predicates, whereas the abstract domain in predicate abstraction is only doubly exponential. Hence, in order to get equally precise abstract domains in both frameworks, predicate abstraction needs 2^n state predicates compared to n node predicates in our framework. However, the lower bound for the complexity of the abstract post operator is in both frameworks linear in the number of abstraction predicates. Therefore, the use of unary abstraction predicates in our framework gains a better ratio between precision and complexity.

6.3 Decidable Logics for Shape Analysis

Decidable logics that are expressive enough to be useful in shape analysis are rare. Weak monadic second order logic over trees is an example for a logic that is decidable and used in this application domain [22]. In the following, we discuss some more recent results on decidable logics that have been primarily studied for their potential application in shape analysis.

In [15] the boundary between decidability and undecidability for transitive closure logics is explored. The authors introduce the logic $\exists\forall(\mathrm{DTC}^{+}[E])$ and prove its decidability. This logic is a fragment of first-order logic with transitive closure adhering the following restrictions: the signature contains just one binary predicate symbol E; formulas are in $\exists\forall$-prenex form; and positive applications of transitive closure are only allowed for deterministic paths of the relation E. They show that weakening any of the above restrictions results in an undecidable logic.

Transitive closure logics can express reachability relations between pairs of variables, while guarded fixed point logics – and thus modal node predicates – can do so only for pairs of unary relations. On the other hand, guarded fixed point logics remain decidable without being restricted in the number of binary predicate symbols. This makes it easier to express properties of more complex data structures. However, [16] gives a way to handle data structures that are intractable due to such limitations by simulating them on structures for decidable logics. The authors show that many data structures appearing in practical application, such as trees and doubly-linked lists, can be handled via simulation using just a single binary predicate symbol.

Role logic [18] is a variable free logic equivalent to first-order logic with transitive closure and designed as a specification and annotation language for shape analysis. The authors identify the decidable fragment RL^2 of role logic. RL^2 is as expressive as two-variable logic with counting. In [20] it is shown that the syntax of RL^2 can be extended by spatial conjunction from separation logic [25], while remaining decidable. Spatial conjunction provides an elegant way to express concatenation of records.

Counting constraints – and thus RL^2 – allow the specification of the precise alias relationship between heap objects. This particularly includes the ability to express information about sharing. Modal node predicates do not offer such expressiveness. On the other hand, RL^2 cannot express reachability, which was our primary intention to propose modal node predicates.

Chapter 7

Conclusion

7.1 Summary

The goal of this work was to obtain a framework for symbolic shape analysis. We proposed an appropriate symbolic abstract domain. Conforming to the abstraction techniques that are used in state-of-the-art shape analysis algorithms such as [27], this abstract domain is parameterized by node predicates.

We used techniques from predicate abstraction to symbolically abstract programs manipulating heap-allocated data structures by Boolean heap programs. Boolean heap programs are characterized as a two-step Cartesian abstraction of the best abstract post operator $\mathsf{post}^{\#}$ on the chosen abstract domain. If the abstract transition system behaves deterministically, Boolean heap programs do not lose precision with respect to $\mathsf{post}^{\#}$.

We identified the class of modal node predicates, a fragment of first-order logic with transitive closure. Modal node predicates can be used for the analysis of reachability properties for linked data structures. We gave a preliminary result regarding decidability of the satisfiability problem for modal node predicates via translation into guarded fixed point logic. Moreover, we showed that for list-like structures, this class of node predicates is closed under weakest liberal preconditions. Both results are useful for the development of an automated abstraction refinement procedure for modal node predicates.

7.2 Future Work

The primary task for future work should be the development of a tool that implements the presented framework and allows a practical evaluation of the developed techniques. However, there is a number of interesting theoretical questions open for future work. We discuss some of them in the following.

7.2.1 Precision of the Abstract Domain

Our notion of an abstract store describes the covering of nodes in concrete stores by abstract nodes, i.e. the equivalence classes of nodes induced by the chosen abstraction predicates. However, it is a bit disturbing that an abstract store does not require each of its abstract nodes to be non-empty. The decision that we restrict ourselves to universal constraints over abstract nodes was made, in order to simplify the application of Cartesian abstraction. An interesting question is, whether the abstract domain can be made more precise by allowing existential constraints over abstract nodes, while still using Cartesian abstraction for the construction of Boolean heap programs.

7.2.2 Automated Abstraction Refinement

We gave preliminary results for the development of an automated abstraction refinement procedure for modal node predicates. However, there are open issues that still have to be solved.

We did not yet answer the question, whether the satisfiability problem for modal node predicates restricted to the class of program stores is decidable. The main problem lies in the treatment of functionality constraints, as they appear in the integrity formulas that define the set of program stores. However, a decision procedure for this problem is a key tool for the automated derivation of context-sensitive abstract weakest liberal preconditions. These are needed for the construction of Boolean heap programs. Hence, this is a main prerequisite to use modal node predicates in a fully automated analysis.

A second open issue is that, so far, we only showed closeness under wlp for the class of modal node predicates over a single binary predicate symbol. Hence, the application of modal node predicates is restricted to list-like data structures. In order to use them for the analysis of more complex data structures, we need to extend this result to an arbitrary number of binary predicate symbols.

7.2.3 Beyond Safety

Temporal properties of transition systems can be divided into two classes: safety properties and liveness properties. Roughly spoken: a safety property specifies the absence of finite error traces, whereas a liveness property specifies the absence of infinite error traces.

Our approach abstracts infinite state systems by finite state systems. The finite state abstraction is used to synthesize state invariants, i.e over-approximations of the reachable set of states of the analyzed system. However, state invariants correspond to safety properties. The information needed for the analysis of liveness properties gets lost during the finite state abstraction.

Transition invariants [24] capture both safety and liveness properties. A transition invariant is an over-approximation of the program's transition relation. *Transition predicate abstraction* [23] is a framework for the automatic synthesis of transition invariants. It solves the limitations of predicate abstraction by replacing the construction of a fixed point operator on an abstract domain of sets of abstract states with the construction of a fixed point operator on an abstract domain of binary relations on abstract states.

In analogy to predicate abstraction the abstract domain is parameterized by a set of *transition predicates*. Transition predicates are constraints denoting binary relations over states. It is a challenging task to find a suitable class of transition predicates for shape analysis that could be used to treat both safety and liveness properties.

Bibliography

[1] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In Tiziana Margaria and Wang Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2001.

[2] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 2002.

[3] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag, 2001.

[4] Randel E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(10):677–691, 1986.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310. ACM Press, 1990.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In Allen E. Emerson and Prasad A. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[8] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[9] Dennis Dams and Kedar S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, 2003.

[10] Bernd Finkbeiner. Language containment checking with nondeterministic BDDs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 24–38. Springer-Verlag, 2001.

[11] Erich Grädel and Igor Walukiewicz. Guarded Fixed Point Logic. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science (LICS '99)*, pages 45–54. IEEE Computer Society, 1999.

[12] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[13] Thomas A. Henzinger, Ranjit Jhala, Rubak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *SPIN 03: Model Checking of Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.

[15] Neil Immerman, Alexander Rabinovich, Thomas Reps, Mooly Sagiv, and Greta Yorsh. The Boundary Between Decidability and Undecidability for Transitive-Closure Logics. 2004. to appear in CSL'04.

[16] Neil Immerman, Alexander Rabinovich, Thomas Reps, Mooly Sagiv, and Greta Yorsh. Verification Via Structure Simulation. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 281–294. Springer-Verlag, 2004.

[17] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 244–256. ACM Press, 1979.

[18] Viktor Kuncak and Martin Rinard. On role logic. Technical Report 925, MIT Computer Science and Artificial Intelligence Laboratory, October 2003.

[19] Viktor Kuncak and Martin Rinard. Boolean Algebra of Shape Analysis Constraints. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 59–72. Springer-Verlag, 2004.

[20] Viktor Kuncak and Martin Rinard. Generalized Records and Spatial Conjunction in Role Logic. In Roberto Giacobazzi, editor, *11th Annual International Static Analysis Symposium (SAS'04)*, volume 3148 of *Lecture Notes in Computer Science*. Springer-Verlag, August 26–28 2004.

[21] Tal Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000.

[22] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 221–231. ACM Press, 2001.

[23] Andreas Podelski and Andrey Rybalchenko. Transition Predicate Abstraction and Fair Termination. Available from the authors.

[24] Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *Proceedings of 19th IEEE Symposium on Logic in Computer Science (LICS '04)*, pages 32–41. IEEE Computer Society, 2004.

[25] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, 2002.

[26] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.

[27] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[28] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theor. Computer Science*, 18:227–258, 1982.

[29] Edward Wang. *Analysis of Recursive Types in an Imperative Language*. PhD thesis, University of California at Berkeley, 1994.

[30] Greta Yorsh. Logical Characterizations of Heap Abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003.

[31] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 530–545. Springer-Verlag, 2004.

Appendix A

Proofs

Proposition 2.1.4. *For a transition system $\mathcal{S}$ the following is equivalent:*

(i) R is total and deterministic,

(ii) $\widetilde{\mathsf{pre}}$ is a homomorphism on the power set Boolean algebra of $State$.

Proof. "$(i) \Rightarrow (ii)$". Let the transition relation R be total and deterministic. By Proposition 2.3.7 (ii) we know that $\widetilde{\mathsf{pre}}$ distributes over set union. Thus, in order to prove that $\widetilde{\mathsf{pre}}$ is a homomorphism on the power set Boolean algebra of $State$, it suffices to show that $\widetilde{\mathsf{pre}}$ commutes with set complementation. For $S \subset State$, let $\overline{S}$ denote the complement of S. We have:

$$\begin{aligned} s \in \widetilde{\mathsf{pre}}(\overline{S'}) &\Rightarrow \forall s' \in State: (s,s') \in R \Rightarrow s' \in \overline{S'} \\ &\Rightarrow \exists s' \in State: (s,s') \in R \text{ and } s' \in \overline{S'} && (R \text{ total}) \\ &\Rightarrow \exists s' \in State: (s,s') \in R \text{ and } s' \notin S' \\ &\Rightarrow s \notin \widetilde{\mathsf{pre}}(S') \\ &\Rightarrow s \in \overline{\widetilde{\mathsf{pre}}(S')} \end{aligned}$$

$$\begin{aligned} s \in \overline{\widetilde{\mathsf{pre}}(S')} &\Rightarrow \exists s' \in State: (s,s') \in R \text{ and } s' \notin S' \\ &\Rightarrow \forall s' \in State: (s,s') \in R \Rightarrow s' \notin S' && (R \text{ deterministic}) \\ &\Rightarrow \forall s' \in State: (s,s') \in R \Rightarrow s' \in \overline{S'} \\ &\Rightarrow s \in \widetilde{\mathsf{pre}}(\overline{S'}) \end{aligned}$$

Thus we have $\widetilde{\mathsf{pre}}(\overline{S'}) = \overline{\widetilde{\mathsf{pre}}(S')}$.

"$(ii) \Rightarrow (i)$". Assume $\widetilde{\mathsf{pre}}$ is a homomorphism on the power set Boolean alegra of $State$. If $State$ is the empty set then R is the empty relation and thus total and deterministic. If otherwise $State$ is non-empty, assume that R was not total. Then there exists $s \in State$ such that:

$$\begin{aligned} & \forall s' \in State: (s,s') \notin R \\ \Rightarrow\ & \forall S \subseteq State: s \in \widetilde{\mathsf{pre}}(S) \\ \Rightarrow\ & \forall S \subseteq State: s \in \widetilde{\mathsf{pre}}(S) \text{ and } s \in \widetilde{\mathsf{pre}}(\overline{S}) \\ \Rightarrow\ & \forall S \subseteq State: s \notin \overline{\widetilde{\mathsf{pre}}(S)} \text{ and } s \in \widetilde{\mathsf{pre}}(\overline{S}) \\ \Rightarrow\ & \forall S \subseteq State: \overline{\widetilde{\mathsf{pre}}(S)} \neq \widetilde{\mathsf{pre}}(\overline{S}) \end{aligned}$$

which contradicts the fact that $\widetilde{\mathsf{pre}}$ commutes with set complementation. Now assume R was not deterministic then there are $s, s_1, s_2 \in \mathit{State}$ such that

$$\begin{aligned} & s_1 \neq s_2 \text{ and } (s, s_1) \in R \text{ and } (s, s_2) \in R \\ \Rightarrow\quad & s \notin \widetilde{\mathsf{pre}}(\{s_1\}) \text{ and } s \notin \widetilde{\mathsf{pre}}(\overline{\{s_1\}}) \\ \Rightarrow\quad & \overline{\widetilde{\mathsf{pre}}(\{s_1\})} \neq \widetilde{\mathsf{pre}}(\overline{\{s_1\}}) \end{aligned}$$

which again gives us a contradiction. ☐

Proposition 2.1.7. *$I \subseteq \mathit{State}$ is an inductive state invariant if and only if it is closed under the operator F, i.e. $F(I) \subseteq I$.*

Proof. Let I be an inductive state invariant. Since I is a state invariant, we have $\mathsf{reach} \subseteq I$ and therefore $\mathsf{init} \subseteq I$. Since I is inductive, we have $\mathsf{post}(I) \subseteq I$. Hence, we have:

$$F(I) = \mathsf{init} \cup \mathsf{post}(I) \subseteq I.$$

Let I be closed under the operator F, i.e. $F(I) \subseteq I$. We have by monotonicity of F:

$$\mathsf{reach} = \mathsf{lfp}(F) = \bigcap\{\, S \subseteq \mathit{State} \mid F(S) \subseteq S \,\}.$$

From this we can conclude $\mathsf{reach} \subseteq I$. Moreover, by definition of F, we have $F(I) \subseteq I$ implies $\mathsf{post}(I) \subseteq I$. Hence, I is an inductive state invariant. ☐

Proposition 2.3.2. *The operators $\mathsf{ext\text{-}post}_c$ and $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$ form a Galois connection on the power set lattice of extended stores.*

Proof. Follows immediately from the definitions of $\mathsf{ext\text{-}post}_c$ and $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$ on sets of extended stores and the fact that post_c and $\widetilde{\mathsf{pre}}_c$ form a Galois connection on the power set lattice of stores. ☐

Proposition 2.3.3. *The relation $\xrightarrow{c}$ is total and deterministic if and only if $\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c$ is a homomorphism on the power set Boolean algebra of extended stores.*

Proof. Follows immediately from the definition of $\widetilde{\mathsf{pre}}_c$ on sets of extended stores and Proposition 2.1.4. ☐

Proposition 2.3.5. *If the relation $\xrightarrow{c}$ is total and deterministic and if for every predicate symbol p there exists a predicate-update formula p'_c then the extended wlp of an $\mathsf{FO}^{\mathsf{TC}}$ formula φ can be computed by substituting syntactically all occurrences of predicate symbols in φ with their predicate-update formulas:*

$$\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi]\!]) = [\![\varphi[p'_c(v_1, \ldots, v_n)/p(v_1, \ldots, v_n)]]\!].$$

Proof. By structural induction on φ.
$\varphi = p(v_1, \ldots, v_n)$. By definition of predicate-update formulas we have:

$$\mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![p(v_1, \ldots, v_n)]\!]) = [\![p'_c(v_1, \ldots, v_n)]\!] = [\![\varphi[p'_c/p]]\!].$$

$\varphi = \varphi_1 \vee \varphi_2$. We have:

$$\begin{aligned} \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi_1 \vee \varphi_2]\!]) &= \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi_1]\!] \cup [\![\varphi_2]\!]) \\ &= \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi_1]\!]) \cup \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi_2]\!]) && \text{(by Prop. 2.3.3)} \\ &= [\![\varphi_1[p'_c/p]]\!] \cup [\![\varphi_2[p'_c/p]]\!] && \text{(Ind. hypothesis)} \\ &= [\![\varphi_1[p'_c/p] \vee \varphi_2[p'_c/p]]\!] \\ &= [\![\varphi[p'_c/p]]\!] \end{aligned}$$

$\varphi = (\mathsf{TC}\, v_1, v_2.\varphi_1(v_1, v_2))(v, v')$. Let S be some store and β an assignment. We have:

$$\begin{aligned}
& (S,\beta) \in \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi]\!]) \\
\iff & \exists u_1,\ldots,u_n \in U : \beta\, v = u_1, \beta\, v' = u_n \text{ and} \\
& \forall 1 \le i < n : \mathsf{post}_c(S), \beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi_1(v_1,v_2) \\
& (\xrightarrow{c} \text{total and deterministic}) \\
\iff & \exists u_1,\ldots,u_n \in U : \beta\, v = u_1, \beta\, v' = u_n \text{ and} \\
& \forall 1 \le i < n : (\mathsf{post}_c(S), \beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) \in [\![\varphi_1(v_1,v_2)]\!] \\
\iff & \exists u_1,\ldots,u_n \in U : \beta\, v = u_1, \beta\, v' = u_n \text{ and} \\
& \forall 1 \le i < n : \mathsf{ext\text{-}post}_c((S, \beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])) \in [\![\varphi_1(v_1,v_2)]\!] \\
& (\text{Def. of } \mathsf{ext\text{-}post}_c) \\
\iff & \exists u_1,\ldots,u_n \in U : \beta\, v = u_1, \beta\, v' = u_n \text{ and} \\
& \forall 1 \le i < n : (S, \beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) \in \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi_1(v_1,v_2)]\!]) \\
& (\text{by Prop. 2.3.2}) \\
\iff & \exists u_1,\ldots,u_n \in U : \beta\, v = u_1, \beta\, v' = u_n \text{ and} \\
& \forall 1 \le i < n : (S, \beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) \in [\![\varphi_1[p'_c/p](v_1,v_2)]\!] \\
& (\text{Ind. hypothesis}) \\
\iff & \exists u_1,\ldots,u_n \in U : \beta\, v = u_1, \beta\, v' = u_n \text{ and} \\
& \forall 1 \le i < n : S, \beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi_1[p'_c/p](v_1,v_2) \\
\iff & S,\beta \models \varphi[p'_c/p] \\
\iff & (S,\beta) \in [\![\varphi[p'_c/p]]\!]
\end{aligned}$$

All remaining cases follow similar argumentations. □

Proposition 2.3.7. *If $\xrightarrow{c}$ is total and deterministic then for a store S, $\mathsf{FO}^{\mathsf{TC}}$ formula φ and assignment β, we have:*

$$S,\beta \models \mathsf{wlp}_c(\varphi) \iff \mathsf{post}_c(S), \beta \models \varphi.$$

Proof.

$$\begin{aligned}
S,\beta \models \mathsf{wlp}_c(\varphi) \iff & (S,\beta) \in \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c([\![\varphi]\!]) && (\text{Def. of } \mathsf{wlp}_c \text{ and Prop. 2.3.5}) \\
\iff & \forall S' : S \xrightarrow{c} S' \Rightarrow (S',\beta) \in [\![\varphi]\!] && (\text{Def. of } \mathsf{ext\text{-}}\widetilde{\mathsf{pre}}_c) \\
\iff & (\mathsf{post}_c(S),\beta) \in [\![\varphi]\!] && (\xrightarrow{c} \text{total and determinstic}) \\
\iff & \mathsf{post}_c(S),\beta \models \varphi
\end{aligned}$$

□

Proposition 3.1.6. *The function γ is a complete meet-morphism and a complete join-morphism.*

Proof. Follows easily from the semantics of $\mathsf{FO}^{\mathsf{TC}}$ formulas. □

Proposition 3.1.8. *Abstraction function α and concretisation function γ form a Galois connection between the posets $\langle 2^{Store}, \subseteq\rangle$ and $\langle AbsDom, \models\rangle$.*

Proof. By Proposition 3.1.6 we know that the function γ is a complete meet-morphism. Thus, by definition of α and Proposition 2.1.9 we get that α and γ form a Galois connection. □

Theorem 3.1.10 (Characterization of Best Abstraction). *Let $\mathcal{M}$ be a set of program stores. The image of $\mathcal{M}$ under α is characterized as follows:*

$$\alpha(\mathcal{M}) \models\!\!\!\dashv \bigvee_{S \in \mathcal{M}} \forall v. \bigvee_{u \in U^S} P_{S,u}(v).$$

Proof. Since α and γ form a Galois connection, we know by Proposition 2.1.9 that α distributes over set union on the concrete lattice, i.e. we have:

$$\alpha(\mathcal{M}) = \bigvee_{S \in \mathcal{M}} \bigwedge \{ \Psi \in \mathit{AbsDom} \mid S \models \Psi \}.$$

Thus, it suffices to consider the abstraction of a single store $S \in \mathcal{M}$ and it remains to prove:

$$\alpha(S) \models\!\!\!\dashv \forall v. \bigvee_{u \in U^S} P_{S,u}(v).$$

From the definition of $P_{S,u}$ for $u \in U^S$ it is clear that we have:

$$S \models \forall v. \bigvee_{u \in U^S} P_{S,u}(v).$$

Hence, we can immediately conclude:

$$\alpha(S) \models \forall v. \bigvee_{u \in U^S} P_{S,u}(v).$$

It remains to prove the right-to-left direction of the congruence. The meet of all formulas in AbsDom that are valid in S must be a single abstract store, rather then a disjunction of abstract stores. If it was a disjunction of abstract stores then S would already satisfy one of its disjuncts contradicting the fact that it is the meet. Hence, there must be some formula over node predicates $\psi(v) \in \mathcal{F}_{\mathit{Pred}}$ such that:

$$\bigwedge \{ \Psi \in \mathit{AbsDom} \mid S \models \Psi \} \quad \models\!\!\!\dashv \quad \forall v. \psi(v).$$

Let $S' \in \mathit{Store}$ such that

$$S' \models \forall v. \bigvee_{u \in U^S} P_{S,u}(v).$$

Given some node $u' \in U^{S'}$, there must be some node $u \in U^S$ such that $S', u' \models P_{S,u}(v)$. Since $P_{S,u}(v)$ is a monomial over Pred satisfied for u in S, it must entail any other Boolean combination of node predicates that is satisfied for u in S. In particular we must have $P_{S,u}(v) \models \psi(v)$. From this we can infer that $S', u' \models \psi(v)$ holds, too. Since u' was chosen free in $U^{S'}$, we have $S' \models \forall v. \psi(v)$ and thus $S' \models \alpha(S)$. Hence, we have:

$$\forall v. \bigvee_{u \in U^S} P_{S,u}(v) \models \alpha(S).$$

$\square$

Proposition 3.2.3. *For a given context Γ, the context-sensitive abstract operators have the following properties:*

(i) $\mathsf{post}^{\#}_{\Gamma}$ *and* $\mathsf{wlp}^{\#}_{\Gamma}$ *form a Galois connection between the two posets* $\langle \mathcal{F}_{\mathit{Pred}}, \models_{\Gamma} \rangle$ *and* $\langle \mathcal{F}_{\mathit{Pred}}, \models_{\mathsf{post}(\gamma(\Gamma))} \rangle$*, formally:*

$$\forall \varphi, \psi \in \mathcal{F}_{\mathit{Pred}} : \mathsf{post}^{\#}_{\Gamma}(\varphi) \models_{\mathsf{post}(\gamma(\Gamma))} \psi \iff \varphi \models_{\Gamma} \mathsf{wlp}^{\#}_{\Gamma}(\psi),$$

(ii) $\mathsf{post}^{\#}_{\Gamma}$ *and* $\mathsf{wlp}^{\#}_{\Gamma}$ *are monotone,*

(iii) $\mathsf{post}^{\#}_{\Gamma} \circ \mathsf{wlp}^{\#}_{\Gamma}$ *is reductive,*

(iv) $\mathsf{wlp}^{\#}_{\Gamma} \circ \mathsf{post}^{\#}_{\Gamma}$ *is extensive,*

(v) $\mathsf{post}^{\#}_{\Gamma}$ *distributes over disjunctions,*

(vi) $\mathsf{wlp}^{\#}_{\Gamma}$ *distributes over conjunctions.*

Proof. We prove properties *(ii)*, *(iii)*, and *(iv)*. Properties *(i)*, *(v)*, and *(vi)* then follow from the general properties of Galois connections given in Proposition 2.1.9.
Proof of *(ii)*. Let $\varphi_1, \varphi_2 \in \mathcal{F}_{Pred}$. In order to prove monotonicity of $\mathsf{post}^{\#}_{\Gamma}$, assume $\varphi_1 \models_{\Gamma} \varphi_2$. By definition of $\mathsf{post}^{\#}_{\Gamma}(\varphi_2)$ we have $\varphi_2 \models_{\Gamma} \mathsf{wlp}(\mathsf{post}^{\#}_{\Gamma}(\varphi_2))$. Thus, by assumption, we can conclude $\varphi_1 \models_{\Gamma} \mathsf{wlp}(\mathsf{post}^{\#}_{\Gamma}(\varphi_2))$. However, $\mathsf{post}^{\#}_{\Gamma}(\varphi_1)$ is the conjunction of all formulas $\psi \in \mathcal{F}_{Pred}$ satisfying $\varphi_1 \models_{\Gamma} \mathsf{wlp}(\psi)$, i.e. we have $\mathsf{post}^{\#}_{\Gamma}(\varphi_1) \models \mathsf{post}^{\#}_{\Gamma}(\varphi_2)$. This implies $\mathsf{post}^{\#}_{\Gamma}(\varphi_1) \models_{\mathsf{post}(\gamma(\Gamma))} \mathsf{post}^{\#}_{\Gamma}(\varphi_2)$. Hence, $\mathsf{post}^{\#}_{\Gamma}$ is monotone.

In order to prove monotonicity of $\mathsf{wlp}^{\#}_{\Gamma}$, assume $\varphi_1 \models_{\mathsf{post}(\gamma(\Gamma))} \varphi_2$. Let S be some store such that $S \models \Gamma$ and β an assignment. We have:

$$\begin{array}{rll} & S, \beta \models \mathsf{wlp}(\varphi_1) & \\ \Leftrightarrow & \mathsf{post}(S), \beta \models \varphi_1 & \text{(by Prop. 2.3.7)} \\ \Rightarrow & \mathsf{post}(S), \beta \models \varphi_2 & \text{(by assumption)} \\ \Leftrightarrow & S, \beta \models \mathsf{wlp}(\varphi_2) & \text{(by Prop. 2.3.7)} \end{array}$$

Thus, we have

$$\mathsf{wlp}(\varphi_1) \models_{\Gamma} \mathsf{wlp}(\varphi_2) \tag{1}$$

By definition of $\mathsf{wlp}^{\#}_{\Gamma}$ we have $\mathsf{wlp}^{\#}_{\Gamma}(\varphi_1) \models \mathsf{wlp}(\varphi_1)$. With (1) we can conclude $\mathsf{wlp}^{\#}_{\Gamma}(\varphi_1) \models_{\Gamma} \mathsf{wlp}(\varphi_2)$. However, by definition of $\mathsf{wlp}^{\#}_{\Gamma}$ we know that $\mathsf{wlp}^{\#}_{\Gamma}(\varphi_2)$ is the disjunction of all formulas $\psi \in \mathcal{F}_{Pred}$ satisfying $\psi \models \mathsf{wlp}(\varphi_2)$, i.e. we have $\mathsf{wlp}^{\#}_{\Gamma}(\varphi_1) \models_{\Gamma} \mathsf{wlp}^{\#}_{\Gamma}(\varphi_2)$. Therefore $\mathsf{wlp}^{\#}_{\Gamma}$ is monotone.

Proof of *(iii)*. Let $\varphi \in \mathcal{F}_{Pred}$. By definition of $\mathsf{post}^{\#}_{\Gamma}$ we know $\mathsf{post}^{\#}_{\Gamma}(\mathsf{wlp}^{\#}_{\Gamma}(\varphi))$ is the conjunction of all $\psi \in \mathcal{F}_{Pred}$, such that $\mathsf{wlp}^{\#}_{\Gamma}(\varphi) \models_{\Gamma} \mathsf{wlp}(\psi)$. Moreover, by definition of $\mathsf{wlp}^{\#}_{\Gamma}$ we have $\mathsf{wlp}^{\#}_{\Gamma}(\varphi) \models_{\Gamma} \mathsf{wlp}(\varphi)$. Thus, we have $\mathsf{post}^{\#}_{\Gamma}(\mathsf{wlp}^{\#}_{\Gamma}(\varphi)) \models_{\mathsf{post}(\gamma(\Gamma))} \varphi$, i.e. $\mathsf{post}^{\#}_{\Gamma} \circ \mathsf{wlp}^{\#}_{\Gamma}$ is reductive.

Proof of *(iv)*. Let $\varphi \in \mathcal{F}_{Pred}$. By definition of $\mathsf{wlp}^{\#}_{\Gamma}$ we know $\mathsf{wlp}^{\#}_{\Gamma}(\mathsf{post}^{\#}_{\Gamma}(\varphi))$ is the disjunction of all $\psi \in \mathcal{F}_{Pred}$, such that $\psi \models_{\Gamma} \mathsf{wlp}(\mathsf{post}^{\#}_{\Gamma}(\varphi))$. Moreover, by definition of $\mathsf{post}^{\#}_{\Gamma}$ we have $\varphi \models_{\Gamma} \mathsf{wlp}(\mathsf{post}^{\#}_{\Gamma}(\varphi))$. Thus, we have $\varphi \models_{\Gamma} \mathsf{wlp}^{\#}_{\Gamma}(\mathsf{post}^{\#}_{\Gamma}(\varphi))$, i.e. $\mathsf{wlp}^{\#}_{\Gamma} \circ \mathsf{post}^{\#}_{\Gamma}$ is extensive. □

Proposition 3.2.6. *Let $\Psi = \forall v.\psi$ be an abstract store. The image of Ψ under $\mathsf{post}^{\#}_{\mathsf{Cart}_1}$ is obtained by applying the context-sensitive post operator for Ψ to ψ:*

$$\mathsf{post}^{\#}_{\mathsf{Cart}_1}(\Psi) \models\!\mid \forall v.\, \mathsf{post}^{\#}_{\Psi}(\psi).$$

Proof. In the following, consider φ_1 and φ_2 to be defined as follows:

$$\begin{array}{rcl} \varphi_1 & \stackrel{def}{=} & \bigwedge\{\, \varphi \in \mathcal{F}_{Pred} \mid \mathsf{post}^{\#}(\Psi) \models \forall v.\varphi \,\} \\ \varphi_2 & \stackrel{def}{=} & \bigwedge\{\, \varphi \in \mathcal{F}_{Pred} \mid \psi \models_{\Psi} \mathsf{wlp}(\varphi) \,\} \end{array}$$

In order to prove the proposition, we have to show that φ_1 and φ_2 are equivalent.

"$\varphi_2 \models \varphi_1$": Let S be some store such that S is a model of Ψ, i.e. for all nodes u we have $S, u \models \psi$. Now assume there was some node u such that $S, u \not\models \mathsf{wlp}(\varphi_1)$. We have:

$$\begin{aligned} S, u \not\models \mathsf{wlp}(\varphi_1) &\Rightarrow \mathsf{post}(S), u \not\models \varphi_1 && \text{(by Prop. 2.3.7)} \\ &\Rightarrow \mathsf{post}(S) \not\models \forall v.\varphi_1 \\ &\Rightarrow \mathsf{post}^\#(\Psi) \not\models \forall v.\varphi_1 && (S \models \Psi \text{ implies } \mathsf{post}(S) \models \mathsf{post}^\#(\Psi)) \end{aligned}$$

This contradicts the definition of φ_1. Thus, for all nodes u we have $S, u \models \mathsf{wlp}(\varphi_1)$ and therefore $\psi \models_\Psi \mathsf{wlp}(\varphi_1)$. Since φ_2 is the greatest lower bound of all those formulas, we have $\varphi_2 \models \varphi_1$.

"$\varphi_1 \models \varphi_2$": Let S be some store such that $S \models \mathsf{post}^\#(\Psi)$, but assume $S \not\models \forall v.\varphi_2$. Then there is some node u such that $S, u \not\models \varphi_2$. If $P_{S,u}$ is the abstract node of u in S then we must have $P_{S,u} \models \neg\varphi_2$.

Case 1: there is some store S' which is a model of Ψ and some node u' such that $\mathsf{post}(S'), u' \models P_{S,u}$. We have:

$$\begin{aligned} \mathsf{post}(S'), u' \models P_{S,u} &\Rightarrow \mathsf{post}(S'), u' \not\models \varphi_2 && (P_{S,u} \models \neg\varphi_2) \\ &\Rightarrow S', u' \not\models \mathsf{wlp}(\varphi_2) && \text{(by Prop. 2.3.7)} \end{aligned}$$

which contradicts the definition of φ_2, since $S', u' \models \psi$.

Case 2: for all stores S' either $S' \not\models \Psi$ or for all nodes u' we have $\mathsf{post}(S'), u' \not\models P_{S,u}$. From this we can conclude:

$$\forall S' : S' \models \Psi \Rightarrow \forall u' \in U : \mathsf{post}(S'), u' \models \varphi_2 \wedge \neg P_{S,u}$$

and again by Prop. 2.3.7 we get:

$$\psi \models_\Psi \mathsf{wlp}(\varphi_2 \wedge \neg P_{S,u})$$

which contradicts the definition of φ_2.

Thus, altogether we can conclude:

$$\mathsf{post}^\#(\Psi) \models \forall v.\varphi_2$$

and we finally get $\varphi_1 \models \varphi_2$. □

Theorem 3.2.9 (Soundness of Cartesian Post). *The operator* $\mathsf{post}^\#_{\mathsf{Cart}}$ *is an approximation of* $\mathsf{post}^\#$*:*

$$\forall \Psi \in \mathit{AbsStore} : \mathsf{post}^\#(\Psi) \models \mathsf{post}^\#_{\mathsf{Cart}}(\Psi).$$

Proof. Let Ψ be some abstract store of the form $\forall v. \bigvee_i P_i$. By definition of α_{Cart_1} it is clear that α_{Cart_1} is reductive, i.e. we immediately observe:

$$\mathsf{post}^\#(\Psi) \models \alpha_{\mathsf{Cart}_1} \circ \mathsf{post}^\#(\Psi) = \mathsf{post}^\#_{\mathsf{Cart}_1}(\Psi).$$

By Proposition 3.2.6 and the fact that $\mathsf{post}^\#_\Psi$ distributes over disjunctions we get:

$$\mathsf{post}^\#(\Psi) \models \mathsf{post}^\#_{\mathsf{Cart}_1}(\Psi) = \forall v. \bigvee_i \mathsf{post}^\#_\Psi(P_i).$$

Again from its definition it is clear that α_{Cart_2} is reductive. Thus, we finally get:

$$\mathsf{post}^\#(\Psi) \models \forall v. \bigvee_i \alpha_{\mathsf{Cart}_2} \circ \mathsf{post}^\#_\Psi(P_i) = \mathsf{post}^\#_{\mathsf{Cart}}(\Psi).$$

□

Theorem 3.2.10 (Characterization of Cartesian Post). *Let $\Psi = \forall v. \bigvee_i P_i$ be an abstract store. The Cartesian post of Ψ is characterized as follows:*

$$\mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) = \forall v. \bigvee_i \bigwedge \{\, p \in \mathit{Pred} \mid P_i \models_\Psi \mathsf{wlp}^{\#}_{\Psi}(p) \,\}.$$

Proof. We have:

$$\begin{aligned}
\mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) &= \forall v. \bigvee_i \alpha_{\mathsf{Cart}_2} \circ \mathsf{post}^{\#}_{\Psi}(P_i) \\
&\models\!\mid \forall v. \bigvee_i \{\, p \in \mathit{Pred} \mid \mathsf{post}^{\#}_{\Psi}(P_i) \models_{\mathsf{post}(\Psi)} p \,\} \\
&\models\!\mid \forall v. \bigvee_i \{\, p \in \mathit{Pred} \mid P_i \models_\Psi \mathsf{wlp}^{\#}_{\Psi}(p) \,\} && \text{(by Prop. 3.2.3)}
\end{aligned}$$

□

Proposition 3.2.12. *Let Ψ be an abstract store. If $\mathsf{post}^{\#}$ is deterministic with respect to Ψ then $\mathsf{post}^{\#}_{\mathsf{Cart}}$ does not lose precision with respect to $\mathsf{post}^{\#}$, i.e.*

$$\mathsf{post}^{\#}(\Psi) \models\!\mid \mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi).$$

Proof. Let Ψ be some abstract store with $\Psi = \forall v. \bigvee_i P_i$, where each P_i is a monomial. Assume $\mathsf{post}^{\#}$ is deterministic with respect to Ψ, then we have:

$$\mathsf{post}^{\#}(\forall v. \bigvee_i P_i) = \forall v. \bigvee_i P_i' \quad \text{where for all } i \quad P_i' = \mathsf{post}^{\#}_{\Psi}(P_i).$$

and each P_i' is a monomial. By Definition of $\mathsf{post}^{\#}_{\mathsf{Cart}}$ we have:

$$\begin{aligned}
\mathsf{post}^{\#}_{\mathsf{Cart}}(\Psi) &\models\!\mid \forall v. \bigvee_i \bigwedge_i \{\, p \in \mathit{Pred} \mid \mathsf{post}^{\#}_{\Psi}(P_i) \models_{\mathsf{post}(\Psi)} p \,\} \\
&\models\!\mid \forall v. \bigvee_i \bigwedge_i \{\, p \in \mathit{Pred} \mid P_i' \models_{\mathsf{post}(\Psi)} p \,\} \\
&\models\!\mid \forall v. \bigvee_i \bigwedge_i P_i' && (P_i' \text{ monomial}) \\
&\models\!\mid \mathsf{post}^{\#}(\Psi)
\end{aligned}$$

□

Proposition 3.2.13. *Let S be a store. The operator $\mathsf{post}^{\#}$ is deterministic with respect to $\alpha(S)$ if and only if for all node predicates p in Pred we have:*

$$\mathsf{wlp}(p) \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p).$$

Proof. For the *only if* direction, let S be some store with $\alpha(S) = \forall v. \bigvee_i P_i$ where each P_i is a monomial. Assume there is some $p \in \mathit{Pred}$ such that:

$$\mathsf{wlp}(p) \not\models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p).$$

Thus, there is some store S' and node u' such that $S' \models \alpha(S)$ and $S', u' \models \mathsf{wlp}(p)$ but $S', u' \not\models \mathsf{wlp}^{\#}_{\alpha(S)}(p)$. Since $S' \models \alpha(S)$ there must be some i such that $S', u' \models P_i$. Then we have:

$$S', u' \models P_i \text{ and } S', u' \not\models \mathsf{wlp}^{\#}_{\alpha(S)}(p) \;\Rightarrow\; P_i \not\models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p)$$

$$\Rightarrow \ \mathsf{post}^{\#}_{\alpha(S)}(P_i) \not\models_{\mathsf{post}(\alpha(S))} p$$

but we have in addition:

$$\begin{aligned} S', u' \models \mathsf{wlp}(p) &\Rightarrow \mathsf{post}(S'), u' \models p \\ &\Rightarrow \mathsf{post}^{\#}_{\alpha(S)}(P_i) \not\models_{\mathsf{post}(\alpha(S))} \neg p \end{aligned}$$

Hence, $\mathsf{post}^{\#}_{\alpha(S)}(P_i)$ is not a monomial and $\mathsf{post}^{\#}$ is not deterministic with respect to $\alpha(S)$. In order to prove the *if* direction, assume that for all node predicates p in *Pred* we have:

$$\mathsf{wlp}(p) \models_{\Psi} \mathsf{wlp}^{\#}_{\Psi}(p).$$

We show that the following two properties hold:

(i) for all i : $\mathsf{post}^{\#}_{\alpha(S)}(P_i)$ is a monomial,

(ii) $\forall v. \bigvee_i \mathsf{post}^{\#}_{\alpha(S)}(P_i) \models \alpha(\mathsf{post}(S))$.

From (ii) and the fact that $\alpha(\mathsf{post}(S)) \models \mathsf{post}^{\#}(\alpha(S))$ holds we can conclude:

$$\forall v. \bigvee_i \mathsf{post}^{\#}_{\alpha(S)}(P_i) \models \mathsf{post}^{\#}(\alpha(S)).$$

Together with (i) this gives us that $\mathsf{post}^{\#}$ is deterministic with respect to $\alpha(S)$.
Proof of (i). Let P_i be some abstract node in $\alpha(S)$. We have for any node predicate p:

$$\begin{array}{lll} & \text{either} \quad P_i \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p) & \\ & \text{or} \quad P_i \models_{\alpha(S)} \neg\, \mathsf{wlp}^{\#}_{\alpha(S)}(p) & (P_i \text{ monomial}) \\ \Rightarrow & \text{either} \quad P_i \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p) & \\ & \text{or} \quad P_i \models_{\alpha(S)} \neg\, \mathsf{wlp}(p) & (\mathsf{wlp}(p) \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p)) \\ \Leftrightarrow & \text{either} \quad P_i \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p) & \\ & \text{or} \quad P_i \models_{\alpha(S)} \mathsf{wlp}(\neg p) & \\ \Rightarrow & \text{either} \quad P_i \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(p) & \\ & \text{or} \quad P_i \models_{\alpha(S)} \mathsf{wlp}^{\#}_{\alpha(S)}(\neg p) & (\mathsf{wlp}^{\#}_{\alpha(S)}(\neg p) \models_{\alpha(S)} \mathsf{wlp}(\neg p)) \\ \Leftrightarrow & \text{either} \quad \mathsf{post}^{\#}_{\alpha(S)}(P_i) \models_{\alpha(S)} p & \\ & \text{or} \quad \mathsf{post}^{\#}_{\alpha(S)}(P_i) \models_{\alpha(S)} \neg p & \end{array}$$

Thus, $\mathsf{post}^{\#}_{\alpha(S)}(P_i)$ is a monomial.
Proof of (ii). Let P_i be some abstract node in $\alpha(S)$. By definition of α we know there is some node u such that $S, u \models P_i$. Moreover, we know that both:

$$\mathsf{post}(S), u \models \mathsf{post}^{\#}_{\alpha(S)}(P_i) \ \text{ and } \ \mathsf{post}(S), u \models P_{\mathsf{post}(S),u}$$

hold. Since both $\mathsf{post}^{\#}_{\alpha(S)}(P_i)$ and $P_{\mathsf{post}(S),u}$ are monomials over *Pred* we must have:

$$\mathsf{post}^{\#}_{\alpha(S)}(P_i) \models\!\!\!\!\dashv P_{\mathsf{post}(S),u}.$$

Thus, we can conclude:

$$\forall v. \bigvee_i \mathsf{post}^{\#}_{\alpha(S)}(P_i) \models \forall v. \bigvee_{u \in U} P_{\mathsf{post}(S),u} = \alpha(\mathsf{post}(S)).$$

□

Proposition 3.2.15. $\mathsf{post}^\#$ *is deterministic if and only if the set of node predicates Pred is closed under* wlp.

Proof. Follows directly from Definition 3.2.14 and Proposition 3.2.13. □

Proposition 3.2.19 (Soundness of Cartesian Post with Splitting). *Let* $\mathsf{split}_c[\mathcal{P}]$ *be a splitting operator for* $\mathcal{P}$ *and command c. The Cartesian post operator with splitting* $\mathsf{post}^\#_{c,\mathsf{split}_c[\mathcal{P}]}$ *is an approximation of* $\mathsf{post}^\#_c$ *on* $AbsDom[Pred]$*:*

$$\forall \Psi \in AbsStore[Pred] : \ \mathsf{post}^\#_c(\Psi) \models \mathsf{post}^\#_{c,\mathsf{split}_c[\mathcal{P}]}(\Psi).$$

Proof. Let us first define the following abbreviation:

$$Pred' \overset{def}{=} Pred \cup \mathsf{wlp}_c(\mathcal{P}).$$

Let $\mathsf{split}_c[\mathcal{P}]$ be a splitting operator and let Ψ be some abstract store over $Pred$. We have:

$$\begin{aligned}
\mathsf{post}^\#_c(\Psi) &= \alpha[Pred] \circ \mathsf{post}_c \circ \gamma(\Psi) \\
&\models \alpha[Pred] \circ \gamma \circ \alpha[Pred'] \circ \mathsf{post}_c \circ \gamma \circ \alpha[Pred'] \circ \gamma(\Psi) && (\gamma \circ \alpha \text{ extensive}) \\
&= \alpha[Pred] \circ \gamma \circ \mathsf{post}^\#_c[Pred'] \circ \alpha[Pred'] \circ \gamma(\Psi) \\
&\models \alpha[Pred] \circ \gamma \circ \mathsf{post}^\#_{c,\mathsf{Cart}}[Pred'] \circ \alpha[Pred'] \circ \gamma(\Psi) && (\text{by Thm. 3.2.9}) \\
&\models \alpha[Pred] \circ \gamma \circ \mathsf{post}^\#_{c,\mathsf{Cart}}[Pred'] \circ \mathsf{split}_c[\mathcal{P}](\Psi) && (\text{by def. of } \mathsf{split}_c[\mathcal{P}]) \\
&= \mathsf{post}^\#_{c,\mathsf{split}_c[\mathcal{P}]}(\Psi).
\end{aligned}$$

□

Proposition 3.2.20. *The Cartesian post with splitting for the most precise splitting operator* $\mathsf{split}^\#_c[Pred]$ *coincides with the best abstract post operator on the abstract domain* $AbsDom[Pred]$*.*

Proof. By Proposition 3.2.19 it suffices to show:

$$\forall \Psi \in AbsStore[Pred] : \ \mathsf{post}^\#_{c,\mathsf{split}^\#_c[Pred]}(\Psi) \models \mathsf{post}^\#_c(\Psi).$$

Let us first define the following abbreviation:

$$Pred' \overset{def}{=} Pred \cup \mathsf{wlp}_c(Pred).$$

Now let Ψ be an abstract store over node predicates $Pred$. By definition of α and $\mathsf{post}^\#_{c,\mathsf{Cart}}$ we have:

$$\begin{aligned}
&\mathsf{post}^\#_{c,\mathsf{Cart}}[Pred'] \circ \alpha[Pred'] \circ \gamma(\Psi) \\
\mathrel{=\!\!|\!\!\models}\ & \bigvee_{S \in \gamma(\Psi)} \forall v.\ \bigvee_{u \in U} \bigwedge \{\, p \in Pred' \mid P_{S,u} \models_{\alpha[Pred'](S)} \mathsf{wlp}^\#_{c,\alpha[Pred'](S)}(p) \,\}
\end{aligned}$$

where each $P_{S,u}$ is the monomial over $Pred'$ such that $S, u \models P_{S,u}$. Now we have:

$$\begin{aligned}
& \bigvee_{S \in \gamma(\Psi)} \forall v.\ \bigvee_{u \in U} \bigwedge \{\, p \in Pred' \mid P_{S,u} \models_{\alpha[Pred'](S)} \mathsf{wlp}^\#_{c,\alpha[Pred'](S)}(p) \,\} \\
\models\ & \bigvee_{S \in \gamma(\Psi)} \forall v.\ \bigvee_{u \in U} \bigwedge \{\, p \in Pred \mid P_{S,u} \models_{\alpha[Pred'](S)} \mathsf{wlp}^\#_{c,\alpha[Pred'](S)}(p) \,\} \\
& (Pred \subseteq Pred')
\end{aligned}$$

$$\begin{aligned}
&\models \bigvee_{S \in \gamma(\Psi)} \forall v. \bigvee_{u \in U} \bigwedge \{\, p \in \mathit{Pred} \mid P_{S,u} \models_{\alpha[\mathit{Pred}'](S)} \mathsf{wlp}_c(p) \,\} \\
&\models\!\!\!\!| \bigvee_{S \in \gamma(\Psi)} \forall v. \bigvee_{u \in U} \bigwedge \{\, p \in \mathit{Pred} \mid S, u \models \mathsf{wlp}_c(p) \,\} \\
&\quad (P_{S,u} \text{ monomial over } \mathit{Pred}' \text{ and } \mathsf{wlp}_c(p) \in \mathit{Pred}') \\
&\models\!\!\!\!| \bigvee_{S \in \gamma(\Psi)} \forall v. \bigvee_{u \in U} \bigwedge \{\, p \in \mathit{Pred} \mid \mathsf{post}_c(S), u \models p \,\} \\
&\quad (\text{by Prop. 2.3.7}) \\
&\models\!\!\!\!| \alpha[\mathit{Pred}] \circ \mathsf{post}_c \circ \gamma(\Psi) \\
&= \mathsf{post}_c^{\#}(\Psi)
\end{aligned}$$

Thus, we have:

$$\forall \Psi \in \mathit{AbsStore}[\mathit{Pred}] : \mathsf{post}^{\#}_{c,\mathsf{Cart}}[\mathit{Pred}'] \circ \alpha[\mathit{Pred}'] \circ \gamma(\Psi) \models \mathsf{post}_c^{\#}(\Psi).$$

Since $\alpha[\mathit{Pred}] \circ \gamma$ is reductive with respect to $\models$ on $\mathit{AbsDom}[\mathit{Pred}]$, as well as monotone with respect to $\models$ in general, we have for any $\mathsf{FO}^{\mathsf{TC}}$ formula Φ and abstract value Ψ in $\mathit{AbsDom}[\mathit{Pred}]$:

$$\Phi \models \Psi \text{ implies } \alpha[\mathit{Pred}] \circ \gamma(\Phi) \models \alpha[\mathit{Pred}] \circ \gamma(\Psi) \models \Psi.$$

From this we can conclude:

$$\forall \Psi \in \mathit{AbsStore}[\mathit{Pred}] : \alpha[\mathit{Pred}] \circ \gamma \circ \mathsf{post}^{\#}_{c,\mathsf{Cart}}[\mathit{Pred}'] \circ \alpha[\mathit{Pred}'] \circ \gamma(\Psi) \models \mathsf{post}_c^{\#}(\Psi)$$

which proves the proposition. □

Proposition 3.2.21. *Let $\mathsf{split}_c[\mathcal{P}]$ be a splitting operator. If $\mathsf{post}_c^{\#}$ is deterministic then $\mathsf{post}_c^{\#}$ and the Cartesian post with splitting for $\mathsf{split}_c[\mathcal{P}]$ coincide.*

Proof. Follows immediately from Proposition 3.2.12 and the fact that the Cartesian post on the extended abstract domain $\mathit{AbsDom}[\mathit{Pred} \cup (\mathsf{wlp}_c\ \mathcal{P})]$ is at least as precise as the one on the original abstract domain $\mathit{AbsDom}[\mathit{Pred}]$. □

Proposition 4.2.3. *Let S be a finite structure over Σ and $u \in U^S$. The modal node predicates $p.\mathsf{U}_{\langle R \rangle}.q$, $p.\mathsf{S}_{\langle R \rangle}.q$, $p.\mathsf{U}_{[R]}.q$, and $p.\mathsf{S}_{[R]}.q$ are characterized as follows:*

- $S, u \models p.\mathsf{U}_{\langle R \rangle}.q(v) \iff$ *there is an R-path π in S starting in u such that:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$
- $S, u \models p.\mathsf{S}_{\langle R \rangle}.q(v) \iff$ *there is an R^{-1}-path π in S starting in u such that:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$
- $S, u \models p.\mathsf{U}_{[R]}.q(v) \iff$ *for all R-paths π in S starting in u:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$
- $S, u \models p.\mathsf{S}_{[R]}.q(v) \iff$ *for all R^{-1}-paths π in S starting in u:* $\exists i \geq 1 : S, \pi(i) \models q(v)$ *and* $\forall j < i : S, \pi(j) \models p(v)$

Proof. We give the proof for $p.\mathsf{S}_{[R]}.q$ explicitly. The proof for $q.\mathsf{U}_{[R]}.q$ is analogous, $p.\mathsf{U}_{\langle R \rangle}.q$ and $p.\mathsf{S}_{\langle R \rangle}.q$ are simple.

Let $S = \langle U, \iota \rangle$ be a finite structure over Σ and let $u \in U$.
"$\Rightarrow$" Assume $S, u \models p.\mathsf{S}_{[R]}.q(v)$. Let π be some R^{-1}-path starting in u. Assume that for all $i \geq 1$ we have $S, \pi(i) \models \neg q(v)$.

Case 1: there is $k \geq 1$ such that for all $n \in R, u \in U : (\iota\ n)(u, \pi(k)) = 0$. Then we have:

$$S, [v \mapsto u, v' \mapsto \pi(k)] \models (\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v', v) \vee v \approx v'$$
$$\text{and } S, \pi(k) \models \neg q(v) \wedge \forall v'.r^+(v', v) \rightarrow \neg q(v')$$

From this we can conclude $S, u \not\models p.\mathsf{S}_{[R]}.q(v)$ which gives us a contradiction.

Case 2: for all $i \geq 1$ there is $n \in R$ such that $(\iota\ n)(\pi(i+1), \pi(i)) = 1$. Since U is finite, there must be a cycle in π, i.e. there are k_1, k_2 such that $k_1 < k_2$ and $\pi(k_1) = \pi(k_2)$. Then we have:

$$S, [v \mapsto u, v' \mapsto \pi(k_1)] \models (\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v, v') \vee v \approx v'$$
$$\text{and } S, \pi(k_1) \models \neg q(v) \wedge (\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v, v)$$

this implies $S, u \not\models p.\mathsf{S}_{[R]}.q(v)$ which again gives us a contradiction.

Thus, we know there is at least one $i \geq 1$ such that $S, \pi(i) \models q(v)$. Let i_{min} be the smallest of all those i. Assume there is $j < i_{min}$ such that $S, \pi(j) \models \neg p(v)$. Then we have:

$$S, [v \mapsto u, v' \mapsto \pi(j)] \models (\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v, v') \vee v \approx v'$$
$$\text{and } S, \pi(j) \models \neg q(v) \wedge \neg p(v)$$

This once more contradicts the fact that $S, u \models p.\mathsf{S}_{[R]}.q(v)$ holds. Hence, we have $S, \pi(i_{min}) \models q(v)$ and for all $j < i_{min} : S, \pi(j) \models p(v)$.

"$\Leftarrow$" Assume that for all R^{-1}-paths in S which start in u we have:

$$\exists i \geq 1 : \ S, \pi(i) \models q(v) \text{ and } \forall j < i : \ S, \pi(j) \models p(v) \tag{1}$$

Let $u' \in U$ such that

$$S, [v \mapsto u, v' \mapsto u'] \models (\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v, v') \vee v \approx v' \tag{2}$$

Case 1: $S, u' \models q(v)$. Done.

Case 2: $S, u' \models \neg q(v)$. From (2) we know there is an R^{-1}-path π starting in u such that for some $k \geq 1 : \pi(k) = u'$. Since by assumption π satisfies (1), there is some $i \geq k$ such that $S, \pi(j) \models q(v)$ and for all $j < i : S, \pi(j) \models p(v)$. Thus, we have:

$$S, u' \models p(v) \wedge \exists\, v'.r^+(v', v) \wedge q(v')$$

Now assume

$$S, u' \models (\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v, v)$$

then there is $m \geq 1$ and $u_0, \ldots, u_m \in U$ such that

- $u_0 = u_m = u' = \pi(k)$ and
- $\forall 1 \leq j < m : \ S, [v \mapsto u_j, v' \mapsto u_{j+1}] \models r(v, v') \wedge \neg q(v)$

Hence, we can construct an R^{-1}-path π' starting in u as follows:

$$\pi'(n) = \begin{cases} \pi(n) & \text{for } n \leq k \\ u_j & \text{for } n > k \text{ and } j = (n-k) \bmod m \end{cases}$$

However, by construction, π' does not satisfy (1) which contradicts the assumption. Hence, we must have:

$$S, u' \models \neg(\mathsf{TC}\, v, v'.r(v, v') \wedge \neg q(v))(v, v)$$

Since u' was chosen free in U, we can conclude $S, u \models p.\mathsf{S}_{[R]}.q(v)$. ☐

Proposition 4.3.4 (Correctness of Translation). *Let $p \in \mathsf{GMNP}[R]$ then p and $t(p)$ are equivalent on finite structures, i.e. for every finite structure S and $u \in U^S$:*

$$S, u \models p \iff S, u \models t(p).$$

Proof. The proof goes by structural induction on p. The only interesting cases are those involving fixed point operators. However, according to Proposition 4.2.3 these modal node predicates semantically correspond to CTL operators. The translation follows the standard translation of CTL to the modal mu-calculus. ☐

Proposition 4.4.1. *For commands c of the form* `x = y`, `x = NULL`, *and* `x = y->n`, *the class $\mathsf{MNP}[n]$ is closed under wlp_c. Formally, for any modal node predicate p there is a finite subset of modal node predicates Pred such that:*

$$\forall S \in \mathit{Store} : \mathsf{wlp}^{\#}_{c,\alpha(S)}(p) \models\mid \mathsf{wlp}_c(p).$$

Proof. For all those commands c and all modal node predicates p, we have that $\mathsf{wlp}_c(p)$ is contained in $\mathsf{MNP}[n]$. Hence, choose *Pred* such that is contains $\mathsf{wlp}_c(p)$. ☐

Lemma 4.4.3. *For any program variable* `x` *and any $\mathsf{FO}^{\mathsf{TC}}$ formula $\varphi(v)$:*

$$\forall S \in \mathit{Store} : S \models \exists v.x(v) \wedge \varphi(v) \iff S \models \forall v.x(v) \rightarrow \varphi(v).$$

Proof. Follows immediately from the fact that all program stores S interpret x as a singleton. ☐

Proposition 4.4.4. *For commands c of the form* `x->n = y` *and any modal node predicate p, $f_c(p)$ and $\mathsf{wlp}_c(p)$ are equivalent on program stores, i.e.*

$$\forall S \in \mathit{Store}, u \in U : S, u \models f_c(p) \iff S, u \models \mathsf{wlp}_c(p).$$

Proof. By induction on the structure of p. Let S be a store and $u \in U^S$ some node. The proof is quite tedious. We only sketch two of the possible cases, here. All remaining cases are either trivial or follow similar argumentation.

$p = \langle n \rangle.q$:

$$\begin{aligned}
& S, u \models \mathsf{wlp}_c(\langle n \rangle.q) \\
\iff & S, u \models \mathsf{wlp}_c(\exists v'.q(v') \wedge n(v, v')) \\
\iff & S, u \models \exists v'.\,\mathsf{wlp}_c(q)(v')) \wedge (n(v, v') \wedge \neg x(v) \vee x(v) \wedge y(v')) && \text{(Def. of } \mathsf{wlp}_c\text{)} \\
\iff & S, u \models \exists v'.f_c(q)(v')) \wedge (n(v, v') \wedge \neg x(v) \vee x(v) \wedge y(v')) && \text{(Ind. hypotheses)} \\
\iff & S, u \models \neg x(v) \wedge \exists v'.f_c(q)(v') \wedge n(v, v') \vee \\
& \qquad x(v) \wedge \exists v'.f_c(q)(v') \wedge y(v') \\
\iff & S, u \models \neg x(v) \wedge \exists v'.f_c(q)(v') \wedge n(v, v') \vee \\
& \qquad x(v) \wedge \forall v'.y(v') \rightarrow f_c(q)(v') && \text{(Lemma 4.4.3)} \\
\iff & S, u \models \neg x(v) \wedge \langle n \rangle.f_c(q) \vee x(v) \wedge \forall v'.y(v') \rightarrow f_c(q)(v') \\
\iff & S, u \models f_c(\langle n \rangle.q) && \text{(Def. of } f\text{)}
\end{aligned}$$

$p = p_1.\mathsf{U}_{\langle n \rangle}.p_2$:

$$S, u \models \mathsf{wlp}_c(p_1.\mathsf{U}_{\langle n \rangle}.p_2)$$

$\iff S, u \models \exists\, v'.\, \mathsf{wlp}_c(p_2)(v') \wedge ((\mathsf{TC}\, v_1, v_2.\, \mathsf{wlp}_c(p_1)(v_1) \wedge \mathsf{wlp}_c(n))(v, v') \vee v \approx v')$
(Def. of wlp_c)
$\iff S, u \models \exists\, v'.f_c(p_2)(v') \wedge ((\mathsf{TC}\, v_1, v_2.f_c(p_1)(v_1) \wedge \mathsf{wlp}_c(n))(v, v') \vee v \approx v')$
(Ind. hyp.)

where
$\mathsf{wlp}_c(n) = n(v_1, v_2) \wedge \neg x(v_1) \vee x(v_1) \wedge y(v_2)$

Assume $S, u \models \mathsf{wlp}_c(p)$ holds. If $S, u \models f_c(p_2)$ then we have:

$$S, u \models (\neg x \wedge p_1).\mathsf{U}_{\langle n \rangle}.p_2$$

and by definition of f we directly get $S, u \models f_c(p)$. Thus, assume that $S, u \not\models f_c(p_2)$ holds, then there exist $u_1, \ldots, u_n$ such that:

- $u = u_1$, and
- $S, u_n \models f_c(p_2)$, and
- for all $1 \leq i < n : S, [v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models f_c(p_1)(v_1) \wedge \mathsf{wlp}_c(n)(v_1, v_2)$.

Case 1: for all $1 \leq k < n : S, u_k \models \neg x(v)$. Then there are $u_1, \ldots, u_n$ such that:

- $u = u_1$, and
- $S, u_n \models f_c(p_2)$, and
- for all $1 \leq i < n$:
  $S, [v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models f_c(p_1)(v_1) \wedge n(v_1, v_2) \wedge \neg x(v_1)$.

Thus, we have $S, u \models (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.p_2$ and by definition of f we get $S, u \models f_c(p)$.

Case 2: there is at least one k with $1 \leq k < n$ such that $S, u_k \models x(v)$. Let k_{min} be the smallest and k_{max} be the greatest of all those k. Then there are nodes $u_1, \ldots, u_n \in U^S$ such that:

- $u = u_1$, and
- $S, u_{k_{min}} \models f_c(p_1)$, and
- for all $1 \leq i < k_{min}$:
  $S, [v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models f_c(p_1)(v_1) \wedge n(v_1, v_2) \wedge \neg x(v_1)$, and
- $S, u_{k_{max}+1} \models y(v)$, and
- $S, u_n \models f_c(p_2)$, and
- if $k_{max} + 1 < n$ then for all $k_{max} < i < n$:
  $S, [v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models f_c(p_1)(v_1) \wedge n(v_1, v_2) \wedge \neg x(v_1)$.

Thus, we get:

- $S, u \models (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.(p_1 \wedge x)(v)$, and
- $S, u_{k_{max+1}} \models y(v) \wedge (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.p_2(v)$.

From this we can conclude:

$$S, u \models (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.(p_1 \wedge x)(v) \wedge \exists\, v'.y(v') \wedge (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.p_2(v')$$

which by Lemma 4.4.3 is equivalent to:

$$S, u \models (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.(p_1 \wedge x)(v) \wedge \forall v'.y(v') \rightarrow (p_1 \wedge \neg x).\mathsf{U}_{\langle n \rangle}.p_2(v').$$

Hence, we finally get that $S, u \models f_c(p)$ and therefore:

$$\forall S \in \mathit{Store}, u \in U : S, u \models \mathsf{wlp}_c(p) \Rightarrow S, u \models f_c(p).$$

The left-to-right direction follows the same argumentation backwards. Starting from the fact that $f_c(p)$ is satisfied by u in S, we can construct a satisfying path for the transitive closure operator that occurs in $\mathsf{wlp}_c(p)$. This gives us soundness of $f_c(p)$:

$$\forall S \in \mathit{Store}, u \in U : S, u \models f_c(p) \Rightarrow S, u \models \mathsf{wlp}_c(p).$$

$\square$

Proposition 4.4.6. *Let p be a* $\mathsf{GMNP}[n]$ *predicate and let* Γ *be a closed* $\mathsf{FO}^{\mathsf{TC}}$ *formula. The evaluation of p under* true *and* Γ *results in an under-approximation of p, i.e.*

$$\mathsf{eval}\ \mathsf{true}\ \Gamma\ p \models_\Gamma p.$$

Proof. We show the following, more general statement:

$$\begin{aligned} &\mathsf{eval}\ \mathsf{true}\ \Gamma\ p \models_\Gamma p \\ \text{and}\quad &p \models_\Gamma \mathsf{eval}\ \mathsf{false}\ \Gamma\ p \end{aligned}$$

by structural induction on p. We only consider the two interesting cases for negation and universal constraints. All other cases follow from monotonicity of the appropriate operators with respect to $\models_\Gamma$.

Let $p = \neg p'$. By definition of eval we have:

$$\mathsf{eval}\ t\ \Gamma\ p = \neg(\mathsf{eval}\ (\neg t)\ \Gamma\ p')$$

If $t = \mathsf{true}$ then we have by induction hypothesis:

$$p' \models_\Gamma \mathsf{eval}\ \mathsf{false}\ \Gamma\ p'$$

and thus we get:

$$\mathsf{eval}\ \mathsf{true}\ \Gamma\ p = \neg(\mathsf{eval}\ \mathsf{false}\ \Gamma\ p') \models_\Gamma \neg p' = p.$$

If on the other hand $t = \mathsf{false}$ then we dually have again by induction hypothesis:

$$\mathsf{eval}\ \mathsf{true}\ \Gamma\ p' \models_\Gamma p'$$

and thus we get:

$$p = \neg p' \models_\Gamma \neg(\mathsf{eval}\ \mathsf{true}\ \Gamma\ p') = \mathsf{eval}\ \mathsf{false}\ \Gamma\ p.$$

Let $p = \forall v.x(v) \to p'(v)$. By definition of eval we have:

$$\mathsf{eval}\ t\ \Gamma\ (\forall v'.x(v') \to p'(v')) = \begin{cases} t, & \text{if } \Gamma \models \forall v.x(v) \to (t \leftrightarrow \mathsf{eval}\ t\ \Gamma\ p') \\ \neg t, & \text{otherwise} \end{cases}$$

Consider the first case where $t = \mathsf{true}$. If

$$\Gamma \models \forall v.x(v) \to \mathsf{eval}\ \mathsf{true}\ \Gamma\ p'$$

then by application of the induction hypothesis we get:

$$\forall S \in \mathit{Store} : S \models \Gamma \Rightarrow S \models \forall v.x(v) \to p'$$

From this we can conclude:

$$\mathsf{eval\ true}\ \Gamma\ p = \mathsf{true} \models_\Gamma p.$$

If on the other hand:

$$\Gamma \not\models \forall v.x(v) \to \mathsf{eval\ true}\ \Gamma\ p'$$

then we trivially get:

$$\mathsf{eval\ true}\ \Gamma\ p = \mathsf{false} \models_\Gamma p.$$

Now consider the second case where $t = \mathsf{false}$. If

$$\Gamma \models \forall v.x(v) \to \neg(\mathsf{eval\ false}\ \Gamma\ p')$$

then by application of the induction hypothesis we get:

$$\forall S \in \mathit{Store} :\ S \models \Gamma \Rightarrow S \models \forall v.x(v) \to \neg p'$$

From this we can conclude:

$$\exists\, v.x(v) \wedge \mathsf{eval\ false}\ \Gamma\ p \models_\Gamma \mathsf{false}.$$

Making use of Lemma 4.4.3 we get:

$$p = \forall v.x(v) \to \mathsf{eval\ false}\ \Gamma\ p \models_\Gamma \mathsf{false} = \mathsf{eval\ false}\ \Gamma\ p.$$

If on the other hand:

$$\Gamma \not\models \forall v.x(v) \to \neg(\mathsf{eval\ true}\ \Gamma\ p')$$

then we trivially get:

$$p \models_\Gamma \mathsf{true} = \mathsf{eval\ false}\ \Gamma\ p.$$

□

Lemma 4.4.7. *Let S be some store and let Pred be the finite set of abstraction node predicates. For any program variable* x *with $x(v) \in \mathit{Pred}$ and formula $\varphi(v) \in \mathcal{F}_{\mathit{Pred}}$:*

$$\begin{aligned} &\textit{either } \alpha(S) \models \forall v.x(v) \to \varphi(v) \\ &\quad\textit{or } \alpha(S) \models \forall v.x(v) \to \neg\varphi(v). \end{aligned}$$

Proof. Let $\alpha(S) = \forall v. \bigvee_i P_i(v)$ where for all $i : P_i(v)$ is a monomial over Pred. Since $x(v)$ is in Pred and since S is a store, there is exactly one monomial $P(v)$ among the P_i such that $P(v) \models x(v)$ and $P(v)$ is satisfiable. Since $P(v)$ is a monomial over Pred and $\varphi(v) \in \mathcal{F}_{\mathit{Pred}}$, we have:

$$\begin{aligned} &\text{either } P(v) \models \varphi(v) \\ &\quad\text{or } P(v) \models \neg\varphi(v). \end{aligned}$$

Since P is unique, we have:

$$\begin{aligned} &\text{either } \bigvee_i P_i(v) \models x(v) \to \varphi(v) \\ &\quad\text{or } \bigvee_i P_i(v) \models x(v) \to \neg\varphi(v). \end{aligned}$$

From this we can conclude:

$$\begin{aligned} &\text{either } \forall v. \bigvee_i P_i(v) \models \forall v.x(v) \to \varphi(v) \\ &\quad\text{or } \forall v. \bigvee_i P_i(v) \models \forall v.x(v) \to \neg\varphi(v). \end{aligned}$$

□

Lemma 4.4.9. *Let p be a* $\mathsf{GMNP}[n]$ *predicate. The closure of p contains all possible modal node predicates that may occur in the image of* eval*, i.e. for every closed* $\mathsf{FO}^{\mathsf{TC}}$ *formula Γ and truth value t we have:*

$$\mathsf{eval}\ t\ \Gamma\ p \in \mathsf{cl}_1(p) \textit{ and } \mathsf{eval}\ t\ \Gamma\ p \in \mathsf{cl}(p).$$

Proof. The function eval replaces every quantified sub-formula occuring in p with either true or false. So does cl_1. A simple induction proof on the structure of p shows:

$$\mathsf{eval}\ t\ \Gamma\ p \in \mathsf{cl}_1(p)$$

By definition of cl we have $\mathsf{cl}_1(p) \subseteq \mathsf{cl}(p)$ and get:

$$\mathsf{eval}\ t\ \Gamma\ p \in \mathsf{cl}(p).$$

□

Lemma 4.4.10. *Let p be a* $\mathsf{GMNP}[n]$ *predicate and let*

$$\forall v.x(v) \rightarrow q(v)$$

be some sub-formula of p then $x(v) \in \mathsf{cl}(p)$ and for every closed $\mathsf{FO}^{\mathsf{TC}}$ *formula Γ and truth value t:*

$$\mathsf{eval}\ t\ \Gamma\ q \in \mathsf{cl}(p).$$

Proof. From the definition of cl_2 and Lemma 4.4.9 it follows immediately that:

$$x(v) \in \mathsf{cl}_2(p) \text{ and } \mathsf{eval}\ t\ \Gamma\ q \in \mathsf{cl}_2(p).$$

Hence, we have:

$$x(v) \in \mathsf{cl}(p) \text{ and } \mathsf{eval}\ t\ \Gamma\ q \in \mathsf{cl}(p).$$

□

Proposition 4.4.11. *Let p be a* $\mathsf{GMNP}[n]$ *predicate. If* $\mathsf{cl}(p) \subseteq \mathit{Pred}$ *then:*

$$\forall S \in \mathit{Store} :\ p \models\!\!\!\dashv_{\alpha(S)} \mathsf{eval\ true}\ (\alpha\ S)\ p.$$

Proof. The right-to-left direction follows already from Proposition 4.4.6. For the left-to-right direction we prove the following more general statement:

$$\begin{aligned} &\forall S \in \mathit{Store} :\ p \models_{\alpha(S)} \mathsf{eval\ true}\ (\alpha\ S)\ p \\ \text{and } &\forall S \in \mathit{Store} :\ \mathsf{eval\ false}\ (\alpha\ S)\ p \models_{\alpha(S)} p. \end{aligned}$$

The proof goes by structural induction on p. We only consider the interesting case where p is a guarded universal constraint. All other cases follow by simple application of the induction hypothesis. So, let p be a guarded universal constraint with:

$$p = \forall v.x(v) \rightarrow p'(v).$$

If t is true then by definition of eval we have:

$$\mathsf{eval\ true}\ (\alpha\ S)\ p = \begin{cases} \mathsf{true}, & \text{if } \alpha(S) \models \forall v.x(v) \rightarrow \mathsf{eval\ true}\ (\alpha\ S)\ p' \\ \mathsf{false}, & \text{otherwise} \end{cases}$$

If $\alpha(S) \models \forall v.x(v) \rightarrow \mathsf{eval\ true}\ (\alpha\ S)\ p'$ holds then we trivially have:

$$p \models_{\alpha(S)} \mathsf{eval\ true}\ (\alpha\ S)\ p.$$

Assume on the other hand that the entailment does not hold, i.e.

$$\alpha(S) \not\models \forall v.x(v) \to \textsf{eval true}\ (\alpha\ S)\ p'.$$

By Lemma 4.4.10 we know that both $x(v)$ and $\textsf{eval true}\ (\alpha\ S)\ p'$ are in $\textsf{cl}(p)$. Since $\textsf{cl}(p)$ is a subset of *Pred*, we can conclude by Lemma 4.4.7:

$$\begin{aligned}
&\alpha(S) \models \forall v.x(v) \to \neg(\textsf{eval true}\ (\alpha\ S)\ p') \\
\Longrightarrow\ &\forall S' \in \mathit{Store} :\ S' \models \alpha(S) \Rightarrow S' \models \forall v.x(v) \to \neg p' && \text{(by Proposition 4.4.6)} \\
\Longleftrightarrow\ &\forall S' \in \mathit{Store} :\ S' \models \alpha(S) \Rightarrow S' \models \neg\exists\, v.x(v) \wedge p' \\
\Longleftrightarrow\ &\forall S' \in \mathit{Store} :\ S' \models \alpha(S) \Rightarrow S' \models \neg\forall v.x(v) \to p' && \text{(by Lemma 4.4.7)} \\
\Longleftrightarrow\ &\forall S' \in \mathit{Store} :\ S' \models \alpha(S) \Rightarrow S' \models \neg p
\end{aligned}$$

From this we can finally conclude:

$$p \models_{\alpha(S)} \textsf{eval true}\ (\alpha\ S)\ p.$$

The case where t is $\textsf{false}$ is analogous. □

Proposition 4.4.13. *For commands c of the form* `x->n = y` *and any* $\textsf{MNP}[n]$ *predicate p there is a finite subset Pred of* $\textsf{MNP}[n]$ *predicates such that:*

$$\forall S \in \mathit{Store} :\ \textsf{wlp}^{\#}_{c,\alpha(S)}(p) \models\!\!\!\!\dashv_{\alpha(S)} \textsf{wlp}_c(p).$$

Proof. The set $\textsf{cl}(p)$ is finite. Hence, we can choose $\mathit{Pred} \overset{def}{=} \textsf{cl}(p)$. From Corollary 4.4.12 we know:

$$\forall S \in \mathit{Store} :\ \textsf{wlp}_c(p) \models\!\!\!\!\dashv_{\alpha(S)} \textsf{eval true}\ (\alpha\ S)\ f\ p.$$

By Lemma 4.4.9 and the definition of *Pred* we know:

$$\textsf{eval true}\ (\alpha\ S)\ f\ p \in \mathit{Pred}.$$

By definition, $\textsf{wlp}^{\#}_{c,\alpha(S)}(p)$ is the best under-approximation of $\textsf{wlp}_c(p)$ with respect to $\models_{\alpha(S)}$ and *Pred*. Hence, we must have:

$$\forall S \in \mathit{Store} :\ \textsf{wlp}^{\#}_{c,\alpha(S)}(p) \models\!\!\!\!\dashv_{\alpha(S)} \textsf{eval true}\ (\alpha\ S)\ f\ p.$$

from which we can finally conclude:

$$\forall S \in \mathit{Store} :\ \textsf{wlp}^{\#}_{c,\alpha(S)}(p) \models\!\!\!\!\dashv_{\alpha(S)} \textsf{wlp}_c(p).$$

□

Appendix B

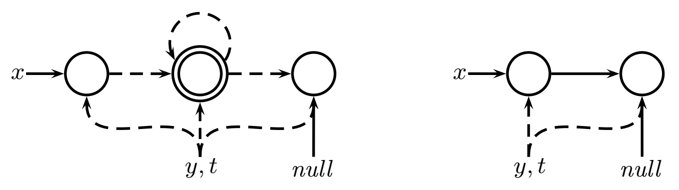
Least Fixed Point for Program Reverse

The following table shows the iteration of the abstract post operator $\mathsf{post}^{\#}$ on the initial abstract state $\mathsf{init}^{\#}$ according to the definitions given in Section 5.5. We use the representation of abstract stores introduced in that section. For each of the images under $\mathsf{post}^{\#}$ we only list those components which are different from false.

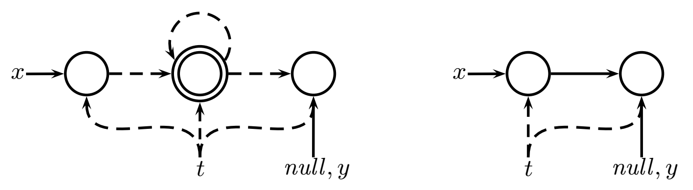
The least fixed point of $\mathsf{post}^{\#}$ under $\mathsf{init}^{\#}$ is given by a tuple over $AbsDom[Pred]$ whose components correspond to the disjunction of all abstract stores that are listed for the corresponding program locations in any of the iteration steps.

The least fixed point is reached after 22 iterations of $\mathsf{post}^{\#}$. For program location `l6` it is already reached after 17 iterations. Since we are mainly interested in the abstract stores emerging at program location `l6` we show the computation of $\mathsf{lfp}(\mathsf{post}^{\#}(\mathsf{init}^{\#}))$ only up to the 17-th iteration.

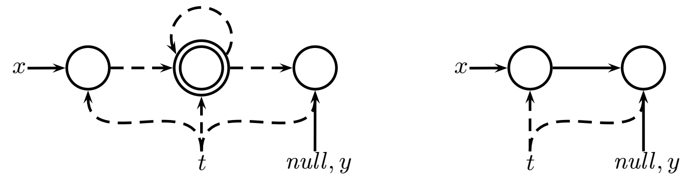
$\mathsf{post}^{\#0}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
`l0` :	1	$*$	$*$	0	1	$*$	$*$	0	1	$*$
	0	$*$	$*$	0	1	$*$	$*$	1	1	$*$
	0	$*$	$*$	1	1	$*$	1	1	1	$*$
										

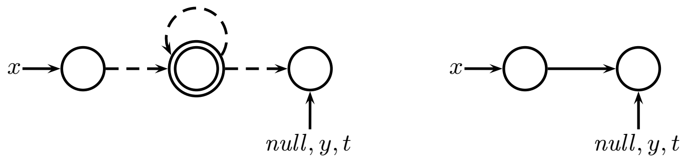
$\mathsf{post}^{\#1}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
`l1` :	1	0	$*$	0	1	1	1	0	1	$*$
	0	0	$*$	0	1	1	1	1	1	$*$
	0	1	$*$	1	1	1	1	1	1	$*$
										

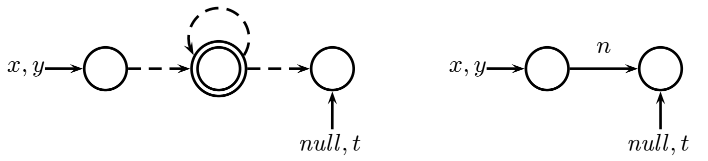
$\mathsf{post}^{\#2}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
12 :	1	0	$*$	0	1	1	1	0	1	$*$
	0	0	$*$	0	1	1	1	1	1	$*$
	0	1	$*$	1	1	1	1	1	1	$*$
										

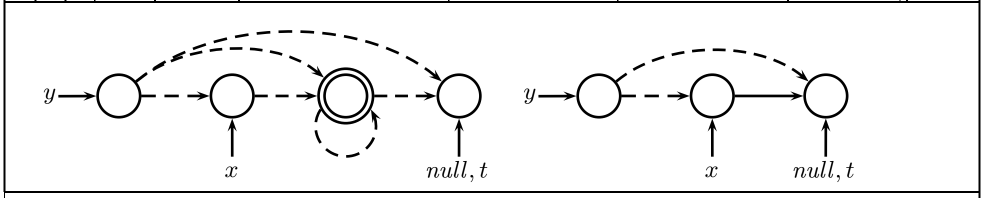
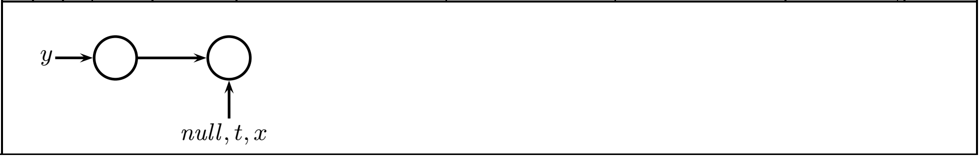
$\mathsf{post}^{\#3}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
13 :	1	0	0	0	1	1	1	0	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	0	1	1	1	1	1	1	1	1	$*$
										

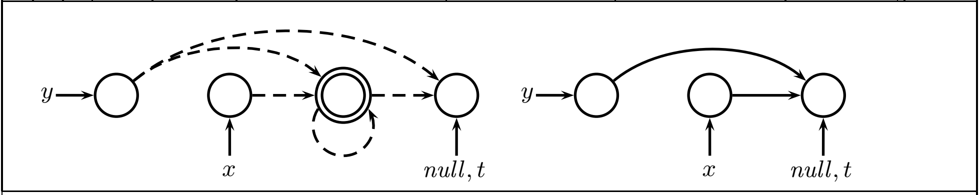
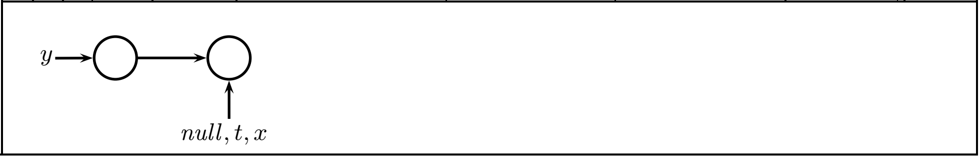
$\mathsf{post}^{\#4}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
14 :	1	1	0	0	1	0	0	0	1	0
	0	0	0	0	1	0	1	1	1	$*$
	0	0	1	1	1	0	1	1	1	$*$
										

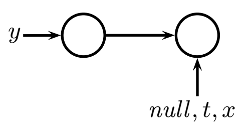
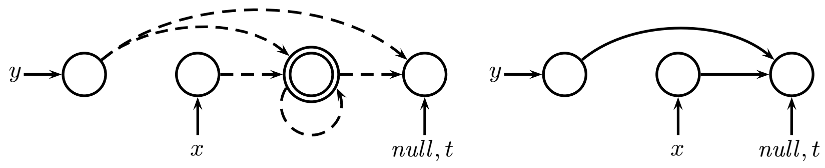
$\mathsf{post}^{\#5}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x\wedge\neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
l5 :	0	1	0	0	0	0	0	0	1	0
	1	0	0	0	1	1	1	0	1	0
	0	0	0	0	1	1	1	1	1	$*$
	0	0	1	1	1	1	1	1	1	$*$
										
	0	1	0	0	0	0	0	0	1	0
	0	0	0	0	1	1	1	1	1	$*$
	1	0	1	1	1	1	1	1	1	0
										

$\mathsf{post}^{\#6}(\mathsf{init}^{\#})$

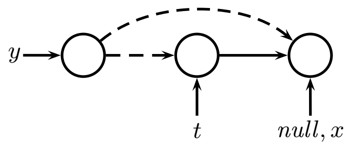
	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x\wedge\neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
l1 :	0	1	0	0	0	0	0	1	1	$*$
	1	0	0	0	1	1	1	0	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	0	0	1	1	1	1	1	1	1	$*$
										
	0	1	0	0	0	0	0	1	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	1	0	1	1	1	1	1	1	1	$*$
										

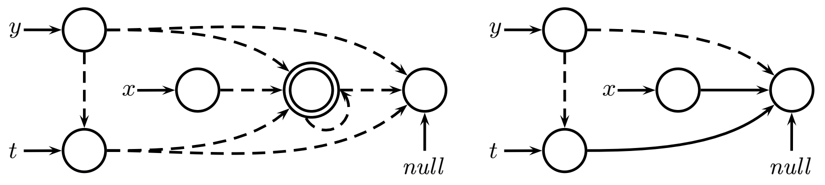
$\mathsf{post}^{\#7}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
	0	1	0	0	0	0	0	1	1	$*$
16 :	0	0	0	0	1	1	1	1	1	$*$
	1	0	1	1	1	1	1	1	1	$*$
										
	0	1	0	0	0	0	0	1	1	$*$
12 :	1	0	0	0	1	1	1	0	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	0	0	1	1	1	1	1	1	1	$*$
										

⋮

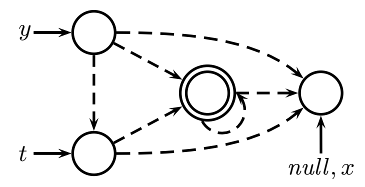
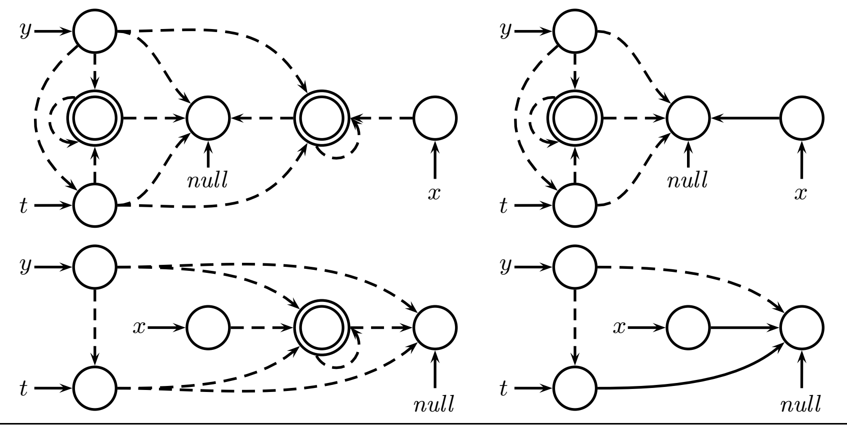
$\mathsf{post}^{\#12}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
	0	0	1	0	0	0	1	1	1	$*$
16 :	0	1	0	0	0	0	0	1	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	1	0	1	1	1	1	1	1	1	$*$
										

	x	y	t	$null$	$x.\langle n^*\rangle$	$(\neg y).\mathsf{S}_{\langle n\rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n\rangle}.null$	$(\neg x).\mathsf{U}_{\langle n\rangle}.null$	$\langle n^*\rangle.null$	$x.\langle n\rangle$
	0	0	1	0	0	0	1	1	1	$*$
12 :	0	1	0	0	0	0	0	1	1	$*$
	1	0	0	0	1	1	1	0	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	0	0	0	1	1	1	1	1	1	$*$
										

⋮

$\mathsf{post}^{\#17}(\mathsf{init}^{\#})$

	x	y	t	$null$	$x.\langle n^* \rangle$	$(\neg y).\mathsf{S}_{\langle n \rangle}.(x \wedge \neg y)$	$(\neg y).\mathsf{U}_{\langle n \rangle}.null$	$(\neg x).\mathsf{U}_{\langle n \rangle}.null$	$\langle n^* \rangle.null$	$x.\langle n \rangle$
	0	0	0	0	0	0	1	1	1	$*$
16 :	0	0	1	0	0	0	1	1	1	$*$
	0	1	0	0	0	0	0	1	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	1	0	1	1	1	1	1	1	1	$*$
										
	0	0	0	0	0	0	1	1	1	$*$
12 :	0	0	1	0	0	0	1	1	1	$*$
	0	1	0	0	0	0	0	1	1	$*$
	1	0	0	0	1	1	1	0	1	$*$
	0	0	0	0	1	1	1	1	1	$*$
	0	0	0	1	1	1	1	1	1	$*$
	 y t $null$ x y t $null$ x y x t $null$ y x t $null$									