Albert Ludwigs Universität Freiburg

Department of Computer Science

Decision Procedures for List Manipulating Programs

Master Thesis

from

Marco Muñiz

Advised by: Prof. Dr. Andreas Podelski
Dipl. Inf. Thomas Wies

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den 24. März 2009

Acknowledgments

I would like to thank my family whose support made this work possible. I thank Thomas Wies for giving me an interesting topic, constant guidance, active support and for his patience. I thank Prof. Dr. Andreas Podelski for his guidance and all the given enriching opportunities. I would like to thank Dr. Jochen Hoenicke for his accurate observations and in particular to Prof. Dr. Viktor Kuncak for allowing me to develop this thesis in the context of the Jahob system.

Abstract

Software systems are ubiquitous in modern technology. Therefore, reliable software is desirable. One way of ensuring software reliability is formal verification. A class of programs that is particularly challenging for formal verification techniques are programs that dynamically allocate and manipulate linked data structures. We call such programs heap-manipulating programs. The correctness of heap-manipulating programs often depends on the consistency of the linked data structures. Data structure consistency involves properties such as: a given linked list is acyclic, elements in a list are sorted and two references point to disjoint regions in the heap. An expressive, yet, decidable logic for reasoning about heap-manipulating programs is Lahiri's and Qadeer's *Logic of Interpreted Sets and Bounded Quantification* (LISBQ). This logic allows one to express complex properties of list-like data structures.

In this thesis, we explore the usefulness of LISBQ in the context of the Jahob verification system for the verification of heap-manipulating programs. Toward this goal, we give an embedding of LISBQ into Higher-Order Logic. Then, we describe a translation of the HOL fragment corresponding to LISBQ into First-Order Logic. Finally, we describe an implementation of the decision procedure for LISBQ in the Jahob verification system.

We used our implementation to verify properties of operations on various list-like data structures including singly-liked lists, sorted lists, cyclic lists and association lists. Our evaluation showed, that the use of LISBQ helps in both, increasing the scope of verifiable data structures and reducing substantially the annotations on data structure implementations that have been previously verified in Jahob. In addition, we used LISBQ in combination with other specialized decision procedures to verify properties of operations on threaded trees with integer keys. The verified properties are beyond the scope of any individual decision procedure that is integrated in the Jahob verification system, including LISBQ itself.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are ubiquitous in modern technology. Therefore, reliable software is desirable. One way of ensuring software reliability is formal verification. A class of programs that is particularly challenging for formal verification techniques are programs that dynamically allocate and manipulate linked data structures. We call such programs heap-manipulating programs. The correctness of heap-manipulating programs often depends on the consistency of the linked data structures. Data structure consistency involves properties such as: a given linked list is acyclic, elements in a list are sorted and two references point to disjoint regions in the heap. An expressive, yet, decidable logic for reasoning about heap-manipulating programs is Lahiri's and Qadeer's *Logic of Interpreted Sets and Bounded Quantification* (LISBQ). This logic allows one to express complex properties of list-like data structures.

Properties of data structures such as acyclicity can be expressed in terms of reachability in the graph that is spanned by the program's memory. A common way to reason about such properties is by using a logic that provides a transitive closure operator [17]. There exist many approaches that use transitive closure logics, some of them include: the Pointer Assertion Logic Engine (PALE) [25], Three-valued Shape Analysis [30] and Symbolic Shape Analysis [34]. Unfortunately, the satisfiability problem for these logics tends to be either undecidable [17] or have very high complexity.

A noticeable exception from this rule is the *Logic of Interpreted Sets and Bounded Quantification* (LISBQ) [23]. This logic provides a specialized transitive closure operator that is restricted to functional relations. This operator enables reasoning about reachability in list-like data structures. LISBQ was specifically designed for program verification and has many useful properties. In particular, this logic is decidable. In fact, the satisfiability problem for LISBQ has shown to be NP-complete which makes it possible to implement a practically useful decision procedure for LISBQ on top of DPLL-based SMT solvers.

In this thesis, we explore the usefulness of LISBQ for verifying heap-manipulating programs in the Jahob Verification System [20]. The contributions of this thesis are summarized as follows:

- We present an embedding of LISBQ formulas into Higher-Order Logic (HOL) which is the underlying logic of the Jahob Verification System.

- We present a translation of proof obligations which are expressed in Jahob's Higher-Order Logic into the HOL fragment of LISBQ. Then we present a translation from the HOL fragment of LISBQ to First Order Logic.

- We implemented a decision procedure for LISBQ in the Jahob system that is built on top of standard SMT-solvers and first-order theorem provers.

We used our implementation to verify properties of operations on various list-like data structures including singly-liked lists, sorted lists, cyclic lists and association lists. Our evaluation showed, that the use of LISBQ helps in both, increasing the scope of verifiable data structures and reducing substantially the annotations on data structure implementations that have been previously verified in Jahob. In addition, we used LISBQ in combination with other specialized decision procedures to verify properties of operations on threaded trees with integer keys. The verified properties are beyond the scope of any individual decision procedure that is integrated in the Jahob verification system, including LISBQ itself.

1.1 Related Work

In the following, we put our work on LISBQ in the context of another approach for reasoning about heap-manipulating programs in the Jahob Verification System.

Field Constraint Analysis. In order to verify data structure consistency and in particular reachability properties, the Jahob verification system relies on a technique called Field Constraint Analysis [34, Chapter 5]. Field Constraint Analysis is implemented on the context of the Monadic Second-Order Logic (MSOL) over trees [19]. This technique enables the verification of a wide range of data structures. However, it has some limitations, in particular since its implementation uses MSOL over trees, the verifiable data structures are restricted to have a tree like shape. i.e. no cycles nor sharing are allowed. In addition, the complexity of deciding a formula in MSOL is Non-elementary, and the combination of MSOL with other theories is difficult.

The *Logic of Interpreted Sets and Bounded Quantification* (LISBQ) in contrast to MSOL, is complete for reasoning about reachability in lists, i.e. cycles and sharing are allowed. Important characteristics of LISBQ are that it is closed under the computation of weakest preconditions and that it is stably infinite, allowing Nelson-Oppen combination with other theories (e.g. linear integer arithmetic). In addition, the expressiveness of LISBQ enables the specification of rich data structure invariants in a natural way. Despite the expressiveness of LISBQ, the decision problem is NP-complete.

The Jahob Verification System, while verifying a data structure, may use several different specialized decision procedures, i.e. some verification conditions can be discharged by MSOL and some others by LISBQ. Therefore, the use of LISBQ in Jahob will not only simplify the verification process and increase the range verifiable data structures, but also enabling the verification of complex data structures that where beyond the scope of LISBQ or MSOL independently.

1.2 Outline

The outline of this thesis is as follows. In Chapter 2 we provide the foundations for our work. Section 2.1 describes a the subset of Higher-Order Logic that is used to express proof obligations in Jahob. In Section 2.3, we describe the *Logic of Interpreted Sets and Bounded Quantification* and present its corresponding decision procedure. In Chapter 3 we describe how to embed formulas in LISBQ into HOL. The corresponding LISBQ fragment of HOL is called HOL LISBQ. We then describe how to translate proof obligations in Jahob's HOL subset to formulas in HOL LISBQ formulas and from HOL LISBQ to first-order logic. Finally, we show how the decision procedure for LISBQ is implemented on top Jahob's existing back ends to SMT-solvers using the SMT-LIB interface and resolution-based first-order theorem provers using the TPTP interface. In Chapter 4, we present the evaluation of our implementation. Chapter 5 concludes the thesis.

1.3 Verifying List-Manipulating Programs in Jahob

In this section, we provide an example of the verification of a Singly-Linked List. Our intention is that of illustrating Jahob's specification constructs and the use of LISBQ, a detailed description of Jahob's specification language can be found in [20, Chapter 3]. Figure 1.1 shows an annotated Java program. The program implements a Single-Linked List by means of two classes. The class Node has field next for referencing to the next node in the list, and a boolean data field. The class List maintains a reference to the first node of the list with the reference first. Jahob's specifications are introduced as Java comments. The specification variable [14] content is defined as the set of objects different than null which are reachable from the reference first by following the next field. The construct $(\mathsf{first}, x) \in \{(v, w).v..\mathsf{next} = w\}^*$ is used to specify all the elements x such that (first, x) is in the reflexive transitive closure of the next field (In Jahob a field is represented as a total function). The specification variable reach, is a shorthand for the binary predicate that denotes the reflexive transitive closure of next. The class invariant acyclic asserts that the list to be acyclic before and after procedure calls.

The method contains, evaluates if a given node is a member of the list. It takes as argument a node n and by starting at first, it traverses the list maintaining a reference e until it finds the node n or until it reaches the node null, then it returns the boolean value obtained by comparing e with null. This value will be also assigned to the specification variable result. The keywords requires and ensures are used to specify the preconditions and postconditions of a procedure. In this case the method has no precondition and the postcondition ensures, the result to be equal to the value obtained by evaluating the membership of n to the set content. The method is annotated whit a simple loop invariant which specifies that the reference e is always reachable from first and that if there is a path from first to n then there is also a path from e to n.

```
public final class Node {
    public /*: claimedby List */ Node next;
    public /*: claimedby List */ boolean data;
}

public class List
{
    private static Node first;
    /*:
     public static specvar content :: objset;
       vardefs "content == {x. x ≠ null ∧ (first,x) ∈ {(v,w). v..next=w }* }";

     private static specvar reach :: "obj => obj => bool";
       vardefs "reach == ((λ a b. (a,b) ∈ {(v,w). v..next=w }*) a b) ";

     invariant "comment ''acyclic'' (( first , null ) ∈ {(x,y). x..next = y}*)";
    */

    public static boolean contains(Node n)
    /*: ensures " result = ( n ∈ content )" */
    {
        Node e = first;
        while /*: inv " (reach first e) ∧
                        (reach first n → reach e n) "; */
          (e != null && e != n) {
          e = e.next;
      }
      return (e != null);
   }
}
```

Figure 1.1: List class definitions and contains method

Chapter 2

Preliminaries

2.1 Higher-Order Logic

In this section we describe the Jahob's formula language for proof obligations and program annotations, the language is a fragment of classic higher-order logic [1] which we call HOL, a full description of Jahob's HOL can be found in [20, chapter 4].
The motivation for using HOL in Jahob is that HOL is a expressive foundation for reasoning about algebra, sets and relations. This concepts are important because they allow to naturally describe data structure content and enable modular analysis in Jahob by separating verification of data structure uses from verification of data structure implementation.

Formulas Figure 2.1 characterizes the core syntax of HOL. The core of HOL is simply typed lambda calculus [5]. The term $\lambda x :: t.f$ denotes a function h such that $h(x) = f$ for all x. where f typically contains x as a free variable. The equality function takes two arguments and returns true if and only if the arguments are equal. In lambda calculus, expressions are usually called terms, but because of their use in Jahob we call them formulas. However, we keep in mind that in addition to truth values, formulas can denote any other type.

$f ::=$	$\lambda x :: t.f$	lambda abstraction
$\mid$	$f_1 f_2$	function application
$\mid$	$=$	equality
$\mid$	x	variable or constant
$\mid$	$f :: t$	typed formula

Figure 2.1: Core syntax of HOL

Types In Jahob's formula annotation language, the developer does not need to explicitly give the type of a formula. However, Jahob's type inference engine expects that after performing simple transformations such as beta reductions, every formula can be inferred to have ground types. Figure 2.2 gives the ground types of Jahob's HOL; the type bool

represents the truth values, int represents mathematical integers, obj represents objects in the heap, t set denotes sets of elements of type t and $t_1 * t_2$ denotes the type whose interpretation is the Cartesian product of the interpretations $[\![t_1]\!]$ and $[\![t_2]\!]$.

$t ::=$	bool	truth values
$\mid$	int	integers
$\mid$	obj	uninterpreted objects
$\mid$	$'\alpha$	type variable
$\mid$	$t_1 \Rightarrow t_2$	total functions
$\mid$	t set	sets
$\mid$	$t_1 * t_2$	pairs

Figure 2.2: Types of HOL

Semantics Figure 2.3 presents the standard set theoretic semantics of HOL, which interpret a function of type $t_1 \Rightarrow t_2$ as the set of all total functions from $[\![t_1]\!]$ to $[\![t_2]\!]$. The semantics represent functions as functional relations, which is, as particular sets of pairs. The sets $\{true, false\}, \mathbb{Z}$ and $\mathbb{O}$ are pairwise disjoint. The finite set $\mathbb{O}$ has N objects.

$$
\begin{array}{rcl}
[\![\lambda x :: t.f]\!]e & = & \{(v, [\![f]\!](e[x := v])) \mid v \in [\![t]\!]\} \\
[\![f_1 f_2]\!]e & = & ([\![f_1]\!]e)([\![f_2]\!]e) \\
[\![=:: t \Rightarrow t \Rightarrow \mathsf{bool}]\!] & = & \{f \mid \forall u, v \in [\![t]\!].(fu)v = true \leftrightarrow (u = v)\} \\
[\![x]\!]e & = & ex \\
[\![f :: t]\!]e & = & [\![f]\!]e \\
[\![\mathsf{bool}]\!] & = & \{true, false\} \\
[\![\mathsf{int}]\!] & = & \mathbb{Z} \\
[\![\mathsf{obj}]\!] & = & \mathbb{O} = \{o_1, o_2, \ldots, o_N\} \\
[\![t_1 \Rightarrow t_2]\!] & = & \{f \mid f \subseteq [\![t_1]\!] \times [\![t_2]\!] \wedge \forall x \in [\![t_1]\!].\exists_{1y} \in [\![t_2]\!].(x, y) \in f\} \\
[\![t \ \mathsf{set}]\!] & = & \{f \mid f \subseteq [\![t]\!]\} \\
[\![t_1 * t_2]\!] & = & \{(x_1, x_2) \mid x_1 \in [\![t_1]\!] \wedge x_2 \in [\![t_2]\!]\}
\end{array}
$$

Figure 2.3: Semantics of HOL formulas and ground types

Constants and shorthands Table 2.1 shows the most common constant symbols used in Jahob's formulas, i.e., logical connectives, quantifiers, set operators and arithmetic operators. The constant rtrancl_pt is used to express the reflexive transitive closure of a binary predicate, this transitive-closure operator is used to express reachability properties about the heap of a program. As an example consider a null terminated list linked through a field *next* with a reference first to the first element of the list. Then the rtrancl_pt operator can be used to express the fact that the list is acyclic, e.g.

$$\mathsf{rtrancl\_pt}\ (\lambda xy.next\ x = y)\ first\ \mathsf{null}$$

The rtrancl_pt operator is also useful for constructing sets, such as the set of nodes reachable from *first*. The constant update will map the function f to a new function in which the value of x is equal to v.

constant	type	semantics	notation
$\neg$	bool $\Rightarrow$ bool	negation	prefix
$\wedge, \vee, \rightarrow$	bool $\Rightarrow$ bool $\Rightarrow$ bool	and, or, implication	infix
$\forall$	$('\alpha \Rightarrow '\beta) \Rightarrow$ bool	$[\![\forall]\!]h = \forall v.hv$	$\forall x.f$ denotes $\forall(\lambda x.f)$
$\exists$	$('\alpha \Rightarrow '\beta) \Rightarrow$ bool	$[\![\exists]\!]h = \exists v.hv$	$\exists x.f$ denotes $\exists(\lambda x.f)$
Collect	$('\alpha \Rightarrow$ bool$) \Rightarrow '\alpha$ set	Collect $h = \{v \mid hv\}$	$\{x.f\}$
$\cup, \cap, \setminus$	$'\alpha$ set $\Rightarrow '\alpha$ set $\Rightarrow '\alpha$ set	union,intersection difference	
$\subseteq$	$'\alpha$ set $\Rightarrow '\alpha$ set $\Rightarrow$ bool	subset	
$<, \leq$	int $\Rightarrow$ int $\Rightarrow$ bool	ordering	$x > y$ denotes $y < x$
$+, -$	int $\Rightarrow$ int $\Rightarrow$ int	plus,minus	infix
comment	$'\alpha \Rightarrow$ boolbool	comment "text" f=f	
rtrancl_pt	$('\alpha \Rightarrow '\alpha \Rightarrow$ bool$) \Rightarrow$ $'\alpha \Rightarrow '\alpha \Rightarrow$ bool	rtrancl_pt $p\ z_0 y = (z_0 = y)$ $\vee(\exists z_1, \ldots, z_n.z_n = y \wedge$ $\bigwedge_{i=0}^{n-1}(p\ z_i z_{i+1}))$	$(a,b) \in \{(x,y).f\}^*$ denotes rtrancl_pt$(\lambda xy.f)ab$
update	$('\alpha \Rightarrow '\beta) \Rightarrow '\alpha \Rightarrow$ $'\beta \Rightarrow '\alpha \Rightarrow '\beta$	$f(x := v)x = v$ $f(x := v)y = fy$, for $y \neq x$	$f(x := v)$ denotes update $f\ x\ y$

Table 2.1: Logical constants and shorthands

2.2 Programs

The input of Jahob are programs written in an expressive subset of Java, these programs may contain annotations specifiying procedure contracts and loop invariants, these loop invariants will be used to unroll while loops. Figure 2.4 shows the syntax of Jahob's loop free guarded command language; the statement assert$F$ is used to introduce preconditions and conditional statements, the statement assert$F$ is used to introduce assertions and postconditions, the statement $x := F$ evaluates F and assigns it to x, the statement c_1, c_2 evaluates c_1 and c_2 sequentially, and finally, the statement $c_1 \Box c_2$ evaluates either c_1 or c_2 non-deterministically.

Weakest Preconditions Figure 2.5 shows how to compute the weakest liberal precondition wlp for the guarded command language described in Figure 2.4. The assigment $x = F$ can be desugared as follows:

$$\text{havoc } x; \text{assume}(x = F)$$

$$
\begin{array}{rll}
c ::= & x := F & \text{(assignment statement)} \\
| & \mathsf{havoc}\ x & \text{(non-deterministic assignment to x)} \\
| & \mathsf{assume}\ F & \text{(assume statement)} \\
| & \mathsf{assert}\ F & \text{(assert statement)} \\
| & c_1; c_2 & \text{(sequential composition)} \\
| & c_1 \Box c_2 & \text{(non-deterministic choice)}
\end{array}
$$

Figure 2.4: Loop free guarded command language

$$
\begin{array}{rcl}
\mathsf{wlp}(\mathsf{assert}\ F, G) & \equiv & F \wedge G \\
\mathsf{wlp}(\mathsf{assume}\ F, G) & \equiv & F \rightarrow G \\
\mathsf{wlp}(\mathsf{havoc}\ x, G) & \equiv & \forall x.G \\
\mathsf{wlp}(c_1; c_2, G) & \equiv & \mathsf{wlp}(c_1, \mathsf{wlp}(c_2, G)) \\
\mathsf{wlp}(c_1 \Box c_2, G) & \equiv & \mathsf{wlp}(c_1, G) \wedge \mathsf{wlp}(c_2, G)
\end{array}
$$

Figure 2.5: Weakest liberal precondition

The translation of a Jahob program T into a formula F has the property that F is unsatisfiable iff T does not fail any assertion.

2.3 Logic of Interpreted Sets and Bounded Quantification

In this section we describe the *Logic of Interpreted sets and bounded quantification* as it is presented in [23]. However, we restrict ourselves to the fragment which is relevant to the implementation of LISBQ in the Jahob system.

2.3.1 Logic

The logic presented in Figure 2.6, is interpreted over a finite partially-ordered set $\mathcal{D}$ of sorts, where constant symbols are of sort $\mathit{Integer} \in \mathcal{D}$, variables are of some sort $D \in \mathcal{D}$ and uninterpreted functions f are of sort $D \rightarrow E$ and $D, E \in \mathcal{D}$. A model M for a formula in the logic provides an interpretation M_D for each sort $D \in \mathcal{D}$, where M_{integer} is just the set of integers. In addition, the model also provides an interpretation $M_x \in M_D$ for each variable x of sort D and an interpretation $M_f : M_D \rightarrow M_E$ for each function f of sort $D \rightarrow E$. The interpretation is extended to arbitrary terms in the logic in the natural way.

An important characteristic of the logic is that it provides a ternary reachability predicate $. \xrightarrow{f} . \xrightarrow{f} .$ for every function f of sort $D \rightarrow D$ where $D \in \mathcal{D}$. The intuition of the ternary reachability predicate is depicted in Figure 2.7, the predicate will hold if by starting at t_1 and by applying f 0 or more times, we can reach t_2 and from t_2 we can reach t_3 by applying f 0 or more times. In particular u_i has to be in between u_0 and u_n.

$$
\begin{array}{lcll}
c & \in & \mathit{Integer} & \\
x & \in & \mathit{Variable} & \\
f & \in & \mathit{Function} & \\
\varphi & \in & \mathit{Formula} & ::= \quad \alpha \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \\
\alpha & \in & \forall\mathit{Formula} & ::= \quad \gamma \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid \forall x \in S.\alpha \\
\gamma & \in & \mathit{GFormula} & ::= \quad t_1 = t_2 \mid t_1 < t_2 \mid t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \mid \neg\gamma \\
t & \in & \mathit{Term} & ::= \quad c \mid x \mid t_1 - t_2 \mid t_1 + t_2 \mid f(t) \\
 & & & \quad \mid \mathit{ite}(t = t', t_1, t_2) \\
S & \in & \mathit{Set} & ::= \quad g^{-1}(t) \mid \mathit{Btwn}(f, t_1, t_2)
\end{array}
$$

Figure 2.6: Logic of Interpreted Sets and Bounded Quantification

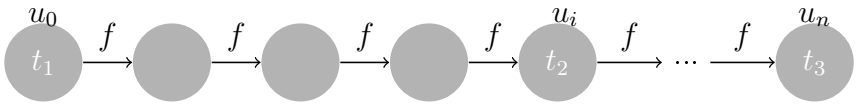

Figure 2.7: Ternary reachability predicate

Formally, the Model M satisfies $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ written $M \models t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ iff. there are distinct $u_0, u_1, u_2, ..., u_n \in M_D$ such that $M_f(u_i) = u_{i+1}$ for all $i \in [0, n)$, $u_0 = M_{t_1}$, $u_n = M_{t_3}$, $u_i = M_{t_2}$ for some $i \in [0, n]$.
Binary reachability can be expressed as

$$t_1 \xrightarrow{f} t_2 \equiv t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_2$$

The logic provides two set constructors. The set constructor $f^{-1}(t)$ consisting of a function f of sort $D \to E$ where $D, E \in \mathcal{D}$ and $E \neq D$. Formally, we have $M \models x \in f^{-1}$ iff $M_{f(x)} = M_t$. The set $\mathit{Btwn}(f, t_1, f_2)$ will collect all the elements in between the elements t_1 and t_2 by applying the function $f : D \to D$. Formally we have $M \models x \in \mathit{Btwn}(f, t_1, t_2)$ iff $M \models t_1 \xrightarrow{f} x \xrightarrow{f} t_3$.
In order to ensure termination of the decision procedure described in Section 2.3.3, a sort-restriction in formulae $\forall x \in S.\alpha$ is required. Specifically, the sort of any term in α containing x as a strict sub-term is less than the sort of x.

2.3.2 Closeness under weakest preconditions

We describe now, how to compute weakest preconditions for a program T in a formula φ in LISBQ. Assuming all formulas in assert and assume statements are in LISBQ, then it is easy to see that LISBQ is closed under the computation of wlp according to Figure 2.5 with exception of field updates $f(p) := q$. In the following, we show that for any formula φ in LISBQ, $\mathsf{wlp}(f(x) := y, \varphi$ is expressible in LISBQ.

The main difficulty is that a local change to a field, can change the truth value of the ternary reachability predicate. In order to illustrate this we refer to Figure 2.8. The first model illustrates the field f. The model below illustrates an update to the field f at position p to q. As an example, the truth value of the predicate $t_1 \xrightarrow{f} t_2$ in the figures will differ.

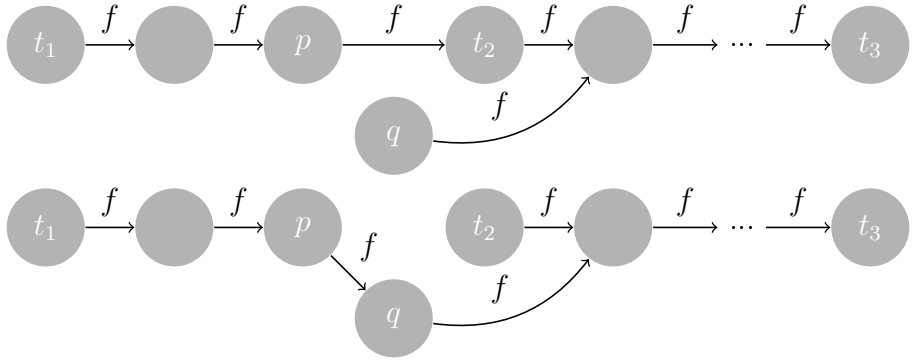

Figure 2.8: Field update $f(p) := q$

Definition of $\Gamma(\varphi, f, p, q)$

$$\Gamma(\varphi_1 \wedge \varphi_2, f, p, q) \equiv \Gamma(\varphi_1, f, p, q) \wedge \Gamma(\varphi_2, f, p, q)$$

$$\Gamma(\varphi_1 \vee \varphi_2, f, p, q) \equiv \Gamma(\varphi_1, f, p, q) \vee \Gamma(\varphi_2, f, p, q)$$

$$\Gamma(\neg\varphi_1, f, p, q) \equiv \neg\Gamma(\varphi_1, f, p, q)$$

Definition of $\Gamma(\alpha, f, p, q)$

$$\Gamma(\alpha_1 \wedge \alpha_2, f, p, q) \equiv \Gamma(\alpha_1, f, p, q) \wedge \Gamma(\alpha_2, f, p, q)$$

$$\Gamma(\alpha_1 \vee \alpha_2, f, p, q) \equiv \Gamma(\alpha_1, f, p, q) \vee \Gamma(\alpha_2, f, p, q)$$

$$\Gamma(\forall x \in g^{-1}(t).\alpha_1, f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t' = \Gamma(t, f, p, q), \alpha' = \Gamma(\alpha, f, p, q)$$
$$\quad in$$
$$\quad\quad \forall x \in g^{-1}(t').\alpha'$$

$$\Gamma(\forall x \in f^{-1}(t).\alpha_1, f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t' = \Gamma(t, f, p, q), \alpha' = \Gamma(\alpha, f, p, q)$$
$$\quad in$$
$$\quad\quad \wedge \quad q = t' \Rightarrow \alpha'[x/p] \wedge \forall x \in f^{-1}(t').\alpha'$$
$$\quad\quad \wedge \quad q \neq t' \Rightarrow \forall x \in f^{-1}(t').x = p \vee q$$

$$\Gamma(\forall x \in Btwn(g, t_1, t_2).\alpha, f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t'_1 = \Gamma(t_1, f, p, q), t'_2 = \Gamma(t_2, f, p, q), \alpha' = \Gamma(\alpha, f, p, q)$$
$$\quad in$$
$$\quad\quad \forall x \in Btwn(g, t'_1, t'_2).\alpha'$$

$$\Gamma(\forall x \in Btwn(f, t_1, t_2).\alpha, f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t'_1 = \Gamma(t_1, f, p, q), t'_2 = \Gamma(t_2, f, p, q), \alpha' = \Gamma(\alpha, f, p, q)$$
$$\quad in$$
$$\quad\quad \wedge \quad t'_1 \xrightarrow[p]{f} t'_2 \Rightarrow \forall x \in Btwn(f, t'_1, t'_2).\alpha'$$
$$\quad\quad \wedge \quad p \neq t'_2 \wedge t'_1 \xrightarrow[t'_2]{f} p \wedge q \xrightarrow[p]{f} t'_2 \Rightarrow \forall x \in Btwn(f, t'_1, t'_2).\alpha'$$
$$\quad\quad \wedge \quad p \neq t'_2 \wedge t'_1 \xrightarrow[t'_2]{f} p \wedge q \xrightarrow[p]{f} t'_2 \Rightarrow \forall x \in Btwn(f, q, t'_2).\alpha'$$

Definition of $\Gamma(\gamma, f, p, q)$

$$\Gamma(\neg\gamma, f, p, q) \equiv \neg\Gamma(\gamma, f, p, q)$$

$$\Gamma(t_1 = t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) = \Gamma(t_2, f, p, q)$$

$$\Gamma(t_1 < t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) < \Gamma(t_2, f, p, q)$$

$$\Gamma(t_1 \xrightarrow{g} t_2 \xrightarrow{g} t_3, f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t'_1 = \Gamma(t_1, f, p, q), t'_2 = \Gamma(t_2, f, p, q), t'_3 = \Gamma(t_3, f, p, q)$$
$$\quad in$$
$$\quad\quad t'_1 \xrightarrow{g} t'_2 \xrightarrow{g} t'_3$$

$$\Gamma(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3, f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t'_1 = \Gamma(t_1, f, p, q), t'_2 = \Gamma(t_2, f, p, q), t'_3 = \Gamma(t_3, f, p, q)$$
$$\quad in$$
$$\quad\quad t'_1 \xrightarrow{f^p_q} t'_2 \xrightarrow{f^p_q} t'_3$$

Definition of $\Gamma(t, f, p, q)$

$$\Gamma(c, f, p, q) \equiv c$$

$$\Gamma(x, f, p, q) \equiv x$$

$$\Gamma(t_1 + t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) + \Gamma(t_2, f, p, q)$$

$$\Gamma(t_1 - t_2, f, p, q) \equiv \Gamma(t_1, f, p, q) - \Gamma(t_2, f, p, q)$$

$$\Gamma(g(t), f, p, q) \equiv g(\Gamma(t, f, p, q))$$

$$\Gamma(f(t), f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t' = \Gamma(t, f, p, q)$$
$$\quad in$$
$$\quad\quad ite(p = t', q, f(t'))$$

$$\Gamma(ite(t_1 = t_2, t_3, t_4), f, p, q) \equiv$$
$$\quad let$$
$$\quad\quad t'_1 = \Gamma(t_1, f, p, q), t'_2 = \Gamma(t_2, f, p, q),$$
$$\quad\quad t'_3 = \Gamma(t_3, f, p, q), t'_4 = \Gamma(t_4, f, p, q)$$
$$\quad in$$
$$\quad\quad ite(t'_1 = t'_2, t'_3, t'_4)$$

Figure 2.9: Definition of the Γ function

In addition, the ternary reachability predicate is also used as a set constructor, as an example consider the corresponding sets for $Btwn(f, t_1, t_3)$, the set for the field f will have t_2 as a member, whereas in the updated field, q will be an element and not t_2.

In order to capture this updates in the logic a Γ function is introduced.

$$\mathsf{wlp}(f(x) := y, \varphi) = \Gamma(\varphi, f, x, y)$$

The definition of the Γ function is given in figure 2.9, in most cases the computation is by straight forward recursion in the structure of the formula. The interesting cases are when the current formula contains a ternary reachability predicate or a set constructor which uses a field f which is the updated field.

In the following, we discuss the case for $\Gamma(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3, f, p, q)$ in detail. In order explain this the following predicate and the identity in Figure 2.10 are defined.

$$u \underset{w}{\xrightarrow{f}} v \equiv u \xrightarrow{f} v \xrightarrow{f} w \vee (u \xrightarrow{f} v \wedge \neg u \xrightarrow{f} w)$$

We will refer to this predicate as reachability with a forbidden node. Intuitively, the predicate holds if by starting at u and by taking 0 or more f links it can reach v and w is not in the path from u to v. In addition, let f^p_q denote the field exactly equal to f except that at position p its value is q. With the above predicate in mind, we describe identity showed in Figure 2.10. The first disjunct captures the case when the path from u over v to w already exist and p does not occur between u and w, idest, the update will not destroy the path. In the second disjunct, the path from u over v to p already exist and the update at p to q will create a path to w. In the third disjunct, the path from q over v to w already exists and the update at p to q will create a path from u to v. This captures all possible paths from u to w over v via the updated field f^p_q.

$$\begin{aligned} u \xrightarrow{f^p_q} v \xrightarrow{f^p_q} w \quad \equiv \quad & \vee \quad u \xrightarrow{f} v \xrightarrow{f} w \wedge u \underset{p}{\xrightarrow{f}} w \\ & \vee \quad p \neq w \wedge u \underset{w}{\xrightarrow{f}} p \wedge u \xrightarrow{f} v \xrightarrow{f} p \wedge q \underset{p}{\xrightarrow{f}} w \\ & \vee \quad p \neq w \wedge u \underset{w}{\xrightarrow{f}} p \wedge q \xrightarrow{f} v \xrightarrow{f} w \wedge q \underset{p}{\xrightarrow{f}} w \end{aligned}$$

Figure 2.10: Identity for ternary reachability over updated fields

Then the computation of $\Gamma(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3, f, p, q)$ is given by recursion on t_1, t_2, t_3 and applying the above described identity as described in figure 2.9.

2.3.3 Decision Procedure

In this section, we first present the decision procedure for deciding satisfiability of quantifier free formulas in LISBQ, then the decision procedure for ground formulas is extended to work with quantifiers.

The algorithm maintains a so called context, which is a conjunction of formulas asserted to be true. Then applies to the context a set of rewriting rules, a subset of this rules is shown in Figure 2.11. For the full set of rules see [23]. This subset is used to reason

about reachability predicates. For instance consider the rule cycle, the rule uses the fact that the application of f to t_1 is equal to t_1 and that t_2 is reachable from t_1 to conclude that there is a cycle on t_1 and thus $t_1 = t_2$. More formally, since $f(t_1) = t_1$ and t_2 is reachable from t_1 and since the definition of the ternary reachability predicate requires that there is a distinct u_i in between t_1 and t_2 it follows that u_i is unique and thus $t_1 = t_2$.

The decision procedure assumes the presence of an oracle which decides the combination of the theories of equality and arithmetical in where $. \xrightarrow{f} . \xrightarrow{f} .$ is uninterpreted. The context is called consistent if the set of literals in it are satisfiable, otherwise the context is called inconsistent. A rule is called a matching rule if the antecedents of the rule are in the context, the context is called saturated if for every matching rule the context contains one of the consequents.

$$\textit{reflexive}: \frac{}{t \xrightarrow{f} t} \qquad \textit{reach}: \frac{f(t_1) \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2 \quad t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3}$$

$$\textit{step}: \frac{f(t)}{t \xrightarrow{f} f(t)} \qquad \textit{cycle}: \frac{f(t_1) = t_1 \quad t_1 \xrightarrow{f} t_2}{t_1 = t_2}$$

$$\textit{sandwich}: \frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_1}{t_1 = t_2} \qquad \textit{order1}: \frac{t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3 \quad t_1 \xrightarrow{f} t_3 \xrightarrow{f} t_2}$$

$$\textit{order2}: \frac{t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_2,\ t_2 \xrightarrow{f} t_3} \qquad \textit{transitive2}: \frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_1 \xrightarrow{f} t \xrightarrow{f} t_2}{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t,\ t_0 \xrightarrow{f} t \xrightarrow{f} t_2}$$

$$\textit{transitive1}: \frac{t_1 \xrightarrow{f} t_2 \quad t_2 \xrightarrow{f} t_3}{t_1 \xrightarrow{f} t_3} \qquad \textit{transitive3}: \frac{t_0 \xrightarrow{f} t_1 \xrightarrow{f} t_2 \quad t_0 \xrightarrow{f} t \xrightarrow{f} t_1}{t_0 \xrightarrow{f} t \xrightarrow{f} t_2,\ t \xrightarrow{f} t_1 \xrightarrow{f} t_2}$$

Figure 2.11: Rules for reasoning about reachability

The decision procedure explores a decision tree while operating in the context. The context is initialized with the input formula φ and in every step a rewriting rule is applied. Whenever the current context is inconsistent, the decision procedure backtracks to the previous point and tries a new rule. If there is no matching rule, the decision procedure returns "unsatisfiable". Otherwise it continues applying rules until the context is saturated and returns that φ is satisfiable.

Example In this, we illustrate the decision procedure by deciding the validity of a formula $F : a \xrightarrow{f} b \wedge b \xrightarrow{f} c \Rightarrow a \xrightarrow{f} c$. Since the decision procedure decides satisfiability of a formula. We first negate F. After some simplifications, we obtain the formula $\neg F$ which will be the input of the decision procedure. We initialize the context C_1 whit the input formula. Then a basic rule *[and]* is applied three times, now the context C_2 consist of three literals. This literals are satisfiable. Therefore, our context is consistent. Since the context is not saturated, i.e. we can still apply rules, we apply the rule *[transitive1]*. This

leads to context C_3, here the literals $\neg a \xrightarrow{f} c, a \xrightarrow{f} c$ are a contradiction, Therefore the set of literals is unsatisfiable, making the context C_3 inconsistent.

At this point the decision procedure will backtrack to a previous satisfiable context, i.e. C_2. And apply any other matching rule. In order for a formula to be satisfiable. The context have to be saturated and consistent. However, since the rule *[transitive1]* is always applicable. The formula $\neg F$ in unsatisfiable. Therefore F is valid.

$$
\begin{array}{rcll}
F & : & a \xrightarrow{f} b \wedge b \xrightarrow{f} c \Rightarrow a \xrightarrow{f} c & \\
\neg F & : & \neg(\neg(a \xrightarrow{f} b \wedge b \xrightarrow{f} c) \vee a \xrightarrow{f} c) & \\
\neg F & : & a \xrightarrow{f} b \wedge b \xrightarrow{f} c \wedge \neg a \xrightarrow{f} c & \\
C_1 & = & \langle a \xrightarrow{f} b \wedge b \xrightarrow{f} c \wedge \neg a \xrightarrow{f} c \rangle & \textit{[and]}\times_3 \\
C_2 & = & \langle a \xrightarrow{f} b, b \xrightarrow{f} c, \neg a \xrightarrow{f} c \rangle & \textit{[transitive1]} \\
C_3 & = & \langle a \xrightarrow{f} b, b \xrightarrow{f} c, \neg a \xrightarrow{f} c, a \xrightarrow{f} c \rangle & \text{inconsistent}
\end{array}
$$

Figure 2.12 describes a possible execution of the decision procedure for another formula F which is satisfiable.

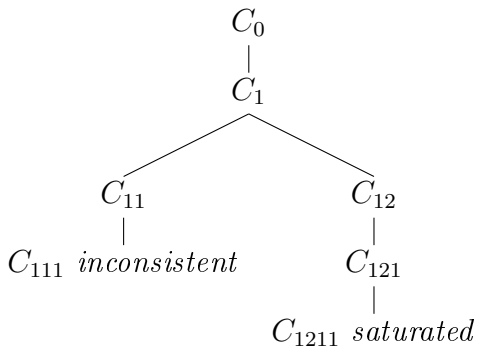

Figure 2.12: Possible execution tree of the decision procedure

The decision procedure for the quantified logic extends the set of rules with the rules given in Figure 2.13. If the current context contains the fact $t_1 \xrightarrow{f} t \xrightarrow{f} t_2$ then the rule *btwn* will instantiate the quantifier at t. The rule *inv* operates in a similar way.

$$
\textit{inv}\colon \frac{f(t') = t \quad \forall x \in f^{-1}(t).\alpha}{\alpha[x/t']} \qquad \textit{btwn}\colon \frac{t_1 \xrightarrow{f} t \xrightarrow{f} t_2 \quad \forall x \in \mathit{Btwn}(f, t_1, t_2).\alpha}{\alpha[x/t]}
$$

Figure 2.13: Quantifier instantiation

In [23] Lahiri and Qadeer proved that the described decision procedure terminates and that its complexity is in NP. Then the satisfiability problem for LISBQ is NP-complete.

Chapter 3

LISBQ in the Jahob Verification System

In this chapter we present our approach for deciding formulas in the LISBQ fragment of HOL. In Section 3.1, as the foundation of our implementation, we present the embedding of formulas in LISBQ to a fragment of HOL. In Section 3.2 we present a translation from Jahob's HOL formulas to the LISBQ fragment of HOL. Section 3.2.2, presents how to transalate formulas in the LISBQ fragment of HOL to FOL formulas. In Section 3.2.3 we present the implementation of the decision procedure and its use either on top of SMT-solvers or by using First-Order theorem provers.

3.1 Embedding LISBQ into HOL

In this section we present a α function for embedding formulas in LISBQ into a fragment of HOL which we call HOL LISBQ.

$$\alpha : \text{LISBQ} \rightarrow \text{HOL LISBQ}$$

The α function will preserve the types of LISBQ, i.e. the truth values, integers, objects, sets and functions of LISBQ will have the Jahob's HOL types of $\mathsf{bool}, \mathsf{int}$ and obj, t set and $t_1 \Rightarrow t_2$ respectively.

Figure 3.1 defines the α function. In the case of constants and variables the α returns the corresponding constants or variables as HOL variables, in the case of a function application α returns the HOL function and recurses on the argument. For the cases of addition, subtraction, inequality and equality the α returns the corresponding constant symbols of HOL and it is recursively applied to the arguments. For the case $\alpha(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3)$ we define a HOL constant symbol $\mathsf{reach3}$, whose type and semantics are defined in Table 3.1.

The semantics of the $\mathsf{reach3}$ constant are described by means of the $\mathsf{rtrancl\_pt}$ operator. The $\mathsf{rtrancl\_pt}$ operator was defined in Section 2.1.

The ternary reachability predicate of LISBQ is then defined in terms of the $\mathsf{reach3}$ constant and the α is applied to its arguments. The set constructor *Btwn* is also defined by means of the $\mathsf{reach3}$ constant symbol. For the cases of negation, conjunction and disjunction α is computed in a straight forward manner.

Constant	Type	Semantics
reach3$f\ t_1\ t_2\ t_3$	(obj ⇒ obj) ⇒ obj ⇒ obj ⇒ obj ⇒ bool	rtrancl_pt$(\lambda xy.fx = y \wedge x \neq t_3)t_1\ t_2$ $\wedge$ rtrancl_pt$(\lambda xy.fx = y)t_2\ t_3$

Table 3.1: HOL constant symbol for the Ternary Reachability Predicate

$$
\begin{aligned}
\alpha(c) &\equiv c \\
\alpha(x) &\equiv x \\
\alpha(f) &\equiv f \\
\alpha(f(t)) &\equiv f(\alpha(t)) \\
\alpha(t_1 + t_2) &\equiv \alpha(t_1) + \alpha(t_2) \\
\alpha(t_1 - t_2) &\equiv \alpha(t_1) - \alpha(t_2) \\
\alpha(t_1 < t_2) &\equiv \alpha(t_1) < \alpha(t_2) \\
\alpha(t_1 = t_2) &\equiv \alpha(t_1) = \alpha(t_2) \\
\alpha(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3) &\equiv \mathsf{reach3}\ \alpha(f)\ \alpha(t_1)\ \alpha(t_2)\ \alpha(t_3) \\
\alpha(Btwn(f, t_1, t_2)) &\equiv \{x.\mathsf{reach3}\ \alpha(f)\ \alpha(t_1)\ x\ \alpha(t_2)\} \\
\alpha(\forall x \in g^{-1}(t).F(x)) &\equiv \forall x.\alpha(g(x) = t) \rightarrow \alpha(F(x)) \\
\alpha(\neg F_1) &\equiv \neg\alpha(F_1) \\
\alpha(F_1 \wedge F_2) &\equiv \alpha(F_1) \wedge \alpha(F_2) \\
\alpha(F_1 \vee F_2) &\equiv \alpha(F_1) \vee \alpha(F_2)
\end{aligned}
$$

Figure 3.1: Definition of the function α

Theorem 1 *Let F be a LISBQ formula and $\mathcal{M}$ be a LISBQ model. Then F and $\alpha(F)$ are equisatisfiable. i.e.*

$$\mathcal{M} \models F \text{ iff } [\![\alpha(F)]\!]\alpha(\mathcal{M}) = \mathsf{true}$$

Proof. The proof is by structural induction on F. For the left-to-right direction, as base cases, consider the formulas F_1 and F_2 such that F_1 and F_2 are free of logical connectives. Then for F_1 or F_2 we have the following cases.

- $t_1 > t_2$, $t_1 = t_2$ or $f(t_1) = t_2$. For this cases, if $\mathcal{M} \models F_1$ then there exist a model for t_1, t_2 and f in $\mathcal{M}$. Therefore, $\alpha(\mathcal{M})$ will provide the HOL assignments such that $[\![\alpha(F_1)]\!]\alpha(\mathcal{M}) = \mathsf{true}$ as desired.

- $t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$. In this case, if $\mathcal{M} \models t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$ then by definition of the Ternary Reachability Predicate there exist distinct elements $u_0, u_1, \ldots, u_n$ in $\mathcal{M}$ such that

$\mathcal{M}_f(u_i) = u_{i+1}$ for all $i \in [0, n)$ and $u_0 = \mathcal{M}_{t_1}$, $u_n = \mathcal{M}_{t_3}$ and $u_i = \mathcal{M}_{t_2}$ for some $i \in [0, n]$. Then, by unfolding $\mathsf{reach3}\ f\ t_1\ t_2\ t_3$ we obtain.

$$[\![\mathsf{rtrancl\_pt}(\lambda xy.fx = y \wedge x \neq t_3)t_1\ t_2 \wedge\ \mathsf{rtrancl\_pt}(\lambda xy.fx = y)t_2\ t_3]\!]\alpha(\mathcal{M}) = \mathsf{true}$$

Then $\alpha(\mathcal{M})$ will map t_1, t_2, t_3 with u_0, u_i, u_n respectively. This evaluates to true as desired.

In the inductive proof, the cases for $\neg F_1$, $F_1 \wedge F_2$ and $F_1 \vee F_2$ follow directly by the induction hypothesis.
The right-to-left direction is analogous. First, we have to consider that

$$\mathsf{dom}(\alpha(\mathcal{M})) = \mathsf{const}(\mathsf{F}) \text{ and that } \mathsf{dom}(\mathcal{M}) = \Sigma_{\text{LISBQ}}$$

However, since $\alpha(\mathcal{M})$ contains all the constants occurring in F, then the model $\mathcal{M}$ can be safely extended in an arbitrarily manner.
For the base cases consider formulas F_1 and F_2 free of logical connectives. Then for F_1 or F_2 we have the following cases.

- $\alpha(t_1 > t_2)$, $\alpha(t_1 = t_2)$ or $\alpha(f(t_1) = t_2)$. For this cases, if $[\![\alpha(F_1)]\!]\alpha(\mathcal{M}) = \mathsf{true}$ then there exist a model for t_1, t_2 and f in $\alpha(\mathcal{M})$. Therefore, $\mathcal{M}$ will provide the LISBQ assignments such that $\mathcal{M} \models F_1$.

- $\alpha(t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3)$. In this case, if $[\![\mathsf{reach3}ft_1t_2t_3]\!]\alpha(\mathcal{M}) = \mathsf{true}$, i.e.

  $$[\![\mathsf{rtrancl\_pt}(\lambda xy.fx = y \wedge x \neq t_3)t_1\ t_2 \wedge\ \mathsf{rtrancl\_pt}(\lambda xy.fx = y)t_2\ t_3]\!]\alpha(\mathcal{M}) = \mathsf{true}$$

  Then we have in $\alpha(\mathcal{M})$ the following two paths, corresponding to the above conjuncts.

  $$\exists u_0, \ldots, u_i \text{ such that } u_0 = t_1, u_i = t_2 \text{ and } u_0, \ldots, u_{i-1} \neq t_3$$
  $$\exists u_i, \ldots, u_n \text{ such that } u_i = t_2, u_n = t_3$$

  and $f(u_i) = u_{i+1}$. Now we have to show that $u_0, \ldots, u_n$ are distinct. For the individual paths this follows from the fact that we choose the shortest path. For the distinctness of u in the two disjoint paths, assume that there is a common u in the two paths and let $j \in \{0, \ldots, i\}$ and $k \in \{i, \ldots, n\}$. Then

  $$u_j = u_k \Rightarrow u_{k+1} = u_{j+1} \Rightarrow u_{k+2} = u_{j+2} \Rightarrow \ldots$$

  We have two cases, the case when $u_{k+k'} = u_i$ will mean that there exist a shorter path which is a contradiction. For the case when $u_{j+j'} = u_n$ this is a contradiction since $u_0, \ldots, u_{i-1} \neq t_3 = u_n$. Therefore, $\mathcal{M}$ will map t_1, t_2, t_3 with u_0, u_i, u_n respectively and $\mathcal{M} \models t_1 \xrightarrow{f} t_2 \xrightarrow{f} t_3$.

In the inductive proof, the cases for$\neg F_1$, $F_1 \wedge F_2$ and $F_1 \vee F_2$ follow directly from the induction hypothesis. ∎

3.2 Deciding Validity of HOL by translation into HOL LISBQ

In this section we present our approach for deciding validity of a formula F in HOL by translating it to HOL LISBQ is depicted in Figure 3.2.

First, we take the formula F in HOL and by applying the transformations described in Section 3.2.1 we approximate it to HOL LISBQ.

Then, this formula F_1 will be translated to a formula F_2 in First-Order Logic by means of the β. In addition,the β will return a map m, which maps reachability predicates and fields to fresh variables.

Finally, we implement decision procedure for LISBQ by encoding the rules as universally quantified axioms. The function *Axioms* will return a conjunct of axioms in FOL. For this, first we apply the function α to the universally quantified axioms in LISBQ, this returns the axioms in HOL LISBQ. Then we transform this axioms to FOL by means of the β and the map m.

Our translation extends the translation described in [8], in the sense that it allows to maintain reflexive transitive closure operators, and reuses many transformations.

$$
\begin{array}{ll}
\mathit{let} & F_1 = \mathit{HOLtoHOL\text{-}LISBQ}(F) \quad \mathit{in} \\
\mathit{let} & F_2, m = \beta(F_1, \emptyset) \quad \mathit{in} \\
\mathit{let} & \mathit{Axioms}(f) = \\
 & \quad \mathit{let} \quad A, \_ = \beta(\alpha(\mathcal{A}_{1,f} \wedge \cdots \wedge \mathcal{A}_{10,f}), m) \\
 & \quad \mathit{in} \quad A \\
\mathit{in} & \\
 & (\bigwedge_{f \in \mathsf{dom}(m)} \mathit{Axioms}(f)) \Rightarrow F_2
\end{array}
$$

Figure 3.2: Deciding validity of HOL

Input language The input language comprise constructs such as lambda expressions, function updates, sets, tuples, quantifiers, cardinality operators and set comprehensions.

Splitting into sequents The Jahob's verification condition generator will formulate proof obligations consisting of conjunctions of multiple statements which represent all possible paths in the verified procedure as well as invariants, run time assertions and postconditions. Then this proof obligations are split into individual conjuncts which can be proven independently [8].
After splitting, the resulting formulae called sequents have the form

$$A_1 \wedge \cdots \wedge A_n \Rightarrow G$$

where $A_1 \wedge \cdots \wedge A_n$ are called the *assumptions* and G the *goal* of the sequent [34].

3.2.1 Translating HOL constructs to HOL LISBQ

After spliting of proof obligations, the resulting formula may have constructs that are not in HOL LISBQ. Therefore, the next step of the translation is to apply a set of rewrite rules that eliminate this constructs.

Definition Substitution and Function Unfolding When one of the assumptions is a variable definition, the translation substitutes its content in the rest of the formula (using the rules in Figure 3.3). This approach supports definitions of variables that have complex and higher-order types.

$$\textsc{Var-True}\quad \frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge v \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v := \mathsf{true}]}$$

$$\textsc{Var-False}\quad \frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge \neg v \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v := \mathsf{false}]}$$

$$\textsc{Var-Def}\quad \frac{(H_1 \wedge \cdots \wedge H_{i-1} \wedge v = e \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G}{\big((H_1 \wedge \cdots \wedge H_{i-1} \wedge H_{i+1} \wedge \cdots \wedge H_n) \Longrightarrow G\big)[v := e]} \qquad v \notin FV(e)$$

Figure 3.3: Rules for definition substitution

Flattening Flattening introduces fresh quantified variables, which could in princliple create additionall quantifier alternations, making the proof process more difficult. However, each variable can be introduced using either an existential or universal quantifier because $\exists v.v{=}e \wedge F$ is equivalent to $\forall v.v{=}e \rightarrow F$. Our translation therefore chooses the quantifier kind that corresponds to the most recently bound variable in a given scope (taking into account the polarity), preserving the number of quantifier alternations. The starting quantifier kind at the top level of the formula is $\forall$, ensuring that freshly introduced variables for quantifier-free expressions become Skolem constants.

Cardinality Constraints Constant cardinality constraints express natural generalizations of quantifiers. For example, the statement "there exists at most one element satisfying predicate P" is given by $\mathsf{card}(\{x.\, P\, x\}) \geq 1$. Our translation reduces constant cardinality constraints using the rules in Figure 3.4.

Set expresions Our translation uses universal quantification to expand set operations into their set-theoretic definitions in terms of the set membership operator. This process also elmininates set comprehensions by replacing $x \in \{y \mid \varphi\}$ with $\varphi[y \mapsto x]$. Figure 3.5

CARD-CONSTRAINT-EQ
$$\frac{\mathsf{card}(S) = k}{\mathsf{card}(S) \leq k \ \wedge \mathsf{card}(S) \geq k}$$

CARD-CONSTRAINT-LEQ
$$\frac{\mathsf{card}(S) \leq k}{\exists v_1, \ldots, v_k . S \subseteq \{v_1, \ldots, v_k\}}$$

CARD-CONSTRAINT-GEQ
$$\frac{\mathsf{card}(S) \geq k}{\exists v_1, \ldots, v_k. \ \{v_1, \ldots, v_k\} \subseteq S \wedge \bigwedge_{1 \leq i < j \leq k} v_i \neq v_j}$$

Figure 3.4: Rules for constant cardinality constraints

shows the rules for set expresions. These transformation ensure that the only set expressions in formulas are either set variables or set-valued fields occurring on the right-hand side of the membership operator.

SET-INCLUSION
$$\frac{S_1 \subseteq S_2}{\forall x. x \in S_1 \rightarrow x \in S_2}$$

SET-EQUALITY
$$\frac{S_1 = S_2}{\forall x. x \in S_1 \iff x \in S_2}$$

INTERSECTION
$$\frac{x \in S_1 \cap S_2}{x \in S_1 \wedge x \in S_2}$$

UNION
$$\frac{x \in S_1 \cup S_2}{x \in S_1 \vee x \in S_2}$$

DIFFERENCE
$$\frac{x \in S_1 \setminus S_2}{x \in S_1 \wedge x \notin S_2}$$

FINITESET
$$\frac{x \in \{O_1, \ldots, O_k\}}{x = O_1 \vee \cdots \vee x = O_k}$$

COMPREHENSION
$$\frac{x \in \{y \mid \varphi\}}{\varphi[y \mapsto x]}$$

Figure 3.5: Rules for complex set expresions

Tuples Tuples in the input language are useful, for example, as elements of sets representing relations, such as in an association list in which the first element would be the key and the second the corresponding value. The translation maps a variable x denoting a n-tuple into n individual variables $x_1, \ldots, x_n$ bound in the same way as x. A tuple equality becomes a conjunction of equalities of components. The arity of functions changes to accommodate all componets, so a function taking an n-tuple and an m-tubple becomes a function symbol of arity $n + m$.

Approximation The first part of our translation maps HOL formulas into HOL LISBQ, so there are constructs that it cannot translate exactly. e.g. symbolic cardinality constraints (as in BAPA [22]). Our HOL LISBQ translation approximates such subformulas

in a sound way, by replacing them with true or false depending on the polarity of the subformula occurrence. The result of the approximation is a stronger formula whose validity implies the validity of the original formula.

Rewriting Transitive Closure Predicates At this point, we are in HOL LISBQ, with the exception that the transitive closure is expresed in terms Jahob's rtrancl_pt operator. In this section, we look for rtrancl_pt predicates and rewrite them in terms of the reach3 predicate.

In contrast to the ternary reachability predicate of LISBQ the predicate rtrancl_pt, expresses binary reachability. It is important to note that the expressiveness of binary and ternary reachability differ, as an illustrating example consider Figure 3.6, using binary reachability we could express reachability from a to b and from b to c by $a \xrightarrow{f} b$ and $b \xrightarrow{f} c$ respectively. If we want to express c not to be in the path from a to b, this can be done in terms of ternary reachability as

$$a \underset{c}{\xrightarrow{f}} b \equiv a \xrightarrow{f} b \xrightarrow{f} c \vee (u \xrightarrow{f} v \xrightarrow{f} v \wedge \neg u \xrightarrow{f} w \xrightarrow{f} w)$$

However, this is slightly more difficult in terms of the rtrancl_pt predicate. Therefore, in order to preserve the expressiveness of LISBQ, we allow in the predicate p in $(a, b) \in \{(x, y).p\}^*$, to have the constructs described in table 3.2.

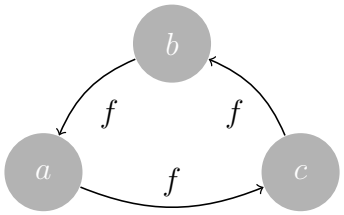

Figure 3.6: Expressiveness of binary and ternary reachability

Jahob notation	HOL LISBQ	LISBQ
rtrancl_pt($\lambda xy.fx = y$) u v	reach3 f u v v	$u \xrightarrow{f} v \xrightarrow{f} v$
rtrancl_pt($\lambda xy.fx = y \wedge y \neq w$) u v	reach3 f u v w $\vee$ (reach3 f u v v $\wedge$ $\neg$reach3 f u w w)	$u \underset{w}{\xrightarrow{f}} v$

Table 3.2: Constructs for expressing reachability

In the first construct, the formula f specifies the field for which transitive closure is required, this construct is then interpreted as binary reachability using the ternary reachability predicate, the second construct specifies additionally a variable w, which will be interpreted as ternary reachability with a forbidden node, i.e. the object w is not to appear in the path from u to v.

$$\Upsilon(x) \equiv x$$

$$\Upsilon(F :: t) \equiv \Upsilon(F) :: t$$

$$\Upsilon(F_1 F_2) \equiv \Upsilon(F_1)\Upsilon(F_2)$$

$$\Upsilon(\mathsf{rtrancl\_pt}(\lambda xy.fx = y)\ u\ v) \equiv$$
$$\quad let$$
$$\quad\quad F_1 = \mathsf{reach3}\ f\ u\ v\ v$$
$$\quad in$$
$$\quad\quad \Gamma'(F_1)$$

$$\Upsilon(\mathsf{rtrancl\_pt}(\lambda xy.fx = y \wedge y \neq w)\ u\ v) \equiv$$
$$\quad let$$
$$\quad\quad F_1 = \mathsf{reach3}\ f\ u\ v\ w, F_2 = \mathsf{reach3}\ f\ u\ v\ v, F_3 = \mathsf{reach3}\ f\ u\ w\ w$$
$$\quad in$$
$$\quad\quad \Gamma'(F_1 \vee (F_2 \wedge \neg F_3))$$

Figure 3.7: Definition of the Υ function

The translation searches recursively on the sequent for the constructs listed in table 3.2, if any is found then it its rewritten in terms of the reach3 predicate.

We define the function Υ which will recurse on the structure of the Jahob's HOL formulas searching for this constructs. The definition of the function Υ is given in figure 3.8. If Υ finds a variable or a constant then it returns the variable or constant. When Υ finds a typed formula it will recurse in the formula. For the case $\Upsilon(F_1F_2)$, which corresponds to a function application Υ is applied to $form_1$ and F_2 independently. The next cases are the most interesting ones since they contain the transitive closure operator; the matching case $\Upsilon(\mathsf{rtrancl\_pt}(\lambda xy.fx = y)\ u\ v)$ which expresses binary reachability from u to v will be rewritten by means of the reach3 predicate as F_1. Since the field f may contain field updates, we apply the Γ' function to F_1. The function Γ' is an adjustment of the function Γ presented in section 2.3.2, the function Γ' will be explained in the next section. The case $\Upsilon(\mathsf{rtrancl\_pt}(\lambda xy.fx = y \wedge y \neq w)\ u\ v)$ which expresses reachability with a forbidden node, is rewritten according to Table 3.2, where the reachability predicates correspond to F_1, F_2 and F_3. Since the field f may contain field updates, we apply the Γ' function to the resulting formula.

Rewriting Function Updates In the previos section we presented how to rewrite transitive closure predicates in Jahob's HOL into the reach3 predicate of HOL LISBQ. This new rewritten terms have the form of $\mathsf{reach3}\ ft_1\ t_2\ t_3$. However the formula f can have field updates. In this section we describe a Γ' function used for computing the weakest precondition with respect to the statement $f(p) := q$. The Γ' funtion is an adjustment of the Γ function described in Section 2.3.2 to our purposes. The main difference is that the Γ' function only needs to consider formulae containing disjunctions, conjunctions, negations and the reach3 predicate.

The full description of the Γ' function is presented in figure 3.8, the first three cases

$$\Gamma'(\neg F_1) \equiv \neg\Gamma'(F_1)$$
$$\Gamma'(F_1 \wedge F_2) \equiv \Gamma'(F_1) \wedge \Gamma'(F_2)$$
$$\Gamma'(F_1 \vee F_2) \equiv \Gamma'(F_1) \vee \Gamma'(F_2)$$
$$\Gamma'(\mathsf{reach3}\ f\ u\ v\ w) \equiv \mathsf{reach3}\ f\ u\ v\ w \qquad \text{when } f \text{ has no field updates}$$
$$\Gamma'(\mathsf{reach3}\ f\ u\ v\ w) \equiv \qquad \text{when } f \text{ has field updates}$$
$$\begin{array}{l} \quad \mathit{let} \\ \quad\quad f = \mathsf{rmvUpd}(f) \\ \quad \mathit{in} \\ \quad \mathit{let} \\ \quad\quad F_1 = \quad \mathsf{reach3}\ f\ u\ v\ w \wedge (\mathsf{reach3}\ f\ u\ w\ p \vee (\mathsf{reach3}\ f\ u\ w\ w \wedge \neg\mathsf{reach3}\ f\ u\ p\ p)) \\ \quad\quad F_2 = \quad p \neq w \wedge (\mathsf{reach3}\ f\ u\ p\ w \vee (\mathsf{reach3}\ f\ u\ p\ p \wedge \neg\mathsf{reach3}\ f\ u\ w\ w)) \wedge \\ \qquad\qquad \mathsf{reach3}\ f\ u\ v\ p \wedge (\mathsf{reach3}\ f\ q\ w\ p \vee (\mathsf{reach3}\ f\ q\ w\ w \wedge \neg\mathsf{reach3}\ f\ q\ p\ p)) \\ \quad\quad F_3 = \quad p \neq w \wedge (\mathsf{reach3}\ f\ u\ p\ w \vee (\mathsf{reach3}\ f\ u\ p\ p \wedge \neg\mathsf{reach3}\ f\ u\ w\ w)) \wedge \\ \qquad\qquad \mathsf{reach3}\ f\ q\ v\ w \wedge (\mathsf{reach3}\ f\ q\ w\ p \vee (\mathsf{reach3}\ f\ q\ w\ w \wedge \neg\mathsf{reach3}\ f\ q\ p\ p)) \\ \quad \mathit{in} \\ \quad\quad \Gamma'(F_1 \vee F_2 \vee F_3) \end{array}$$

Figure 3.8: Definition of the Γ' function

follow by straightforward recursion on the structure of the formula. For the next case, $\Gamma'(u \xrightarrow{f} v \xrightarrow{f} w)$ if f contains no field updates then no rewriting is needed, on the contrary if f has field updates, then the Γ' function with help from the rmvUpd function, will obtain a new f field in which one field update has been removed, then it will express as HOL LISBQ formulas in F_1, F_2 and F_3 the three disjuncts of the the identity used for expresing an updated field by means of the original field. This identity was described in Section 2.3.2 in Figure 2.10. Since the new obtainded f may contain field updates, it applies the Γ' function to the resulting formula.

3.2.2 Translating HOL LISBQ to FOL

In this section we present the β function which translates formulas in HOL LISBQ to formulas in FOL. In addition, the function β returns a map m from transitive closure predicates and fields to fresh variables. Figure 3.9 defines the function β.

HOL LISBQ is close to FOL except that in $\mathsf{reach3}\ f\ t_1\ t_2\ t_3$, reach3 takes a function f as an argument. Our approach replaces every occurrence of $\mathsf{reach3}\ f$ with fresh ternary predicate symbols while maintaining a map m. The map will be used to generate axioms according to the rules of the decision procedure that capture the meaning of the original $\mathsf{reach3}\ f$. Therefore, our translation from HOL LISBQ to FOL is sound and complete.

As an example, consider the predicate $\mathsf{reach3}\ f\ t_1\ t_2\ t_3$ then.

$$\beta(\mathsf{reach3}\ f\ t_1\ t_2\ t_3, \emptyset) = (V\_\_1\ t_1\ t_2\ t_3, \{\mathsf{reach3}\ f \mapsto V\_\_1\})$$

$$\beta(a, m) \equiv (a, m) \qquad \textit{for all atoms } a$$

$$\beta(\neg F, m) \equiv \neg\beta(F, m)$$

$$\begin{array}{l}\beta(F_1 \wedge F_2, m) \equiv \\ \quad \textit{let} \quad F_1', m_1' \equiv \beta(F_1, m) \quad \textit{in} \\ \quad \textit{let} \quad F_2', m_2' \equiv \beta(F_2, m_1') \quad \textit{in} \\ \quad (F_1' \wedge F_2', m_2')\end{array}$$

$$\begin{array}{l}\beta(F_1 \vee F_2, m) \equiv \\ \quad \textit{let} \quad F_1', m_1' \equiv \beta(F_1, m) \quad \textit{in} \\ \quad \textit{let} \quad F_2', m_2' \equiv \beta(F_2, m_1') \quad \textit{in} \\ \quad (F_1' \vee F_2', m_2')\end{array}$$

$$\begin{array}{l}\beta(\mathsf{reach3}\ f\ t_1\ t_2\ t_3) \equiv \\ \quad \textit{if} \quad m(f) \textit{defined then} \quad (m(f)\ t_1\ t_2\ t_3, m) \\ \quad \textit{else} \quad (V\ t_1\ t_2\ t_3, m \cup \{f \mapsto V\}) \quad V \textit{ is fresh}\end{array}$$

Figure 3.9: Translation from HOL LISBQ to FOL

3.2.3 Implementing the Decision Procedure

In this section we describe the implementation of the decision procedure for HOL LISBQ. After the translation described in the previous section we have sequents of the form.

$$A_1 \wedge \cdots \wedge A_n \Rightarrow G$$

Where $A_1 \wedge \cdots \wedge A_n$ are called assumptions and G is called the goal. The implementation of the decision procedure described in section 2.3.3 consist in encoding rules presented in Figure 2.11 for reasoning about reachability as universally quantified axioms. From the previous translation we obtain a map m which maps reachability predicates and fields to ternary predicates. We use this map to generate one set of axioms for each field appearing under the rtrancl_pt operator. Our implementation of the decision procedure uses the function *Axioms* defined in Figure 3.2. For The function *Axioms*, first the α is applied to the axioms in LISBQ the resulting axioms in HOL LISBQ will be transformed to according the map m to axioms in FOL. The sequents will have the following form.

$$A_1 \wedge \cdots \wedge A_n \wedge (\bigwedge_{f \in \mathsf{dom}(m)} \mathit{Axioms}(f)) \Rightarrow G$$

The Jahob system allows the use of several different specialized decision procedures, Figure 3.10 shows the architecture of the Jahob system. We see that the translation is available at two modules, i.e. at the FOL interface and at the SMT-LIB interface.

LISBQ via the SMT-LIB interface In the case of SMT-solvers, our implementation uses the advantage that some SMT-solvers support triggers, this triggers will restrict unnecessary quantifier instantiations. The universally quantified *Transitive1* axiom with triggers will have the following form.

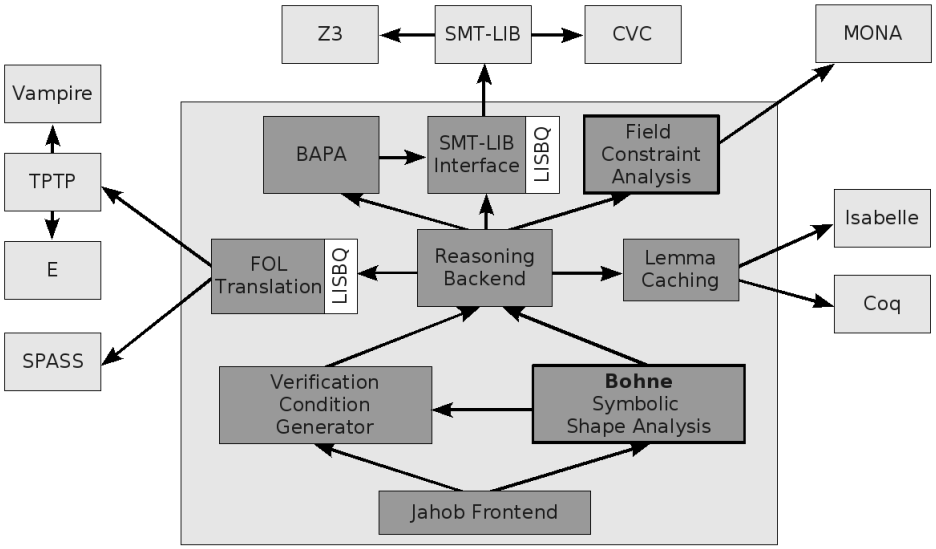

Figure 3.10: Jahob system architecture.

$$\forall x, y, z.\ \ \{\mathsf{reach3}\ f\ x\ y\ y, \mathsf{reach3}\ f\ y\ z\ z\}\ \mathsf{reach3}\ f\ x\ y\ y, \mathsf{reach3}\ f\ y\ z\ z \Rightarrow \mathsf{reach3}\ f\ x\ z\ z$$

Figure 3.11, shows a SMT file for a simple proof obligation. In the file, the predicate (V__1) is the product of the translation form HOL LISBQ to FOL and it is the mapping of a reach3 predicate with a field. After the function definitions, the universally quantified axioms with their corresponding triggers are written listed.

LISBQ via the FOL interface In contrast to SMT-solvers Resolution-based First-Order provers do not support triggers. Figure 3.12 shows a SPASS [33] file, again the predicate (V__1) is a map from a reach3 predicate and a field. We see that the universally quantified axioms appear under the list of axioms of the problem.

An important difference with SMT-solvers is that Resolution-based First-Order provers do not have built-in arithmetic operations. Our translation therefore introduces axioms that provide a partial axiomatization of integer operations $+, <, \leq$. In addition, The translation supplies axioms for the ordering relation between all numeric constants appearing in the input formula. Although incomplete, these axioms are sufficient to verify list, tree and hash table data structures.

Proof obligation:

$$[|(a,b) \in \{(x,y).x..next = y\}^* \wedge \\ (b,c) \in \{(x,y).x..next = y\}^*|] \\ \Longrightarrow (a,c) \in \{(x,y).x..next = y\}^*$$

SMT input file:

```
(benchmark vc0.smt

  :logic AUFLIA

  :extrasorts (Obj)
  :extrafuns ((null Obj))
  :extrafuns ((a Obj))
  :extrapreds ((v__1 Obj Obj Obj))
  :extrafuns ((next Obj Obj))
  :extrafuns ((c Obj))
  :extrafuns ((b Obj))

  :assumption (v__1 a b b)
  :assumption (v__1 b c c)
  :assumption (= (next null) null)
  :assumption
   (forall (?t1 Obj) (forall (?t2 Obj) (forall (?t3 Obj)
   (or (not (v__1 ?t1 ?t2 ?t2)) (not (v__1 ?t2 ?t3 ?t3)) (v__1 ?t1 ?t3 ?t3))
   :pat { (v__1 ?t1 ?t2 ?t2)(v__1 ?t2 ?t3 ?t3)} )))
  :assumption
   (forall (?t Obj) (forall (?t0 Obj) (forall (?t1 Obj) (forall (?t2 Obj)
   (or (not (v__1 ?t0 ?t1 ?t2)) (not (v__1 ?t1 ?t ?t2)) (and (v__1 ?t0 ?t1 ?t) (v__1 ?t0 ?t ?t2)))
   :pat { (v__1 ?t0 ?t1 ?t2)(v__1 ?t1 ?t ?t2)} ))))
  :assumption
   (forall (?t Obj) (forall (?t0 Obj) (forall (?t1 Obj) (forall (?t2 Obj)
   (or (not (v__1 ?t0 ?t1 ?t2)) (not (v__1 ?t0 ?t ?t1)) (and (v__1 ?t0 ?t ?t2) (v__1 ?t ?t1 ?t2)))
   :pat { (v__1 ?t0 ?t1 ?t2)(v__1 ?t0 ?t ?t1)} ))))
  :assumption
   (forall (?t1 Obj) (forall (?t2 Obj) (forall (?t3 Obj)
   (or (not (v__1 ?t1 ?t2 ?t2)) (not (v__1 ?t1 ?t3 ?t3)) (v__1 ?t1 ?t2 ?t3) (v__1 ?t1 ?t3 ?t2))
   :pat { (v__1 ?t1 ?t2 ?t2)(v__1 ?t1 ?t3 ?t3)} )))
  :assumption
   (forall (?t1 Obj) (forall (?t2 Obj) (forall (?t3 Obj)
   (or (not (v__1 ?t1 ?t2 ?t3)) (and (v__1 ?t1 ?t2 ?t2) (v__1 ?t2 ?t3 ?t3)))
   :pat { (v__1 ?t1 ?t2 ?t3)} )))
  :assumption
   (forall (?t Obj) (v__1 ?t ?t ?t))
  :assumption
   (forall (?t1 Obj) (forall (?t2 Obj)
   (or (not (v__1 ?t1 ?t2 ?t1)) (= ?t1 ?t2))
   :pat { (v__1 ?t1 ?t2 ?t1)} ))
  :assumption
   (forall (?t1 Obj) (forall (?t2 Obj)
   (or (not (= (next ?t1) ?t1)) (not (v__1 ?t1 ?t2 ?t2)) (= ?t1 ?t2))
   :pat { (v__1 ?t1 ?t2 ?t2)} ))
  :assumption
   (forall (?t1 Obj) (forall (?t2 Obj)
   (or (not (v__1 ?t1 ?t2 ?t2)) (= ?t1 ?t2) (v__1 ?t1 (next ?t1) ?t2))
   :pat { (next ?t1)(v__1 ?t1 ?t2 ?t2)} ))
  :assumption
   (forall (?t Obj) (v__1 ?t (next ?t) (next ?t))
   :pat { (next ?t)} )

  :formula    (not (v__1 a c c))

  :status     unknown
)
```

Figure 3.11: SMT file

Proof obligation:

$$\begin{array}{c}[|(a,b) \in \{(x,y).x..next = y\}^* \wedge \\ (b,c) \in \{(x,y).x..next = y\}^*|] \\ \Longrightarrow\ (a,c) \in \{(x,y).x..next = y\}^*\end{array}$$

FOL, SPASS input file:

```
begin_problem(jahob).

list_of_symbols.
functions[c, b, null, next, a].
predicates[_V__1].
sorts[int,obj,bool,float,unknown_sort].
end_of_list.

list_of_formulae(axioms).
formula(equal(null, next(null))).
%%%%%%%%%%universally quantified axioms for LISBQ%%%%%%%%%%%%%%%%%%%%%%%%
formula(forall( [_T1], forall( [_T2], forall( [_T3], or(or(not(_V__1(_T1, _T2, _T2)),
not(_V__1(_T2, _T3, _T3))), _V__1(_T1, _T3, _T3)))))).

formula(forall( [_T], forall( [_T0], forall( [_T1_1], forall( [_T2_1], or(or(not(_V__1(
_T0, _T1_1, _T2_1)), not(_V__1(_T1_1, _T, _T2_1))), and(_V__1(_T0, _T1_1, _T), _V__1(_T0,
_T, _T2_1)))))))).

formula(forall( [_T_1], forall( [_T0_1], forall( [_T1_2], forall( [_T2_2], or(or(not(_V__1(
_T0_1, _T1_2, _T2_2)), not(_V__1(_T0_1, _T_1, _T1_2))), and(_V__1(_T0_1, _T_1, _T2_2), _V__1(
_T_1, _T1_2, _T2_2)))))))).

formula(forall( [_T1_3], forall( [_T2_3], forall( [_T3_1], or(or(not(_V__1(_T1_3, _T2_3,
_T2_3)), not(_V__1(_T1_3, _T3_1, _T3_1))), or(_V__1(_T1_3, _T2_3, _T3_1), _V__1(_T1_3,
_T3_1, _T2_3)))))).

formula(forall( [_T1_4], forall( [_T2_4], forall( [_T3_2], or(not(_V__1(_T1_4, _T2_4, _T3_2)),
and(_V__1(_T1_4, _T2_4, _T2_4), _V__1(_T2_4, _T3_2, _T3_2)))))).

formula(forall( [_T_2], _V__1(_T_2, _T_2, _T_2))).

formula(forall( [_T1_5], forall( [_T2_5], or(not(_V__1(_T1_5, _T2_5, _T1_5)),
equal(_T1_5, _T2_5))))).

formula(forall( [_T1_6], forall( [_T2_6], or(or(not(equal(_T1_6, next(_T1_6))),
not(_V__1(_T1_6, _T2_6, _T2_6))), equal(_T1_6, _T2_6))))).

formula(forall( [_T_3], forall( [_Fun_flat_foltrans_2, _Fun_flat_foltrans_1],
or(or(not(equal(_Fun_flat_foltrans_2, next(_T_3))), not(equal(_Fun_flat_foltrans_1,
next(_T_3)))), _V__1(_T_3, _Fun_flat_foltrans_2, _Fun_flat_foltrans_1))))).

formula(forall( [_T1_7], forall( [_T2_7], or(not(_V__1(_T1_7, _T2_7, _T2_7)),
or(equal(_T1_7, _T2_7), forall( [_Fun_flat_foltrans_3], or(not(equal(_Fun_flat_foltrans_3,
next(_T1_7))), _V__1(_T1_7, _Fun_flat_foltrans_3, _T2_7)))))))).
%%%%%%%%%%universally quantified axioms for LISBQ%%%%%%%%%%%%%%%%%%%%%%%%

formula(_V__1(b, c, c)).
formula(_V__1(a, b, b)).
end_of_list.

list_of_formulae(conjectures).
formula(_V__1(a, c, c)).end_of_list.
end_problem.
```

Figure 3.12: FOL, SPASS file

Chapter 4

Evaluation

We evaluated our implementation on varios example programs that manipulate heap-allocated data structures. These programs implement singly-linked lists, cyclic lists, sorted lists, association lists and threaded trees.

Our evaluation showed, that the use of LISBQ helps in both, increasing the scope of verifiable data structures and reducing substantially the annotations on data structure implementations that have been previously verified in Jahob. In addition, we used LISBQ in combination with other specialized decision procedures to verify properties of operations on threaded trees with integer keys. The verified properties are beyond the scope of any individual decision procedure that is integrated in the Jahob verification system, including LISBQ itself.

4.1 Singly-Linked Lists

Our first example shows the usefulness of our implementation for the verification of acyclic lists. Figure 4.1 depicts the structure of an acyclic null terminated singly-linked list, which we consider in this section.

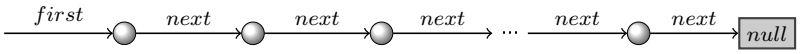

Figure 4.1: Singly-linked list

Figure 4.2 shows the declaration of the class Node and List that implement the data structure. Each node object contains a field next and a boolean field data, the class list maintains a reference first to the first node of the list. The class further declares two specification variables. The specification variable reach is a shorthand for the reflexive transitive closure of the function next. It expresses reachability within the nodes of the singly-linked list. The second specification variable content is a set containing all the nodes different from that can be reached starting from the first node of the list. Finally, a class invariant is given. It states that the null is reachable from first, i.e., that the list is acyclic.

The annotated method remove listed in Figure 4.2 takes as input parameter a node n, which if found, will be deleted from the list. The precondition of method remove requires that the input node n is not null. The postcondition of method remove ensures that the

```
public final class Node {
    public /*: claimedby List */ Node next;
    public /*: claimedby List */ boolean data;
}
public class List
{
    private static Node first;  /*:
     private static specvar reach :: "obj => obj => bool";
       vardefs "reach == (λ a b. rtrancl_pt (λ x y. x..Node.next=y) a b)";
     public static specvar content :: objset;
       vardefs "content == {x. x ≠ null & reach first x}";
     invariant "comment ''acyclic'' (reach first null)";      */

    public static void remove(Node n)
    /*: requires "n ≠ null"
        modifies content
        ensures "content = old content − {n}" */
    {
        Node e = first;  Node prev = null;
        while
           /*: inv " (reach first e) ∧ (reach first null) ∧
                    (reach first prev) ∧ (reach e n | n ∉ content) ∧
                    (prev ≠ n ) ∧ (prev = null → e = first) ∧
                    (prev ≠ null → prev..next = e)  ";    */
            (e != null && e != n)  {
               prev = e; e = e.next;
        }
       if (e != null) {
          if (prev != null) {
              prev.next = e.next;
          } else {
              first = first.next; }
          e.next = null;
       }
    }
    public static void filter ()
    /*: modifies content
        ensures " ∀ n. n ∈ content → ¬n..data" */
    {   ...
        while /*: inv "(e ≠ first → prev ≠ null) ∧
                       (reach first e) ∧ (reach e null) ∧
                       (prev ≠ null →
                          (ALL v. reach v prev ∧ null ≠ v ∧ e ≠ v ∧ reach first v →
                       ¬v..data) ) ∧
                       (prev ≠ null →
                          ¬prev..data ∧ prev..next = e ∧ reach first prev ) "; */
            (e != null) {
             ...
        }
    }
}
```

Figure 4.2: Class defining a linked list

node n is not a member of content, i.e., that n has been removed from the list. A loop invariant is provided, basically it states that e and prev are always reachable by starting at first and n is different than all the previous seen nodes. In order to prove the postcondition, essentially two cases have to be considered. The first case is when the node n is part of the content and n is deleted from the list, this will follow from the fact that first can reach null and that n is no longer in the path from first to null. The second case is when n is not in the list. In this case we need to show that the path from first to null is closed and that n is not in the path. This is implied by the consequent prev..next = e and the conjunct prev = n.

In Figure 4.2 we present the annotated method filter, for convenience the source code is not shown. The method will traverse the list by using the reference e until it reaches null, it will remove from the path all the nodes whose data value is true. The method returns a list containing only nodes in which its data field is set to false. The postcondition ensures that for all the nodes in the set content the data field of the node has to be false. The loop invariant, states that e and prev are reachable from first and that for all nodes between first and prev their data field is false. Note, that since the list is acyclic, it is possible to express the set of nodes in between first and prev, without the use of the ternary reachability predicate with a forbidden node defined in Section 2.3.2. However, this is not the case for cyclic lists as we will see in Section 4.2.

4.2 Cyclic Lists

Previously Jahob did not support reasoning about reachability in the presence of cycles. Still, the given example could be verified by simulating a cyclic list using and acyclic list, leading to an increased annotation burden of the user of the system. LISBQ allows us to directly reason about cyclic lists.

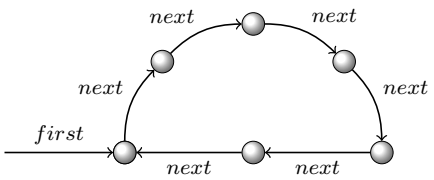

Figure 4.3: Cyclic List

In Figure 4.4 we give the declaration of the classes Node and List, every node contains a field next. Figure 4.3 presents an example of such a list. The List class maintains a reference first to the first node of the list. In addition, the specification variable content represents all the elements of the list. The class maintains a class invariant which expresses the cyclicity of the list by stating that first is reachable from its next successor.

The method add, is listed in Figure 4.4, it takes as an argument a node which will be inserted to the cyclic list, if the list is empty i.e. first equals null, then the given node will also be referenced as first. The precondition requires that n is different from null and not already contained in the list. The postcondition ensures that n is properly inserted into the list.

```
public class Node {
    public /*: claimedby List */ Node next;
}
public class List {
    private static Node first;
   /*:
     invariant " first ≠ null → (first..next, first ) ∈ { (v,w). v..next=w}* ";

     public static specvar content :: objset;
      vardefs
        "content == {x. x ≠ null & (first..next,x) ∈ {(v,w). v..next=w & w ≠ first }*}" */
    public static void add(Node n)
    /*: requires "(n ∉ content & n ≠ null)"
        modifies content
        ensures "(content = old content ∪ {n})" */
    {
        if ( first == null) {
            first = n;  first.next = n;
        }
        else {
            n.next = first.next;  first.next = n;
        }
    }
    public static boolean member(Node n)
    /*: ensures "result = ( n ∈ content) "      */
    {
        if ( first == null)
            return false;
        Node tmp = first.next;
        if ( first == n) {
            return true;
        } else {
        //: ghost specvar seen :: objset = "{}"
        //: seen := "seen ∪ {first}"
            while /*:inv "
                        tmp ∈ content
                        ∧ (comment ''progress'' (tmp ≠ first →
                        seen = {x. ( first , x) ∈ {(v,w). v..next=w ∧ w ≠ tmp}*} − {tmp}))
                        ∧ (tmp = first → seen = content)
                        ∧ (∀ x. x : seen → x ≠ n) " */
            (tmp != first) {
            //: seen := "seen ∪ {tmp}"
            if(tmp == n) {
                return true;
            }
            tmp = tmp.next;
            }
        //: noteThat seenAll: "seen = content";
        return false;
        }
    }
}
```

Figure 4.4: Cyclic list class declarations

Figure 4.4 lists the method member, which takes a node n as an argument. It traverses the cyclic list starting at the next node of first until it reaches the node first. If the node n is found on the path then it will return true and false otherwise. The method has no preconditions, and the postcondition ensures that the returned value is true iff the node n is a member of the set content. To ensure the postcondition, we basically have to consider two cases. The first case is when the node is a node of the list, in this case, since tmp is always part of the content and tmp is equal to the node n, the decision procedure can easily conclude that n is in the content. The second case is when the node is not a member of the list. Showing this is a slightly more complicated, in this case we have chosen the strategy of constructing an specification variable object set called seen, which contains all the visited nodes, at the end of the while loop the set seen will be asserted to be equal to the set content. Constructing the set seen is straightforward. After comparing the node first with n, the new set seen will be the union of the empty set and the first node, after each loop iteration a new member will be included in the set seen. The loop invariant states this progress of the elements in the seen set by using reachability without visiting a forbidden node. It specifies that the set seen is equal to all the nodes reachable by starting at node first until the node tmp. Note that without this predicate, we can not represent a subset of the reachable nodes on a cyclic list. To elucidate the use of this predicate we trace the member method as if it were executed on a cyclic list with more than one element. If first is different than the node n then we go to the else branch and $seen = \{first\}$ then if tmp is not equal to first ($first \neq next(first)$) the loop invariant of the conjunct labeled as "progress" will evaluate as follows.

$$
\begin{aligned}
seen &= \{x.first \xrightarrow[tmp]{f} x\} \setminus \{tmp\} && (4.1)\\
&= \{x.first \xrightarrow{f} x \xrightarrow{f} tmp \vee (first \xrightarrow{f} x \wedge \neg first \xrightarrow{f} tmp)\} \setminus \{tmp\} && (4.2)\\
&= \{x.first \xrightarrow{f} x \xrightarrow{f} tmp \vee (false)\} \setminus \{tmp\} && (4.3)\\
&= \{x.first \xrightarrow{f} x \xrightarrow{f} next(first) \vee (false)\} \setminus \{next(first)\} && (4.4)\\
&= \{first, next(first)\} \setminus \{next(first)\} && (4.5)\\
&= \{first\} && (4.6)
\end{aligned}
$$

Equation 4.1 and 4.2 unfold the ternary reachability predicate with a forbidden node in terms of the ternary reachability predicate. In equation 4.3, since tmp is always reachable from first, the second disjunct of the first set comprehension evaluates to false. In equation 4.4 by definition of the ternary reachability predicate x can satisfy the predicate by being either $first$ or $next(first)$ Finally tmp is subtracted. In every iteration the set of nodes in seen increases until either the list is completely traversed or the node n is found.

4.3 Sorted Lists

In this section we consider a null terminated single list whose nodes are sorted according to their integer key fields. The main difficulty of this example is that we need to reason about reachability and arithmetic. In previous approaches and since Monadic Second-Order Logic does not support arithmetic. There was a need to combine MSOL with a decision procedure for arithmetic. As a result, the proof was slightly more complicated that the one we provide. In addition, there was an increased number of annotations.

In Figure 4.6 the declarations of the class Node and and class SortedList are presented. The class node declares a field next and a field key of type integer. The class SortedList maintains a reference to the first node of the list, it also declares two specification variables. The specification variable reach is syntactic sugar for the reflexive transitive closure predicate of the next field and the specification variable content containing all the nodes different than null that are reachable starting at first until null. The specification also declares a class invariant, which expresses the sortedness of the list, by stating that if the node y is the next node of the node x then the key value of x is strictly less than the key value of y. In other words, the next field induces a total order on the nodes in the content set.

Figure 4.6 presents the annotated remove method, the method takes as input a node n, then it will traverse the list maintaining a reference to the current node, at every iteration the while condition evaluates if the end of the list has been reached or if the key value of the current node is greater than the key value of the node, implying that the node n is not a member of content.

The method precondition requires the node n not to be null and the postcondition ensures that the set content will not contain the node n. The loop invariant states that the previous visited nodes and the current node are always in the set content.

To ensure the postcondition, we have to consider the following cases. The case when the node is part of the content, this is directly implied by the source code which removes the node n from the path. For the case when the node n is not in the list, we have two sub-cases. The case when current equals null, this follows from the fact that the node n is not in the path from first to null and finally the case when the key value of the node n is greater than the key value of current, this case follows from the sortedness invariant.

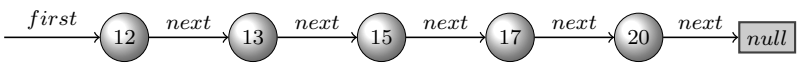

Figure 4.5: Sorted list

```
public final class Node {
    public /*: claimedby SortedList */ int key;
    public /*: claimedby SortedList */ Node next;
}
class SortedList
{
    private static Node first;
    /*:
      private static specvar reach :: "obj => obj => bool";
      vardefs
        "reach == (λ a b. rtrancl_pt (λ x y. x..Node.next=y) a b)";

      public static specvar content :: objset;
      vardefs
        "content == {n. n ≠ null ∧ reach first n}";

      invariant "(∀ x y. x ∈ content ∧ (reach x..next y) ∧ y ≠ null
                          → x..key < y..key)"; */


public static void remove(Node n)
/*: requires "n ≠ null"
    modifies content
    ensures "content = old content − {n}" */
{
    Node prev = null;    Node current = first;
    while /*:
            inv "(prev ≠ null → prev..next = current) ∧
                 (prev = null → current = first) ∧
                 (current ≠ null → current ∈ content) ∧
                 (prev ≠ null → prev ∈ content) ∧
                 (reach current n ∨ n ∉ content)∧
                 null..next = null " */
        ((current != null) && (current.key < n.key)) {
        prev = current;
        current = current.next;
    }
    if (current == n) {
        if (prev != null) {
            prev.next = current.next;
        } else {
            first = current.next;
        }
        current.next = null;
    }
}
```

Figure 4.6: Class defining a sorted list

4.4 Association Lists

Our next example presents an association list, which maps unique keys to their corresponding values. The main difficulty in this data structure is the potential sharing of nodes, i.e., two different keys could map to a single object. A previous verification attempt of this data structure combined several techniques, including interactive theorem proving and manually provided lemmas. Despite of these efforts, some methods previously remained unverified.

In this example, many of the specifications contain existential and universal quantifiers. Therefore, this example nicely demonstrates the advantage of our alternative implementation of the LISBQ decision procedure that uses First-Order resolution theorem provers based on resolution such as SPASS. Since these provers have a better support for quantifiers than SMT-solvers, they were able to verify methods for which the implementation based on SMT-solvers failed. In our example this methods are the methods add and remove.

Figure 4.7 shows the structure of the association list. The backbone of the structure is given by the next field, which connects the nodes from first to null. Every node has in addition to the next field two fields, the field key and the field value. Both of type object.

In Figure 4.8, we provide the declaration of the class Node and the association list class. The association list class, maintains a reference first to the first node of the list. The class declaration describes three specification variables. The specification variable inlist is a set containing all the nodes different than null which are reachable from firstby following first. The specification variable content is a set whose elements are tuples (key,value) corresponding to the mappings implemented by the association list. The specification variable reach is a shorthand for expressing reachability between to nodes along the field next.

In addition to the specification variables, three class invariants are declared. The class invariant "acyclic", states acyclicity by specifying that first can always reach null. The invariant "injective", specifies that the field key is an injective function. This ensures the uniqueness of the keys in the list.

Figure 4.9 presents the method add, which takes as arguments a key and a value, it then adds the key and the value to the association list. The methods precondition requires both key and value to be different from null. The postcondition states that the tuple (key,value) is properly inserted in the set content.
The add method uses a reference n which, starting at first, traverses the list looking for the argument key. If key is found, then it will overwrite the field value with the new value. If key is not found. It will create a new node, and assign the arguments key and value to

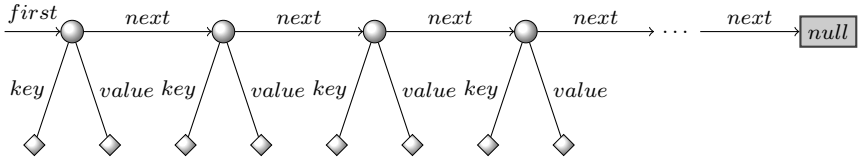

Figure 4.7: Association List

```
public final class Node {
    public /*: claimedby AssocList */ Object key;
    public /*: claimedby AssocList */ Object value;
    public /*: claimedby AssocList */ Node next;
}
public final class AssocList
{
    private static Node first;
    /*:
      private static specvar inlist :: objset;
      vardefs
        " inlist == {n. n ≠ null ∧ rtrancl_pt (λ x y. x..Node.next=y) AssocList.first n}";
      public static specvar content :: "(obj * obj) set";
      vardefs
        "content == {(k,v). ∃ n. k = n..Node.key ∧ v = n..Node.value ∧ n ∈ inlist}";
      private static specvar reach :: "obj => obj => bool";
      vardefs
        "reach == (λ a b. rtrancl_pt (λ x y. x..Node.next=y) a b)";
      invariant "∀ v. v ∈ inlist → (v..Node.key ≠ null) ∧ (v..Node.value ≠ null)";
      invariant "comment ''injective'' (∀ x y. x ∈ inlist ∧ y ∈ inlist
         ∧ x..key = y..key → x = y)";
      invariant "comment ''acyclic'' (reach first null)";    */
```

Figure 4.8: Class Defining an association list

its key and value fields.
In the loop invariant. The first conjunct specifies that first can reach n. The second conjunct specifies that for all the already visited nodes, their key is different from key.

The method remove listed in Figure 4.9, was previously verified by using field constraint analysis, lemmas and interactive theorem proving. The previous version included over twenty lines of annotations. Our loop invariant, in contrast is much simpler. The method takes as argument a key object and removes the node from the list whose key field matches with the argument key.

The precondition of this method requires the argument key to be different from null and thatkey has an entry in the list. The postcondition ensures that the new set content is equal to the old set content minus any tuple including key. The method, will traverse the list starting at first until it finds the key while maintaining the references prev and n. If prev is equal to null then the reference first will be updated to the next node of first. In the case when prev is different from null then the next node of prev is set to the next node of node n.

The loop invariant. States that n is always different from null. The next conjunct specifies that if prev is not null, then first can reach prev an the next node of prev is n. The next conjunct asserts that if prev is null then the iteration have not started and thus first=n. Since the precondition requires the key to be in the set content, the next conjunct expresses that there exist some reachable node from n such that the key value of this node is equal to the argument key. The last conjunct specifies that the key fields of all visited nodes are different from key.

```
public static void add(Object key, Object value)
/*: requires "key ≠ null ∧ value ≠ null"
    modifies content
    ensures "content = (old content \ {(x,y). x=key}) ∪ {(key,value)}"   */
{
    Node n = first;
    while /*: inv "reach first n ∧
                   (∀ v. v ≠ null ∧ reach first v ∧ ¬reach n v → v..Node.key ≠ key)" */
        ((n != null) && (n.key != key)) {
        n = n.next;
    }
    if (n==null) {
        n = new Node();
        n.key = key;
        n.value = value;
        n.next = first;
        first = n;
    } else {
        n.value = value;
    }
}

public static void remove(Object key)
/*: requires "key ≠ null ∧ (∃ v. (key,v) ∈ content)"
    modifies content
    ensures "content = old content \ {(x,y). x = key}"    */
{
   Node prev = null;
   Node n = first;
   while /*: inv "n ≠ null ∧
                  (prev ≠ null → reach first prev ∧ prev..next = n) ∧
                  (prev = null → first = n) ∧
                  (∃ v. v ≠ null ∧ reach n v ∧ v..Node.key = key) ∧
                  (∀ v. v ≠ null ∧ reach first v ∧ ¬reach n v → v..Node.key ≠ key)" */
      (n.key != key) {
      prev = n;
      n = n.next;
   }
   if (prev == null) {
       first = first.next;
   } else {
      prev.next = n.next;
   }
}
```

Figure 4.9: Methods remove and add

4.5 Threaded Trees

In this example we illustrate how LISBQ can be combined with other decision procedures in order increase the scope of verifiable data structures in the Jahob system.

In Figure 4.10 a binary search tree is presented, every node consist of the left and right fields. In addition, every node contains an integer field and a next field. The solid lines represent the conventional left and right fields. The doted line represents a next field, which connects in-order all the nodes of the tree, giving rise to a sorted list whose first element is the left most child node of the root of the tree.

With LISBQ we are able to reason about reachability in list like data structures. However, in this case we need to reason about the set of reachable nodes starting at the root of the tree, to deal with this we make use of the technique *field constraint analysis* [34, 35] and MONA [18, 19], a decision procedure for Monadic Second-Order Logic over strings and trees. Briefly, a field constraint on a field restricts the set of objects to which this field can point. Consider figure 4.11 which list the declarations of the threaded tree class, a field constraint on the field next is declared. The field constraint has the form:

$$\forall vw.f(v) = v \leftrightarrow F(v, w)$$

which means that the field f (next) is functionally determined by the backbone fields which are declared by the invariant "tree [left,right]". The formula F defines the function f. The formula defining the field next, in our example, describes the cases in which a node y in the tree could be pointed through the next field by a node x. For example, if the node y is the next node of x and if the node x is not null but its right child is null then there must be an ancestor y of x such that x is reachable from the left child of y or the value of x is the greatest in the tree and y is null.

The specification variable content is then defined as all the nodes which are reachable starting at first by using the next field, the specification variable next_reach is a short hand for reachability with respect to field next. Furthermore, in the class declaration. Several class invariants are defined. Starting at the top after the tree invariant. The first invariant declares root as non reachable by following either the left or the right field. The second class invariant states that first is not reachable via the next field. The next class invariant specifies that first is always reachable from the root of the tree by following the left field. The next invariant states that if the root of the tree is not null then first is not null and the left child of the node first is always null. The next invariant states acyclicity

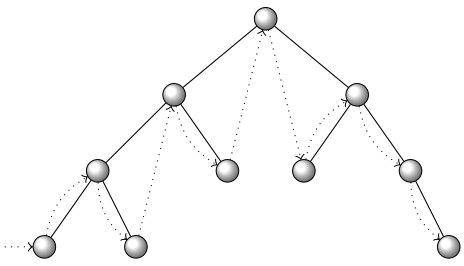

Figure 4.10: Threaded tree

of the sorted list. After the field constraint invariant. The last two invariants specify the sortedness of the list induced by the next field and the uniqueness of the key values.

```
public final class Node {
   public /*: claimedby Tree */ Node right;
   public /*: claimedby Tree */ Node left;
   public /*: claimedby Tree */ Node next;
   public /*: claimedby Tree */ int v;
}
public class Tree
{
   private static Node root;
   private static Node first;
   /*:
       public static specvar content :: objset;
       private vardefs " content ==
              {x. rtrancl_pt (λ x y. x..next = y ) first x}";
       static specvar next_reach :: "obj => obj => bool"
          vardefs "next_reach == (λ x y. rtrancl_pt (λ a b. a..next = b ) x y)";
       invariant "tree [left, right]";
       invariant "root ≠ null → (∀ n. n..left ≠ root ∧ n..right ≠ root)";
       invariant " first ≠ null → (∀ n. n..next ≠ first)";
       invariant "(root, first ) : {(x,y). x.. left = y}*";
       invariant "root ≠ null → first ≠ null ∧ first..left = null";
       invariant "next_reach first null";

       invariant "∀ x y. x..next = y ↔
          (x ≠ null →
            (x.. right = null →
               ((EX z. z.. left = x) → y..left = x) ∧
               ((∀ z. z.. left ≠ x) →
                    (((∀ z. ( left z, x) ~: {(x1,x2). x1.. right = x2}*)) ∧ (y = null) ∨
                    ( left y, x) : {(x1,x2). x1.. right = x2}*))) ∧
            (x.. right ≠ null →
               y.. left = null ∧ y ≠ null ∧ (right x, y) : {(x1, x2). x1.. left = x2}*)) ∧
          (x = null → y = null)";

       invariant "∀ x y. x ≠ null ∧ y ≠ null ∧ (x..right = y ∨ x..left = y) → y : content";
       invariant "∀ x y. x ≠ null ∧ y ≠ null ∧ x..Node.next = y → y : content";
       invariant "∀ x y. next_reach x y ∧ x ≠ null ∧ y ≠ null ∧ x ≠ y → x..v < y..v";
       invariant "∀ x y. x..v = y..v → x = y"; */
```

Figure 4.11: Threaded tree class declarations

Figures 4.12 and 4.13 list the annotated method member, that checks whether the given node e is contained in the tree. The method performs a standard search in the binary search tree, at every iteration it updates the boolean variable wentLeft depending on the value of the input node and the current node n and also maintains the references pred and succ. The search ends when the node n is null indicating that the input node e was not found or when the node n equals the input node e indicating that the node e was found. The method precondition requires e not to be null and the postcondition ensures that the returned value is true iff the node e is a member of the set content.

After the declaration of the prev and the while loop the observe a noteThat specification construct, which is a sequential composition of an assert statement followed by an assume statement [20, Chapter 3]. The intention of these noteThat statements at this position is to propagate facts proved by MONA to be used as assumptions to the used by the SMT-solvers. These noteThat statements assert and assume by using field constraint analysis and MONA, the following invariants. If a node has a left child then the child can reach the parent in the sorted list. If a node has a right child then this node can reach its right child in the sorted list and root is reachable from first via next.

The given loop invariant is by convenience grouped in eight groups of conjuncts labeled with the comment construct.

The first group states that pred, p and n are contained in the set contentand that next of null is null. The second group specifies the relation between root and the variables p, n, wentLeft, in particular specifies the absence of a successor node when the parent node is reachable by following the only the right field starting at the root. The third group specifies the following invariants related to the reference first. The root node is always reachable from first and the value of first is less or equal than the value of the current node. Furthermore, if the value of a node is less than the value of first, then this node is not in the set content. The fourth group, specifies invariants with respect to the parent node and its child nodes according to the wentLeft variable. The fifth group, specifies the possible values of pred an succ. The sixth group, states invariants about pred and succ. In particular, it specifies that the node e is always in between pred and succ and if this is not the case then the node e is not in the set content. The seventh group, states invariants with respect to the current node n, specifying its reachability from the root and its impact on pred and succ. The last group, consisting of universally quantified invariants. The first three invariants, restate some sortedness and reachability invariants. The next three invariants, specify the cases in which the node will not be a part of the sorted list formed by next, i.e., the cases when the value of the node e is greater than the value of the last node of the list or if the value of the node e is in between the values of two nodes directly connected through the next field.

The postcondition states that the returned value is equal to evaluating the membership of e to the content. After the while loop, the node n is compared with the node e, if they are equal then method returns true, which follows from the fact that n is always a member of the set content. In the case when they are not equal, since we have a sorted list and e is always in between pred and succ, we need to show that e is not in the path from pred to succ and that this path is closed, this is implied by the three noteThat statements, listed in the else case.

```
public static boolean member(Node e)
/*: requires "e ≠ null"
    ensures "  result  = (e ∈ content) "     */
{
    Node p = null;        Node n = root;
    Node pred = null;  Node succ = null;
    boolean wentLeft = false;
    //: noteThat "∀ x. x..left  ∈ content ∧ x..left ≠ null → next_reach (x..left) x";
    //: noteThat "∀ x. x ∈ content → next_reach x (x..right)";
    //: noteThat "( first , root) ∈ {(x,y). x .. next = y}*";
    while /*: inv "
            comment ''group 1'' (p ∈ content ∧ n ∈ content ∧ pred ∈ content ∧
            succ ∈ content ∧ (null..next = null)) ∧

            comment ''group 2'' ((root = null → n = null) ∧
            ((root, p) ∈ {(x1,x2). x1.. right = x2}* ∧ ¬wentLeft → succ = null) ∧
            (root = n ∧ n ≠ e → root ≠ e) ∧
            (root ≠n → e ≠ root ) ∧
            (root ≠ null ∧ root..Node.right = null → root..Node.next = null) ∧
            (root ≠ null ∧ root..Node.right ≠ null → root..Node.next ≠ null)) ∧

            comment ''group 3'' (( first , root) ∈ {(x,y). x .. next = y}* ∧
            ( first ≠ null ∧ n ≠ null → first..Node.v ≤ n..Node.v) ∧
            ( first .. Node.v ≥ e..Node.v → e ∉ content )) ∧

            comment ''group 4'' (( p ≠ null → p ≠ e ) ∧
            ((p = null ∧ n = null) → root = null) ∧
            (p ≠ null ∧ wentLeft → p..Node.left = n) ∧
            (p ≠ null ∧ ¬wentLeft → p..Node.right = n) ∧
            (p = null → n = root ) ∧
            (p ≠ null → (wentLeft ↔ succ = p) ) ∧
            (p ≠ null →
               (( p..v < n..v → next_reach p n ) ∨ ( p..v ≥ n..v → next_reach n p) ) )) ∧

            comment ''group 5'' ((¬wentLeft ↔ pred = p) ∧
            (wentLeft → e..Node.v < p..Node.v ) ∧
            (¬wentLeft ∧ p ≠ null → p..Node.v < e..Node.v ) ∧
            (wentLeft ∧ n ≠ null → next_reach n p ) ∧
            (wentLeft ∧ n ≠ null ∧ p ≠ null → ¬(next_reach p n) ) ∧
            (wentLeft ∧ n..Node.left = null ∧ pred ≠ null ∧ n ≠ null
               → pred..Node.next = n ) ∧
            (¬wentLeft ∧ n..Node.right = null ∧ succ ≠ null ∧ n ≠ null
               → n..Node.next = succ )) ∧

            comment ''group 6'' ((pred = null → (root, p) ∈ {(x1,x2). x1..left = x2}*) ∧
            (succ = null → (root, p) ∈ {(x1,x2). x1..right = x2}*) ∧
            (pred = null → (root, n) ∈ {(x1,x2). x1..left = x2}*) ∧
            (succ = null → (root, n) ∈ {(x1,x2). x1..right = x2}*) ∧
            (pred ≠ null → (pred..right,n) ∈ {(x1,x2). x1.. left = x2}*) ∧
            (succ ≠ null → (succ..left,n) ∈ {(x1,x2). x1.. right = x2}*) ∧
```

Figure 4.12: Threaded tree member method

```
/*: "
      (pred ≠ null ∧ p ≠ pred
           → (pred..right,p) ∈ {(x1,x2). x1.. left  = x2}*) ∧
      (succ ≠ null ∧ p ≠ succ → (succ..left,p) ∈ {(x1,x2). x1..right = x2}*) ∧
      (pred ≠ null ∧ n ≠ null ∧ n..Node.left = null → pred..Node.next = n ) ∧
      (succ ≠ null ∧ ¬wentLeft ∧ n..Node.right = null ∧ n ≠ null → n..Node.next = succ ) ∧
      ( pred ≠ null ∨ succ ≠ null → pred ≠ succ ) ∧
      (pred ≠ null → smt_reach pred n) ∧
      (pred ≠ null → smt_reach pred succ) ∧
      (succ ≠ null ∧ n ≠ null → smt_reach n succ ) ∧
      (pred ≠ null ∧ e..v < pred..v → e ∉ content ) ∧
      (succ ≠ null ∧ e..v ≥ succ..v → e ∉ content ) ∧
      (pred ≠ null → pred..v < e..v) ∧
      (succ ≠ null → e..v < succ..v)) ∧

      comment ''group 7'' ((n ≠ null → (smt_reach first e ∨ e ∉ content)) ∧
      (n ≠ null ∧ (root, n) ∈ {(x1,x2). x1..right  = x2}* → succ = null) ∧
      (n ≠ root → ( n = p..Node.left ∨ n = p..Node.right ) ) ∧
      (n ≠ null ∧ (root, n) ∈ {(x1,x2). x1.. left  = x2}* → pred = null) ∧
      (n ≠ null ∧ (root, n) ∈ {(x1,x2). x1..right  = x2}* → succ = null)) ∧

      comment ''group 8'' ((∀ x. x.. left  ∈ content ∧ x..left ≠ null → next_reach (x..left) x) ∧
      (∀ x. x ∈ content → next_reach x (x..right)) ∧
      (∀ x y. next_reach x y ∧ x ≠ null ∧ y ≠ null ∧ x ≠ y → x..v < y..v) ∧
      (∀ x y. ( x ∈ content ∧ x..next = y ∧ x..v < e..Node.v ∧
           y..v > e..Node.v ∧ y ≠ null ∧ x ≠null ) → e ∉ content) ∧
      (∀ x. x ∈ content ∧ x..next = null ∧ x ≠ null ∧ x..v < e..v → e ∉ content))"; */

          ( (n != null) && (n != e) ) {
          wentLeft = (e.v < n.v);
          if (wentLeft) {
              succ = n; p = n; n = n.left;
          } else {
              pred = n; p = n; n = n.right;
          }
      }
      //: noteThat "∀ x. x.. left  ∈ content ∧ x..left ≠ null → next_reach (x..left) x";
      //: noteThat "∀ x. x ∈ content → next_reach x (x..right)";
      if (n == e) {
          return true;
      }
      else {
          //: noteThat " wentLeft ∧ p ≠ null → p = succ ∧ (pred ≠ null → pred..next = p)";
          //: noteThat " ¬wentLeft ∧ p ≠ null → p = pred ∧ (succ ≠ null → p..next = succ)";
          //: noteThat "pred ≠ null → pred..next = succ";
          return false;
      }
  }
```

Figure 4.13: Threaded tree member method

4.6 Summary

Table 4.1 presents the results of checking the postcondition of the previously presented methods and some additional methods. The experiments where conducted on a 1.6GHz Centrino duo with 3.2GB machine running Linux 2.6.27-9.

The last two rows show the results of combining LISBQ over the SMT-solver [13] Z3 with MONA. Though, they proved different sequents, we observe a substantial difference on the running times.

Structure	Method	Total seq.	Prover seq.	Time sec.
Single linked list	filter	26	15(Z3)	3.2
	insert	27	12(Z3)	1.7
	findPrev	12	5(Z3)	0.4
	contains	9	4(Z3)	0.2
	remove	30	9(Z3)	2.6
	removeLast	22	5(Z3)	0.5
	getLast	11	4(Z3)	0.2
Cyclic list	init	4	3(Z3)	0.3
	add	8	4(Z3)	2.5
	member	24	13(Z3)	1.8
Sorted list	add	38	21(Z3)	13.2
	remove	57	26(Z3)	10.1
Association list	init	3	2(SPASS)	0.2
	isEmpty	3	1(SPASS)	0.1
	define	14	5(SPASS)	0.6
	lookup	15	5(SPASS)	0.5
	add	5	3(SPASS)	21.6
	remove	28	19(SPASS)	77.4
Threaded tree	add	160	52(Z3) 34(MONA)	15.5(Z3) 1260.0(MONA)
	member	230	40(Z3) 26(MONA)	246.7(Z3) 238,3(MONA)

Table 4.1: Verified data structures and their corresponding methods

Chapter 5

Conclusion

In this thesis we use the *Logic of Interpreted Sets and Bounded Quantification* (LISBQ) in the context of the Jahob system to ensure data structure consistency of list like data structures. An important characteristic of LISBQ is that it is complete for reasoning about reachability in list like data structures, including cycles and sharing. On the other side it is restricted to lists, i.e. no possible to reason about reachability in trees.

Our implementation of LISBQ in the context of the Jahob system enables us to use LISBQ both over SMT-solvers and First-Order resolution based provers. This capability has proven to be useful, particularly in the cases where SMT-solvers were not able to prove some formula, whereas First-Order resolution based provers could, and vice versa. Using LISBQ over First-Order resolution based provers was particularly useful on the presence of quantifiers. However, in general, the performance of LISBQ over SMT-solvers was superior.

In our evaluation, we verified a wide range of data structures. During this process we could observe that in some cases our implementation led to much simpler proofs. In other cases, we were able to verify data structures that were beyond the scope of previous approaches in Jahob. Finally, we were able to combine LISBQ with Monadic Second Order Logic (MSOL) over trees to verify data structures that were beyond the scope of LISBQ and MSOL when used independently.

Zusammenfassung

Software ist allgegenwärtig in der heutigen Technologie. Es ist daher erwünscht die Zuverläsigkeit von Software zu garantieren. Ein Weg dieses Ziel zu erreichen, liegt in der formalen Verifikation von Software. Ein Klasse von Programme, die eine besondere Herausforderung für Methoden der formalen Verifikation darstellt, sind Programme, die dynamisch allozierten Zeigestrukturen wie zum Beispiel einzelverkettete Liste manipulieren. Wir nennen solche Programme "Zeigerprogramme". Die Korrektheit von Zeigerprogrammen hängt in der Regel von der Konsistenz der manipulierten Zeigerstrukturen ab. Solche Konsistenzeingenschaften beschreiben die Form einer Datenstruktur. Beispiele solcher Eigenschaften sind "eine Liste ist azyklisch" und "die Elemente einer Liste sind geordnet". Eine ausdrucksstarke und zugleich entscheidbare Logik, die es ermoeglicht solche Eigenschaften von Zeigerprogrammen mathematisch zu beweisen, ist Lahiris und Qadeers *Logik über interpretierte Mengen mit beschränkter Quantifizierung* (engl. LISBQ). Diese Logik erlaubt es, komplexe Eigenschaften über listenähnliche Zeigerstrukturen auszudrücken.

In dieser Masterarbeit untersuchen wir die Zweckmäßigkeit der Logik LISBQ für die Verifikation von Zeigerprogrammen im Rahmen des Verifikationssystem Jahob. Wir beschreiben eine Einbettung von LISBQ in Logik höherer Stufe (HOL). Jahob verwendet eine Teilmenge von HOL, um Beweisverpflichtungen auszudrücken, die bei der Programmverifikation generiert werden. Wir beschreiben ferner eine Übersetzung solcher Beweisverpflichtungen in das LISBQ-Fragment von HOL. Schliesslich, skizzieren wir die Implementierung einer Entscheidungsprozedur für LISBQ im Jahob-System.

Wir haben unsere Implementierung verwendet, um Eigenschaften von Operationen auf einer Vielzahl verschiedener Zeigerstrukturen zu verifizieren. Insbesondere haben wir Operationen auf einfach verketteten Listen, geordneten Listen, zyklische Listen, und Assoziationslisten betrachtet. Unsere Evaluation zeigt zwei wesentliche Vorteile, die die Verwendung von LISBQ mit sich bringt. Zum einen ermöglicht es LISBQ, die Menge der verifizierbaren Datenstrukturen zu erhöhen. Zum anderen konnten wir im Vergleich zu früheren Verifikationsversuchen solcher Zeigerprogramme in Jahob, die nicht auf der Verwendung von LISBQ beruhten, sowohl Anzahl als auch Komplexität der nötigen Programmanotationen (wie zum Beispiel Schleifeninvarianten) reduzieren. Außerdem gelang es uns die Entscheidungsprozedur für LISBQ mit anderen Entscheidungen Prozeduren in Jahob zu kombinieren, um Eigenschaften geketteter Binärbäume mit ganzzahligen Schlüsseln zu verifizieren. Keine der individuellen Entscheidungsprozeduren in Jahob, einschliesslich der für LISBQ, wäre für sich genommen in der Lage diese Eigenschaften automatisch zu verifizieren.

Bibliography

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof (Applied Logic Series)*. Springer, 2nd edition, July 2002.

[2] R.-J. Back, A. Akademi, and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[3] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01*, volume 2031 of *LNCS*, pages 268–283. Springer-Verlag, 2001.

[4] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In J.-P. Katoen and P. Stevens, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'02*, volume 2280 of *LNCS*, pages 158–172. Springer-Verlag, 2002.

[5] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[6] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[7] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification, CAV'07*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, 2007.

[8] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. C. Rinard. Using First-Order Theorem Provers in the Jahob Data Structure Verification System. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'07*, volume 4349 of *LNCS*, pages 74–88. Springer-Verlag, 2007.

[9] A. R. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, Berlin, 1 edition, September 2007.

[10] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundam. Inf.*, 89(4):369–392, 2009.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'77*, pages 238–252. ACM Press, 1977.

[12] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, New York, United States, 1984.

[13] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation, PLDI'02*, pages 234–245. ACM Press, 2002.

[15] R. J. Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[16] N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.

[17] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The Boundary Between Decidability and Undecidability for Transitive-Closure Logics. In *Computer Science Logic, CSL'04*, volume 3210 of *LNCS*, pages 160–174. Springer-Verlag, 2004.

[18] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[19] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *International Conference on Implementation and Application of Automata'00*, volume 2088 of *LNCS*, pages 182–194. Springer-Verlag, 2000.

[20] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[21] V. Kuncak. The Jahob project web page. `http://lara.epfl.ch/dokuwiki/doku.php?id=jahob_system`, 2008.

[22] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, 2006.

[23] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'08*, pages 171–182. ACM Press, 2008.

[24] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conference on Automated Deduction, CADE-20*, volume 3632 of *LNCS*, pages 99–115. Springer-Verlag, 2005.

[25] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *ACM Conference on Programming Language Design and Implementation, PLDI'01*, pages 221–231. ACM Press, 2001.

[26] G. Nelson. Verifying reachability invariants of linked structures. In *Annual ACM Symposium on the Principles of Programming Languages, POPL'83*, pages 38–47. ACM Press, 1983.

[27] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[28] A. Pnueli. The temporal logic of programs. In *Annual IEEE Symposium on Foundations of Computer Science, FOCS'77*, pages 46–57. IEEE Computer Society, 1977.

[29] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *Computer Aided Verification, CAV'08*, volume 5123 of *LNCS*, pages 314–327. Springer-Verlag, 2008.

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

[31] M. I. Schwartzbach. Lecture notes on static analysis. 2008.

[32] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[33] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.

[34] T. Wies. *Symbolic Shape Analysis*. PhD thesis, Albert-Ludwigs University of Freiburg, February 2009.

[35] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field Constraint Analysis. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'06*, volume 3855 of *LNCS*, pages 157–173. Springer-Verlag, 2006.

[36] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification for Linked Data Structures. In *ACM Conference on Programming Language Design and Implementation, PLDI'08*. ACM Press, 2008.