Logical Characterizations of Heap Abstractions

Greta Yorsh[1]
School of Computer Science, Tel-Aviv University, Israel

April 12, 2003

[1] gretay@post.tau.ac.il

Acknowledgements

Abstract

Shape analysis concerns the problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. In recent work, it has been shown how shape analysis can be performed, using an abstract interpretation based on 3-valued first-order logic. In that work, concrete stores are finite 2-valued logical structures, and the sets of stores that can possibly arise during execution are represented (conservatively) using a certain family of finite 3-valued logical structures.

This thesis presents results—both negative and positive—about the expressive power of 3-valued logical structures. It also defines a non-standard ("supervaluational") semantics for 3-valued first-order logic that is more precise than a conventional 3-valued semantics. The material presented here is an extended version of [YRSW03].

In addition, this thesis address a more practical aspect of program analysis, namely, dealing with real applications coded in C. It presents C-Simplifier, a tool that translates a C program into an equivalent C program with a simple syntax, called *CoreC*. This tool enables faster development of source-code analyzers; it was used by CSSV [DRS03] to check real C programs for string errors; it can also be used to perform shape analysis on real programs.

Contents

Chapter 1

Introduction

1.1 Background

Abstraction and abstract interpretation [CC77] are key tools for verifying properties of systems, both for hardware systems [CGL94, Dam96] and software systems [NNH99]. In abstract interpretation, possibly infinite sets of concrete stores are represented in a conservative manner by a finite sets of abstract values. Each transition of the system is given an interpretation over abstract values that is conservative with respect to its interpretation over corresponding sets of concrete stores; that is, the result of "executing" a transition must be an abstract value that describes a superset of the concrete stores that actually arise.

The result of applying an abstract interpretation technique to a program is a set of abstract values at each program point. This involves finding the least fixed point of a certain set of equations. When the fixed point is reached, the abstract values that have been collected at program point P describe a superset of all the concrete stores that can occur at P. This methodology guarantees that the results of abstract interpretation overapproximate the sets of concrete stores that actually arise at each point in the system. To determine whether a property always holds at P, one checks whether it holds in all of the abstract values that were collected there.

One of the most challenging problems in abstract interpretation is *shape analysis*. Shape analysis concerns the problem of finding "shape descriptors" that characterize the shapes of the heap-allocated data structures that the program manipulates. Shape analysis generally deals with programs written in languages like C, C++, and Java, which allow (i) dynamic allocation and deallocation of cells from the heap, (ii) destructive updating of structure fields, and, in the case of Java, (iii) dynamic creation and destruction of threads. This combination of features creates considerable difficulties for any abstract-interpretation method; in particular,

- Dynamic storage allocation and dynamic thread creation mean that there is no *a priori* upper bound on either the size of a program's data structures or the number of threads that arise in the system at execution time.

- Destructive updating of structure fields permits a program to create memory configurations that exhibit complicated aliasing relationships.

5

Although the results of shape analysis are important for many purposes—verification, program understanding, anomaly detection, debugging, compile-time garbage collection, instruction scheduling, code optimization, and parallelization—obtaining useful information about linked data structures that can be destructively updated is generally very difficult [JM81, CWZ90, SRW02].

These difficulties are addressed in [SRW02] by creating a shape abstraction method, called *canonical abstraction*. In this method, concrete stores are represented by finite 2-valued logical structures—i.e., a collection of relations. The sets of stores that can possibly arise during execution are represented (conservatively) using a certain family of finite 3-valued logical structures. In fact, this approach can be used not only for shape analysis, but also for any abstract-interpretation problem in which concrete states can be represented by a *logical structure*.

The principle behind canonical abstraction is that concrete individuals (e.g., heap cells) that have the same vector of values for a distinguished collection of unary relations are summarized to the same abstract individual; other relations are collapsed accordingly. Canonical abstraction ensures that abstract structures have an *a priori* bounded size, which guarantees that a fixed-point is always reached.[1]

However, the constraint of working with limited-size descriptors implies a loss of information about the store. Intuitively, some concrete individuals "lose their identity" when they are grouped together with other individuals in one summary individual. Moreover, a property can be true for some concrete individuals of the group but false for other individuals. It is for this reason that 3-valued logic is used; uncertainty about a property's value is captured by means of the third truth value, $1/2$.

1.2 Motivation

One issue that arises when abstraction is employed concerns the *expressive power* of the abstraction method: "What collections of concrete states are expressible using the given abstraction method?" A second issue that arises when abstraction is employed is how to *extract information* from an abstract value. For instance, this is a fundamental problem for clients of abstract interpretation, such as verification tools, program optimizers, program-understanding tools, etc., which need to be able to interpret what an abstract value means. An abstract value a represents a set of concrete stores X; ideally, a query φ should return an answer that summarizes the result of posing φ against each concrete store $S \in X$:

- If φ is true for each S, the summary answer should be "true".

- If φ is false for each S, the summary answer should be "false".

- If φ is true for some $S \in X$ but false for some $S' \in X$, the summary answer should be "unknown".

For instance, if a program optimizer poses the query "Does program condition `x == NULL` evaluate to `true` in all stores that arise at program point `p`?" and the answer is "true", then it has sufficient information to make the simplification

$$\text{p: if (x == NULL) then } S_1 \text{ else } S_2 \text{ fi} \implies \text{p: } S_1.$$

---

[1]An alternative would be to define widening operators that guarantee termination [CC79].

This thesis presents results on both of these questions, for a class of abstractions defined in [SRW02].

Because the notion of abstraction used in [SRW02] is based on logical structures, our results are actually much more broadly applicable than shape-analysis problems: they apply to any abstraction in which concrete states of a system are represented by finite 2-value logical structure and abstraction is performed via either of the mechanisms described in Chapter 2 and Chapter 7.[2]

1.3 Main Results

The thesis investigates the expressive power of finite 3-valued structures by giving a logical characterization of their expressive power; that is, we examine the question

> For a given 3-valued structure S, under what circumstances is it possible to create a formula $\widehat{\gamma}(S)$, such that $S^{\natural}$ satisfies $\widehat{\gamma}(S)$, exactly when $S^{\natural}$ is a 2-valued structure that S represents? I.e.,
> $$S^{\natural} \models \widehat{\gamma}(S) \text{ iff } S \text{ represents } S^{\natural}.$$

The thesis presents three results concerning this question:

- It is not possible to give a $\widehat{\gamma}(S)$ in general, if $\widehat{\gamma}(S)$ is to be written in first-order logic with transitive closure.

- However, it is always possible to give a $\widehat{\gamma}(S)$ written in first-order logic with transitive closure for a well-defined class of 3-valued structures. (It is exactly the class of 3-valued structures that has been shown to be useful for shape analysis [SRW02].)

- Moreover, it is always possible to give a $\widehat{\gamma}(S)$ in general, using a more powerful formalism, namely, monadic second-order formulas, which is a subset of NP formulas [Fag75].

The thesis then uses the results on $\widehat{\gamma}(S)$ to address the problem of reading out information from a 3-valued structure in the most-precise way possible. That is, we give a nonstandard way to check if a formula φ holds in a 3-valued structure S:

- If $\widehat{\gamma}(S) \Rightarrow \varphi$ is valid, i.e., holds in all 2-valued structures, we know that φ evaluates to 1 in all of the 2-valued structures represented by S.

- If $\widehat{\gamma}(S) \Rightarrow \neg\varphi$ is valid, we know that φ evaluates to 0 in all of the 2-valued structures represented by S.

- Otherwise, we know that φ evaluates to 1 in some 2-valued structures represented by S, and evaluates to 0 in other 2-valued structures represented by S.

---

[2]Throughout the thesis, however, we do use shape-analysis examples to illustrate the concepts discussed. The experiments reported on in Chapter 6 used logical structures that arose in actual program-analysis runs of the TVLA system (**T**hree-**V**alued-**L**ogic **A**nalyzer) [LAS00], which is an implementation of the program-analysis method described in [SRW02].

In particular, whenever the above-mentioned method returns $1/2$, any sound method for extracting information from S must also return $1/2$. This is in contrast to the techniques used in [SRW02] and in TVLA, which can return $1/2$ even when all the concrete structures represented by S have the value 1 (or all have the value 0).

Although the validity question is undecidable both for first-order logic with transitive closure, and for NP formulas, several theorem provers for first-order logic have been created. We report on two experiments in which we used these tools to implement symbolic procedures for extracting information from a 3-valued structure in the most-precise way possible.

1.4 Outline of the Thesis

The thesis is organized as follows: Chapter 2 defines our terminology, and explains the use of 3-valued structures as abstractions of 2-valued structures. Chapter 3 and Chapter 4 present the results on the expressive power of 3-valued structures. Chapter 5 discusses the problem of reading out information from a 3-valued structure in the most-precise way possible. Chapter 6 describes two experiments in which we used the $\widehat{\gamma}$ operation and an existing theorem prover for first-order logic to read out information from 3-valued structures. Chapter 7 defines an alternative abstract domain for shape analysis, based on canonical abstraction, and the $\widehat{\gamma}$ operation for that domain. Chapter 8 discusses related work. Chapter 9 makes some final remarks. Omitted proofs appear in Appendix B. Finally, Appendix C presents the design of *CoreC*.

Chapter 2

Preliminaries

Section 2.1 defines the syntax and standard Tarskian semantics of first-order logic with transitive closure and equality. Section 2.2 introduces *integrity formulas*, which exclude structures that do not represent a potential store and specify which structures are actually of interest. Section 2.3 introduces 3-valued logical structures, which extend ordinary logical structures with an extra value, $1/2$, representing "unknown" values that arise when several concrete elements are represented by a single abstract element. Section 2.4 defines an ordering on 3-valued structures.

 Fig. 2.1(a) shows the declaration of a linked-list data type in C, and Fig. 2.1(b) shows a C program that searches a list and splices a new element into the list. This program will be used as a running example throughout this thesis.

2.1 Syntax and Semantics of First-Order Formulas with Transitive Closure

We represent concrete stores by ordinary 2-valued logical structures over a fixed finite set of predicate symbols $\mathcal{P} = \{eq, p_1, \ldots, p_n\}$, where eq is a designated binary predicate, denoting equality of individuals. $maxR$ denotes the maximal arity of the predicates in $\mathcal{P}$. Without loss of generality we exclude constant and function symbols from the logic.[1]

Example 2.1.1 Table 2.1 lists the set of predicates used in the running example. The unary predicates x, y, t, and e correspond to the program variables x, y, t, and e, respectively. The binary predicate n corresponds to the n fields of List elements. The unary predicate is ("is shared") captures "heap sharing", i.e., List elements pointed to by more than one field. (It was introduced in [CWZ90] and also used in [SRW98] to capture list and tree data structures.) The unary predicates r_x, r_y, r_t, and r_e hold for heap nodes reachable from the program variables x, y, t, and e, respectively. A heap node u is said to be *reachable* from a program variable if the variable points to a heap node u', and it is possible to go from u' to u by following zero or more n-links. In practice, we define reachability using reflexive transitive closure of the predicate n.

 The notion of reachability plays a crucial role in defining abstractions that are useful for proving program properties in practice. For instance, it may have the effect of preventing disjoint lists from

---

[1]Constant symbols can be encoded via unary predicates, and n-ary functions via $n + 1$-ary predicates.

```
                              /* insert.c */
                              #include ``list.h''
                              void insert(List x, int d) {
                                List y, t, e;
                                assert(acyclic_list(x) &&
                                   x != NULL);
   /* list.h */              y = x;
   typedef struct node {     while (y->n != NULL && ...) {
     struct node *n;           y = y->n;
     int data;               }
   } *List;                  t = malloc();
                              t->data = d;
                              e = y->n;
                              t->n = e;
                              y->n = t;
                            }
         (a)                              (b)
```

Figure 2.1: (a) Declaration of a linked-list data type in C. (b) A C function that searches a list pointed to by parameter x, and splices in a new element.

being collapsed in the abstract representation. This may significantly improve the precision of the answers obtained by a program analysis.

We define first-order formulas inductively over the **vocabulary** $\mathcal{P}$ using logical connectives $\vee$, $\neg$, quantifier $\exists$, and 'TC' operator in the standard way:

$$\varphi ::= 0 \mid 1 \mid p(v_1, \ldots, v_k) \mid (\neg \varphi_1) \mid (\varphi_1 \vee \varphi_2)$$
$$\mid (\exists v_1 : \varphi_1) \mid (TC\ v_1, v_2 : \varphi_1)(v_3, v_4) \tag{2.1}$$
$$\text{where } p \in \mathcal{P}; v_i \text{ are variables}; \varphi, \varphi_i \text{ are formulas}$$

The operator 'TC' denotes transitive closure. If φ_1 is a formula with free variables V, then $(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$ is a formula with free variables $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$. The set of free variables of other formulas is defined as usual. A formula is **closed** when it has no free variables.

We use several shorthand notations: $\varphi_1 \Rightarrow \varphi_2 \stackrel{\text{def}}{=} (\neg \varphi_1 \vee \varphi_2)$; $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg \varphi_1 \vee \neg \varphi_2)$; $\varphi_1 \Leftrightarrow \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$; and $\forall v : \varphi \stackrel{\text{def}}{=} \neg \exists v : \neg \varphi$. The transitive closure of a binary predicate p is $p^+(v_3, v_4) \stackrel{\text{def}}{=} (TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)$. The *reflexive* transitive closure of a binary predicate p is $p^*(v_3, v_4) \stackrel{\text{def}}{=} ((TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4)) \vee eq(v_3, v_4)$. The order of precedence among the connectives, from highest to lowest, is as follows: $\neg, \wedge, \vee,$ 'TC', $\forall$, and $\exists$. We drop parentheses wherever possible, except for emphasis.

Definition 2.1.2 (**2-valued Logical Structures**) *let $\mathcal{P}_i$ denote the set of predicate symbols with arity i. A **logical structure over** $\mathcal{P}$ is a pair $S = \langle U, \iota \rangle$ in which*

10

Predicate	Intended Meaning
$eq(v_1, v_2)$	Do v_1 and v_2 denote the same heap node?
$q(v)$	Does pointer variable q point to node v?
$n(v_1, v_2)$	Does the n field of v_1 point to v_2?
$is(v)$	Is v pointed to by more than one field ?
$r_q(v)$	Is the node v reachable from q ?

Table 2.1: The set of predicates for representing the stores manipulated by programs that use the `List` data-type from Fig. 2.1(a). q denotes an arbitrary predicate in the set $PVar$, which contains a predicate for each program variable of type `List`. In the case of `insert`, $PVar = \{x, y, t, e\}$.

- U is a set of individuals.

- ι is the interpretation of predicate symbols, i.e., for every predicate symbol $p \in \mathcal{P}_i$, $\iota(p): U^i \to \{0, 1\}$ determines the tuples for which p holds. Also, $\iota(eq)$ is the interpretation of equality, i.e., $\iota(eq)(u_1, u_2) = 1$ iff $u_1 = u_2$.

Below we define standard Tarskian semantics for first-order logic.

Definition 2.1.3 (Semantics of First-Order Logical Formulas) *Consider a logical structure $S = \langle U, \iota \rangle$. An* **assignment** *$Z$ is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z: \{v_1, v_2, \ldots\} \to U$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. In the sequel, we assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ. We say that S and Z* **satisfy** *a formula φ (denoted by $S, Z \models \varphi$) when one of the following holds:*

- *$\varphi \equiv 1$*

- *$\varphi \equiv p(v_1, v_2, \ldots, v_i)$ and $\iota(p)(Z(v_1), Z(v_2), \ldots, Z(v_i)) = 1$.*

- *$\varphi \equiv \neg\varphi_0$ and $S, Z \models \varphi_0$ does not hold.*

- *$\varphi \equiv \varphi_1 \vee \varphi_2$, and either $S, Z \models \varphi_1$ or $S, Z \models \varphi_2$.*

- *$\varphi \equiv \exists v_1 : \varphi_1$ and there exists an individual $u \in U$, such that $S, Z[v_1 \mapsto u] \models \varphi_1$.*

- *$\varphi \equiv (TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$ and there exists $u_1, u_2, \ldots, u_m \in U$ such that $Z(v_3) = u_1$, $Z(v_4) = u_m$ and for all $1 \leq i < m$, $S, Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi_1$.*

For a closed formula φ, we will omit the assignment in the satisfaction relation, and merely write $S \models \varphi$.

2.2   Integrity Constraints

Because not all logical structures represent stores, we use a designated closed formula F, called *the integrity formula,*[2] to exclude impossible stores. This allows us to restrict the set of structures to the ones satisfying F.

Definition 2.2.1 *A structure S is **admissible** if $S \models F$.*

In the rest of the thesis, we assume that we work with a fixed integrity formula F. All our notations are parameterized by $\mathcal{P}$ and F.

Example 2.2.2 For the `List` data type, there are four conditions that define the admissible stores:

(a) Each program variable can point to at most one heap node.

(b) The `n` field of a heap node can point to at most one heap node.

(c) Predicate is ("is shared") holds for exactly those nodes that have two or more predecessors.

(d) The reachability predicate for each variable `q` holds for exactly those nodes that are reachable from program variable `q`.

Thus, the integrity formula F_{List} for the `List` data-type is:

$$
\begin{array}{rll}
& \wedge_{p \in PVar} \forall v_1, v_2 : p(v_1) \wedge p(v_2) \Rightarrow eq(v_1, v_2) & (a) \\
\wedge & \forall v, v_1, v_2 : n(v, v_1) \wedge n(v, v_2) \Rightarrow eq(v_1, v_2) & (b) \\
\wedge & \forall v : is(v) \iff \exists v_1, v_2 : \neg eq(v_1, v_2) & \\
& \qquad\qquad\qquad\qquad \wedge n(v_1, v) \wedge n(v_2, v) & (c) \\
\wedge & \wedge_{q \in PVar} \forall v : r_q(v) \iff \exists v_1 : q(v_1) \wedge n^*(v_1, v) & (d)
\end{array}
$$

2.3 3-Valued Logical Structures

In this section, we define 3-valued logical structures, which provide a way to represent a set of 2-valued logical structures in a compact and conservative way.

We say that the values 0 and 1 are *definite values* and that $1/2$ is an *indefinite value*, and define a partial order $\sqsubseteq$ on truth values to reflect information content. $l_1 \sqsubseteq l_2$ denotes that l_1 possibly has more definite information than l_2:

Definition 2.3.1 [Information Order]. *For $l_1, l_2 \in \{0, 1/2, 1\}$, we define the **information order** on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$. The symbol $\sqcup$ denotes the least-upper-bound operation with respect to $\sqsubseteq$.*

Definition 2.3.2 *A **3-valued logical structure** over $\mathcal{P}$ is the generalization of 2-valued structures given in Definition 2.1.2, in that predicates may have the value $1/2$. This means that $S = \langle U, \iota \rangle$ where for $p \in \mathcal{P}_i$,*

$$\iota(p) \colon (U^S)^i \to \{0, 1, 1/2\}.$$

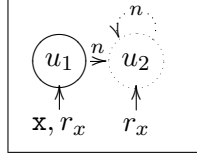[2]In [SRW02] these are called "hygiene conditions".

Figure 2.2: A 3-valued structure that represents possible inputs of the `insert` program. It represents all lists that are pointed to by program variable x and that have at least two elements.

In addition, (i) for all $u \in U^S$, $\iota^S(eq)(u,u) \sqsupseteq 1$, and (ii) for all $u_1, u_2 \in U^S$ such that u_1 and u_2 are distinct individuals, $\iota^S(eq)(u_1, u_2) = 0$.

*An individual $u \in U$ having $\iota(eq)(u,u) = 1/2$ is called a **summary individual**. As we shall see, such an individual may represent more than one individual from a given 2-valued structure.*

We denote the set of 2-valued logical structures by 2-STRUCT[$\mathcal{P}$]. The set of 3-valued logical structures is denoted by 3-STRUCT[$\mathcal{P}$].

Example 2.3.3 Fig. 2.2 shows a 3-valued structure that represents possible inputs of the `insert` program. As we will see, this structure represents all lists that are pointed to by program variable x and that have at least two elements.

A 3-valued structure can be depicted as a directed graph, with individuals as graph nodes. A unary predicate p is represented in the graph by having an arrow from the predicate name p to node u for each individual u for which p holds. An arrow between two nodes indicates whether a binary predicate holds for the corresponding pair of individuals. An indefinite value of a predicate is shown by a dotted arrow; the value 1 is shown by a solid arrow; and the value 0 is shown by the absence of an arrow.

In this example, the structure has 2 individuals, u_1 and u_2, where u_1 is the head of the list pointed to by x, and u_2 is a summary node (drawn as a dotted circle), which represents the tail of the list. Predicate r_x holds for u_1 and u_2, indicating that all nodes are reachable from x. Other unary predicates are not shown, indicating that their values are 0 for all nodes, i.e., the program variables y, e, and t are NULL, and there is no sharing in the list. The dotted edge from u_1 to u_2 indicates that there may be n-links from the head of the list to some elements in the tail. In fact, the (u_1, u_2)-edge represents exactly one n-link that points to exactly one list element, because of the integrity rules in Example 2.2.2. In contrast, the dotted self-loop on u_2 represents all n-links that may occur in the tail.

2.4 Embedding Order

We define the *embedding ordering* on structures as follows:

Definition 2.4.1 *Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two logical structures, and let $f \colon U^S \to U^{S'}$ be a surjective function. We say that f **embeds** S in S' (denoted by $S \sqsubseteq^f S'$) if for every predicate symbol $p \in \mathcal{P}_i$ and all $u_1, \ldots, u_i \in U^S$,*

$$\iota^S(p)(u_1, \ldots, u_i) \sqsubseteq \iota^{S'}(p)(f(u_1), \ldots, f(u_i)) \tag{2.2}$$

13

*We say that S **can be embedded in** S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.*

Concretization of 3-Valued Structures. Embedding allows us to define the (potentially infinite) set of concrete structures that a set of 3-valued structures represents:

Definition 2.4.2 (Concretization of 3-Valued Structures) *For a set of structures $X \subseteq$ 3-STRUCT$[\mathcal{P}]$, we denote by $\gamma(X)$ the set of 2-valued structures that X represents, i.e.,*

$$\gamma(X) = \{S^{\natural} \in \text{2-STRUCT}[\mathcal{P}] \mid \text{exists } S \in X \text{ such that } S^{\natural} \sqsubseteq S \text{ and } S^{\natural} \models F\} \tag{2.3}$$

Also, for a singleton set $X = \{S\}$ we write $\gamma(S)$ instead of $\gamma(X)$.

Chapter 3

Characterizing 3-Valued Structures by First-Order Formulas

This chapter and the next one present our results on the expressive power of 3-valued structures. Given a 3-valued structure S, the question that we wish to answer is whether it is possible to give a formula $\widehat{\gamma}(S)$ that characterizes the set of 2-valued structures that S represents:

$$S^{\natural} \models \widehat{\gamma}(S) \text{ iff } S^{\natural} \sqsubseteq S.$$

This question has different answers depending on what assumptions are made. The task of generating a characterizing formula is challenging because we have to find a formula that identifies when embedding is possible.

3.1 A Negative Result

In this section, we present a negative result about the possibility of characterizing a 3-valued structure by means of first-order formulas. The following theorem shows that it is not always possible to characterize an *arbitrary* 3-valued structure by a first-order formula.

Theorem 3.1.1 *There exists a 3-valued structure that represents a set of concrete structures not expressible by any first-order formula.*

Sketch of Proof: It is well known that there exists no first-order formula that characterizes 3-colorability of undirected graphs (e.g., see [Cou96]).[1] The 3-valued structure S shown in Fig. 3.1 describes undirected graphs. We draw undirected edges as two-way directed edges. This structure uses a set of predicates $\mathcal{P} = \{eq, f, b\}$, where $f(v_1, v_2)$ and $b(v_2, v_1)$ denote the forward and backward directions of an edge between the nodes v_1 and v_2. The absence of a self loop on any of the three summary nodes implies that a concrete structure can be embedded into this structure if and only if it can be colored using 3 colors. Therefore, $\gamma(S)$ cannot be characterized by a first-order formula.

---

[1]This result remains true even if the logic is extended with transitive closure and even if P = NP.
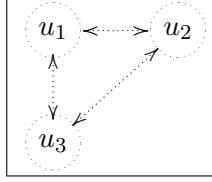
Figure 3.1: A 3-valued structure that represents 3-colorable undirected graphs.

3.2 FO-Identifiable Structures

Intuitively, the difficulty in characterizing the meaning of 3-valued structures is how to identify uniquely the correspondence between concrete and abstract nodes using a first-order formula. To avoid structures like the one shown in Fig. 3.1, we now define a subclass of 3-valued structures, in which it is possible to identify uniquely each individual using a formula.

Definition 3.2.1 *We say that a node u in a 3-valued structure S is **FO-identifiable** if there exists a formula $node_u^S(w)$ with designated free variable w such that for every concrete 2-valued structure $S^\natural$ that embeds into S using a function f, and for every concrete node $u^\natural \in U^{S^\natural}$:*

$$f(u^\natural) = u \iff S^\natural, [w \mapsto u^\natural] \models node_u^S(w) \tag{3.1}$$

S is called FO-identifiable if all the nodes in S are FO-identifiable.

The idea behind Definition 3.2.1 is to have a formula that uniquely identifies each individual u of the 3-valued structure S. This will be used to identify the set of individuals of a 2-valued structure that are mapped to u by the embedding.

Remark. One of the interesting features of FO-identifiable structures is that the structures generated by the focus operation defined in [LA00] are all FO-identifiable. Also, structures like the ones shown in Fig. 3.1 are not FO-identifiable even if $P = NP$.

We now introduce a standard concept for turning valuations into formulas.

Definition 3.2.2 *For a predicate p of arity k and Kleene value $B \in \{0, 1, 1/2\}$, we define the formula $p^B(v_1, v_2, \ldots, v_k)$ to be the **characterizing formula of the value B for p**, by:*

$$
\begin{aligned}
p^0(v_1, v_2, \ldots, v_k) &\stackrel{\text{def}}{=} \neg p(v_1, v_2, \ldots, v_k) \\
p^1(v_1, v_2, \ldots, v_k) &\stackrel{\text{def}}{=} p(v_1, v_2, \ldots, v_k) \\
p^{1/2}(v_1, v_2, \ldots, v_k) &\stackrel{\text{def}}{=} 1
\end{aligned}
$$

The main idea in the above definition is that, for $B \in \{0, 1\}$, p^B holds when the value of p is B, and for $B = 1/2$ the value of p is unrestricted. This is formalized by the following lemma:

Lemma 3.2.3 *For every 2-valued structure $S^\natural$ and assignment Z*

$$S^\natural, Z \models p^B(v_1, \ldots, v_k) \text{ iff } \iota^{S^\natural}(p)(Z(v_1), \ldots, Z(v_k)) \sqsubseteq B$$

Definition 3.2.1 is not a constructive definition, because the premise ranges over arbitrary 2-valued structures and arbitrary embedding functions. For this reason, the next section introduces a testable condition that implies FO-identifiability.

16

3.3   Bounded Structures

We now define a subclass of 3-valued structures for which we can give a constructive way to identify nodes:

Definition 3.3.1 *A **bounded structure** over vocabulary $\mathcal{P}$ is a structure $S = \langle U^S, \iota^S \rangle$ such that for every $u_1, u_2 \in U^S$, where $u_1 \neq u_2$, there exists a predicate symbol $p \in \mathcal{P}_1$ such that (i) $\iota^S(p)(u_1) \neq \iota^S(p)(u_2)$ and (ii) both $\iota^S(p)(u_1)$ and $\iota^S(p)(u_2)$ are not 1/2.*

Remark. This definition of bounded structures was given in [SRW99][2] to guarantee that shape analysis terminates; it is slightly more restrictive than the one given in [SRW02, LA00], which did not impose requirement 3.3.1(ii). However, it does not limit the set of problems handled by our method. Let S be a 3-valued structure that only satisfies the first requirement. It is possible to construct from S a finite set of bounded structures X such that $\gamma(X) = \gamma(S)$. This is based on the idea that a structure with an indefinite unary predicate value on a particular individual u, can be represented by two structures with 0 and 1 values on u, respectively. When u is a summary node we need to create an additional structure with two occurrences of u, for 0 and 1 values for the predicate.

The algorithm for computing such set X from S is shown in Fig. 3.2. It can be shown that the algorithm always terminates. The main idea is that the algorithm reduces the number of indefinite values for unary predicates by enumerating the 0- and the 1-cases.

The consequence of Definition 3.3.1 is that there is an upper bound on the size of any bounded structure S, i.e., $|U^S| \leq 2^{|\mathcal{P}_1|}$.

Example 3.3.2 Consider the class of bounded structures associated with the `List` data type declared from Fig. 2.1(a). Here the predicate symbols are $\mathcal{P} = \{n, eq\} \cup PVar \cup \{r_q \mid q \in PVar\} \cup \{is\}$, yielding unary predicates $\mathcal{P}_1 = \{x, y, t, e, r_x, r_y, r_t, r_e, is\}$ for program `insert`. Therefore, the maximal number of individuals in a structure is $2^9 = 512$. (However, this is a worst-case bound; an application of the analysis does not necessarily create structures that have this many individuals. For instance, at most 6 individuals arise in any structure in the complete analysis of `insert`.)

The following lemma shows that bounded structures are FO-identifiable:

Lemma 3.3.3 *Every bounded 3-valued structure S is FO-identifiable , where*

$$node^S_{u_i}(w) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_1} p^{\iota^S(p)(u_i)}(w) \tag{3.2}$$

Example 3.3.4 The first-order *node* formulas for the structure S shown in Fig. 2.2, are:

$$
\begin{aligned}
\text{node}^S_{u_1}(w) = \quad & x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\
& \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\
\text{node}^S_{u_2}(w) = \quad & \neg x(w) \wedge r_x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\
& \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)
\end{aligned}
\tag{3.3}
$$

---

[2]To be precise, the definition of bounded structures that was given in [SRW99] concerns only a subset of unary predicated, called *abstraction predicates*. Throughout this thesis, to simplify the presentation, we assume that all unary predicates are abstraction predicates.

procedure TR(S: Structure): Set of bounded structures
    $X := \{S\}$
    while there exists a structure $S' \in X$
        such that there exists $u \in U^{S'}$ and $p \in \mathcal{P}_1$
            such that $\iota^{S'}(p)(u) = 1/2$ do

            Select and remove S' from X
            let $S_0 = S'$
            $\iota^{S_0}(p)(u) := 0$
            $X = X \cup \{S_0\}$
            let $S_1 = S'$
            $\iota^{S_1}(p)(u) := 1$
            $X = X \cup \{S_1\}$

            if u is a summary node then
              let $S_{01} = S'$
              add new node u' to S_{01}
              set all predicate values for u' to be the same as for u
              $\iota^{S_{01}}(p)(u) := 0$
              $\iota^{S_{01}}(p)(u') := 1$

              $X = X \cup \{S_{01}\}$
            fi
    od
    return X

Figure 3.2: An algorithm that takes a structure S that is bounded according to the definition in [SRW02], and returns a set of structures X that are bounded according to the more restrictive definition in [SRW99], such that S and X represent the same set of concrete structures.

Remark. Definition 3.2.1 can be generalized to characterize 2-valued structures, by also allowing extra designated free variables for every concrete element and using equality in the *node* formula to check if the concrete element is equal to the designated variable. In particular, for a 2-valued structure S with $U^S = \{u_1, \ldots, u_n\}$, use $node_u^S(w, v_1, \ldots, v_n)$ instead of $node_u^S(w)$. Then, for each $u_i \in U^S$ we define $node_u^S(w, v_1, \ldots, v_n) \stackrel{\text{def}}{=} w = v_i$. However, the equality formula cannot be used to identify nodes in a bounded structure because equality evaluates to $1/2$ on summary nodes.

3.4   Characterizing FO-identifiable structures

To characterize an FO-identifiable 3-valued structure, we have the following issues to cope with:

1. We must ensure the existence of a surjective embedding function.

2. We must ensure that every concrete individual is represented by some abstract individual.

3. We must ensure that corresponding concrete and abstract predicate values meet the embedding condition of Eq. (2.2).

Definition 3.4.1 (First-order Characteristic Formula) *Let* $S = \langle U = \{u_1, u_2, \ldots, u_n\}, \iota \rangle$ *be an FO-identifiable 3-valued structure.*
We define the **totality characterizing formula** *to be the closed formula:*

$$\xi_{total}^S \stackrel{\text{def}}{=} \forall w : \bigvee_{i=1}^n node_{u_i}^S(w) \tag{3.4}$$

We define the **nullary characteristic formula** *to be the closed formula:*

$$\xi_{nullary}^S \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_0} p^{\iota^S(p)()} \tag{3.5}$$

For a predicate p *of arity* $r \geq 1$, *we define the* **predicate characteristic formula** *to be the closed formula:*

$$\xi^S[p] \stackrel{\text{def}}{=} \forall w_1, \ldots, w_r : \bigwedge_{\{u'_1, \ldots, u'_r\} \in U} \bigwedge_{j=1}^r node_{u'_j}^S(w_j) \Rightarrow p^{\iota^S(p)(u'_1, \ldots, u'_r)}(w_1, \ldots, w_r) \tag{3.6}$$

The **characteristic formula of** S *is defined by:*

$$\begin{aligned}
\xi^S & \stackrel{\text{def}}{=} & (\exists v_1, \ldots, v_n : \bigwedge_{i=1}^n node_{u_i}^S(v_i) \wedge \bigwedge_{k \neq j} \neg eq(v_k, v_j)) \\
& \wedge & \xi_{total}^S \\
& \wedge & \xi_{nullary}^S \\
& \wedge & \bigwedge_{r=1}^{maxR} \bigwedge_{p \in \mathcal{P}_r} \xi^S[p]
\end{aligned} \tag{3.7}$$

The **characteristic formula of set** $X \subseteq$ **3-STRUCT**$[\mathcal{P}]$ *is defined by:*

$$\widehat{\gamma}(X) = F \wedge (\bigvee_{S \in X} \xi^S) \tag{3.8}$$

Finally, for a singleton set $X = \{S\}$ *we write* $\widehat{\gamma}(S)$ *instead of* $\widehat{\gamma}(X)$.

The main ideas behind the four conjuncts of Eq. (3.7) are:

- The existential quantification in the first conjunct requires that the concrete structures have at least n distinct individuals. For each abstract individual in S, the first sub-formula locates the corresponding concrete individual. Overall, this conjunct guarantees that embedding is surjective.

- The totality formula ensures that every concrete individual is represented by some abstract individual. It guarantees that the embedding function is well-defined.

- The nullary characteristic formula ensures that the values of nullary predicates in the concrete structures are at least as precise as the values of the corresponding nullary predicates in S.

- The predicate characteristic formulas guarantee that predicate values in the concrete structures obey the requirements imposed by an embedding into S[3].

Example 3.4.2 After a small amount of simplification, the characteristic formula $\widehat{\gamma}(S)$ for the structure S shown in Fig. 2.2 is $F_{List} \wedge \xi^S$, where ξ^S is:

$$
\begin{aligned}
&\exists v_1, v_2 : \text{node}^S_{u_1}(v_1) \wedge \text{node}^S_{u_2}(v_2) \wedge \neg eq(v_1, v_2) \\
&\wedge\ \forall w : \text{node}^S_{u_1}(w) \vee \text{node}^S_{u_2}(w) \\
&\wedge\ \bigwedge_{p \in \mathcal{P}_1} \forall w_1 : \bigwedge_{i=1,2}(\text{node}^S_{u_i}(w_1) \Rightarrow p^{\iota^S(p)(u_i)}(w_1)) \\
&\wedge\ \forall w_1, w_2 : (\text{node}^S_{u_1}(w_1) \wedge \text{node}^S_{u_1}(w_2) \Rightarrow eq(w_1, w_2) \wedge \neg n(w_1, w_2) \wedge \neg n(w_2, w_1)) \\
&\qquad\qquad \wedge\ (\text{node}^S_{u_1}(w_1) \wedge \text{node}^S_{u_2}(w_2) \Rightarrow \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1))
\end{aligned}
$$

The integrity formula F_{List} is given in Example 2.2.2. The *node* formulas are given in Example 3.3.4, and the predicates for the `insert` program in Fig. 2.1(b) are shown in Table 2.1. We simplified the formula in Eq. (3.7) by combining implications that had the same premises.

Note that for a fixed $maxR$, the size of ξ^S is polynomial in the size of S. For example, TVLA only supports predicates of arity at most 2 and thus ξ^S can have at most a quadratic number of terms.

Remark. The formula $\widehat{\gamma}(X)$ is in Existential-Universal normal form (and thus decidable) whenever F is.

The following theorem shows that for every FO-identifiable structure S, the formula $\widehat{\gamma}(S)$ characterizes the set of concrete structures represented by S.

Theorem 3.4.3 *For every FO-identifiable 3-valued structure S and 2-valued structure $S^{\natural}$:*

$$
S^{\natural} \in \gamma(S) \text{ iff } S^{\natural} \models \widehat{\gamma}(S)
$$

---

[3]In bounded structures, $\xi^S[p]$ for unary predicates p can be omitted because it is implied by ξ^S_{total}. In fact, it can be omitted only for the abstraction predicates, as defined in [SRW02]; however throughout this thesis we consider all unary predicates as abstraction predicates.

Chapter 4

Characterizing General 3-Valued Structures by NP Formulas

In this chapter we show how to characterize general 3-valued strucutres.

4.1 Motivating Example

When S does not have FO-identifiable nodes, $\widehat{\gamma}(S)$ only provides a sufficient test for the embedding of concrete structures into S.

Example 4.1.1 When Eq. (3.7) is applied to the 3-valued structure S shown in Fig. 3.1, we get

$$
\begin{aligned}
\exists v_1, v_2, v_3 : \quad & \textstyle\bigwedge_{i=1}^{3} \text{node}_{u_i}^{S}(v_i) \bigwedge_{k \neq j} \neg eq(v_k, v_j) \\
\wedge \quad & \forall w : \textstyle\bigvee_{i=1}^{3} \text{node}_{u_i}^{S}(w) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{k \neq j}(\text{node}_{u_k}^{S}(w_1) \wedge \text{node}_{u_j}^{S}(w_2) \Rightarrow f^{1/2}(w_1, w_2)) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{k \neq j}(\text{node}_{u_k}^{S}(w_1) \wedge \text{node}_{u_j}^{S}(w_2) \Rightarrow b^{1/2}(w_1, w_2)) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{i=1}^{3}(\text{node}_{u_i}^{S}(w_1) \wedge \text{node}_{u_i}^{S}(w_2) \Rightarrow b^{0}(w_1, w_2)) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{i=1}^{3}(\text{node}_{u_i}^{S}(w_1) \wedge \text{node}_{u_i}^{S}(w_2) \Rightarrow f^{0}(w_1, w_2))
\end{aligned}
\tag{4.1}
$$

Because this example does not include unary predicates, the *node* formula given in Lemma 3.3.3 evaluates to $\mathbf{1}$ on all elements. Hence, Eq. (4.1) can be simplified to:

$$
\begin{aligned}
\exists v_1, v_2, v_3 : \quad & \textstyle\bigwedge_{i=1}^{3} \mathbf{1} \bigwedge_{k \neq j} \neg eq(v_k, v_j) \\
\wedge \quad & \forall w : \textstyle\bigvee_{i=1}^{3} \mathbf{1} \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{k \neq j}(\mathbf{1} \wedge \mathbf{1} \Rightarrow \mathbf{1}) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{k \neq j}(\mathbf{1} \wedge \mathbf{1} \Rightarrow \mathbf{1}) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{i=1}^{3}(\mathbf{1} \wedge \mathbf{1} \Rightarrow \neg b(w_1, w_2)) \\
\wedge \quad & \forall w_1, w_2 : \textstyle\bigwedge_{i=1}^{3}(\mathbf{1} \wedge \mathbf{1} \Rightarrow \neg f(w_1, w_2)) \\
= \exists v_1, v_2, v_3 : \quad & \textstyle\bigwedge_{k \neq j} \neg eq(v_k, v_j) \\
\wedge \quad & \forall w_1, w_2 : \neg f(w_1, w_2) \\
\wedge \quad & \forall w_1, w_2 : \neg b(w_1, w_2)
\end{aligned}
$$

The simplification is due to the fact that the implication in Eq. (3.6) unconditionally holds for all pairs of distinct individuals, because f and b evaluate to $1/2$ on those pairs, except for the requirement imposed by the absence of self-loops in S.

This formula is only fulfilled by graphs with at least 3 nodes and no edges, which are obviously 3-colorable. But this formula is too restrictive, and does not capture some 3-colorable graphs.

It is not surprising that 3-colorability cannot be expressed with a first-order formula since it is an NP-complete problem and even with transitive closure, first-order logic can only express non-deterministic logspace computations [Imm99].

4.2 Characterizing General 3-Valued Structures

Existential monadic second-order formulas is a subset of Fagin's second-order formulas [Fag75], named NP formulas, that capture NP computations. A formula in existential monadic second-order logic has the form:

$$\exists V_1, V_2, \ldots, V_n : \varphi$$

where the V_i are set variables, and φ is a first-order formula that can use membership tests in V_i. We show that in this subset of second-order logic, the characteristic formula from Definition 3.4.1 can be generalized to handle arbitrary 3-valued structures using existential quantification over set variables (with one set variable for each abstract individual).

Definition 4.2.1 (NP Characteristic Formula) *Let $S = \langle U = \{u_1, u_2, \ldots, u_n\}, \iota \rangle$ be a 3-valued structure.*

We define the following formula to ensure that the sets are non_empty:

$$\nu^S_{non\_empty}[i] \stackrel{\text{def}}{=} \exists w_i : node^S_{u_i}(w_i) \tag{4.2}$$

We define the following formula to ensure that the sets V_k, V_j are disjoint:

$$\nu^S_{disjoint}[k, j] \stackrel{\text{def}}{=} \forall w_1, w_2 : node^S_{u_k}(w_1) \wedge node^S_{u_j}(w_2) \Rightarrow \neg eq(w_1, w_2) \tag{4.3}$$

*The **NP characteristic formula of** S is defined by:*

$$
\begin{aligned}
\nu^S \stackrel{\text{def}}{=} \ & \exists V_1, \ldots, V_n : \\
& \bigwedge_{i=1}^n \nu^S_{non\_empty}[i] \wedge \bigwedge_{k \neq j} \nu^S_{disjoint}[k, j] \\
& \wedge\ \xi^S_{total} \\
& \wedge\ \xi^S_{nullary} \\
& \wedge\ \bigwedge_{r=1}^{maxR} \bigwedge_{p \in \mathcal{P}_r} \xi^S[p]
\end{aligned}
\tag{4.4}
$$

where ξ^S_{total}, $\xi^S_{nullary}$, $\xi^S[p]$ are given in Definition 3.4.1, except that $node^S_{u_i}$ is the NP formula $node^S_{u_i}(w) \stackrel{\text{def}}{=} (w \in V_i)$. (Here, we abuse notation slightly by referring to V_i in $node^S_{u_i}(w)$. This could have been formalized by passing $V_1, \ldots, V_n$ as extra parameters to $node^S_{u_i}$.)

*The **NP characteristic formula of** set $X \subseteq$ **3-STRUCT**$[\mathcal{P}]$ is defined by:*

$$\widehat{\gamma_{NP}}(X) = F \wedge \big(\bigvee_{S \in X} \nu^S \big) \tag{4.5}$$

Finally, for a singleton set $X = \{S\}$ we write $\widehat{\gamma_{NP}}(S)$ instead of $\widehat{\gamma_{NP}}(X)$.

Example 4.2.2 After a small amount of simplification, the NP characteristic formula ν^S for the graph shown in Fig. 3.1 is:

$$
\begin{aligned}
\exists V_1, V_2, V_3 : \quad & \bigwedge_{i=1}^{3} \exists w_i : \text{node}_{u_i}^{S}(w_i) \\
& \wedge\ \bigwedge_{k \neq j} \forall w_1, w_2 : (\text{node}_{u_k}^{S}(w_1) \wedge \text{node}_{u_j}^{S}(w_2) \Rightarrow \neg eq(w_1, w_2)) \\
& \wedge\ \forall w : \bigvee_{i=1}^{3} \text{node}_{u_i}^{S}(w) \\
& \wedge\ \forall w_1, w_2 : \bigwedge_{i=1}^{3}(\bigwedge_{j=1,2} \text{node}_{u_i}^{S}(w_j) \Rightarrow \neg f(w_1, w_2) \wedge \neg b(w_1, w_2)) \\
& \wedge\ \forall w_1, w_2 : \bigwedge_{k \neq j}(\text{node}_{u_k}^{S}(w_1) \wedge \text{node}_{u_j}^{S}(w_2) \Rightarrow \neg eq(w_1, w_2))
\end{aligned}
$$

and the *node* formulas are given in Definition 4.2.1.

The following theorem generalizes the result in Theorem 3.4.3 for an arbitrary 3-valued structure S, using NP-formula $\widehat{\gamma_{NP}}(S)$ to characterize the set of concrete structures represented by S.

Theorem 4.2.3 *For every* 3-*valued structure* S, *and* 2-*valued structure* $S^{\natural}$:

$$
S^{\natural} \in \gamma(S) \text{ iff } S^{\natural} \models \widehat{\gamma_{NP}}(S)
$$

Chapter 5

Supervaluational Semantics for First-Order Formulas

In this chapter, we consider the problem of how to extract information from a 3-valued structure by evaluating a query. A compositional semantics for 3-valued first-order logic is defined in [SRW02]; however, that semantics is not as precise as the one defined here. The semantics given in this section can be seen as providing the limit of obtainable precision.

5.1 Specification

The notion of supervaluational semantics, defined below, generalizes [vF66, RLS02].

Definition 5.1.1 (Supervaluational Semantics of First-Order Formulas) *Let S be a 3-valued structure and φ be a closed formula. The* **supervaluational value of** φ **in** S, *denoted by* $\langle\!\langle \varphi \rangle\!\rangle(S)$, *is defined to be the join of the values of φ obtained from each of the concrete structures that S represents, i.e.,*

$$\langle\!\langle \varphi \rangle\!\rangle(S) \stackrel{\text{def}}{=} \bigsqcup_{S^{\natural} \in \gamma(S)} [\![\varphi]\!]_2^{S^{\natural}}([]) \tag{5.1}$$

Notice that the above definition does not provide a constructive way to compute $\langle\!\langle \varphi \rangle\!\rangle(S)$ because $\gamma(S)$ is usually an infinite set.

Following Definition 5.1.1, the most-precise conservative value that can be reported for the value of formula φ in the concrete structures represented by S is

$$\langle\!\langle \varphi \rangle\!\rangle(S) = \begin{cases} 1 & \text{if } S^{\natural} \models \varphi \text{ for all } S^{\natural} \in \gamma(S) \\ 0 & \text{if } S^{\natural} \not\models \varphi \text{ for all } S^{\natural} \in \gamma(S) \\ 1/2 & \text{otherwise} \end{cases} \tag{5.2}$$

The compositional semantics given in [SRW02] and used in TVLA can yield $1/2$ for φ, even when $[\![\varphi]\!]_2^{S^{\natural}}$ is 1 for all the concrete structures $S^{\natural}$ that S represents (or when $[\![\varphi]\!]_2^{S^{\natural}}$ is 0 for all the $S^{\natural}$). In contrast, when the supervaluational semantics yields $1/2$, we *know* that any sound extraction of information from S must return $1/2$.

5.2 Implementation

If an appropriate theorem prover is at hand, $\langle\langle\varphi\rangle\rangle(S)$ can be computed as follows:

$$\langle\langle\varphi\rangle\rangle(S) = \begin{cases} 1 & \text{if } \widehat{\gamma}(S) \Rightarrow \varphi \text{ is valid} \\ 0 & \text{if } \widehat{\gamma}(S) \Rightarrow \neg\varphi \text{ is valid} \\ 1/2 & \text{otherwise} \end{cases} \tag{5.3}$$

We provide an experimental implementation of this action, as explained in the following chapter.

Chapter 6

Some Experimental Applications

In this chapter, we report on experiments in which we used $\widehat{\gamma}$ and an existing theorem prover for first-order logic to read out information from 3-valued structures in a conservative, but rather precise way. These experiments demonstrate how the approaches described in this thesis can be harnessed in the context of program analysis: the results described below go beyond what previous systems were capable of.

6.1 Implementation Details

The TVLA ([LAS00]) system performs iterative fixed-point computations and yields at every program point p a set X_p of bounded structures. It guarantees that $\gamma(X_p)$ is a superset of the 2-valued structures that can arise at p in any execution. To carry out the experiments, we have incorporated the procedure for generating $\widehat{\gamma}$ into TVLA and used SPASS [Wei] to verify validity of formulas.

SPASS follows other theorem provers in allowing axioms to express requirements on the set of structures considered. We used SPASS axioms to model integrity rules. However, because SPASS does not support transitive closure, we could only partially model transitive-closure integrity rules. We replaced uses of $n^+(v_1, v_2)$ by uses of a new designated predicate $t[n](v_1, v_2)$. Therefore, SPASS will consider some structures that do not represent potential stores. As a consequence, SPASS can fail to verify that a formula is valid for our intended set of structures; however, the opposite can never happen: whenever SPASS indicates that a formula is valid, it is indeed valid for our intended set of structures. To avoid some of the spurious failures to prove validity, we added axioms to guarantee that (i) $t[n](v_1, v_2)$ is transitive and (ii) $t[n](v_1, v_2)$ includes all of $n(v_1, v_2)$; thus, $t[n](v_1, v_2)$ includes all of $n^+(v_1, v_2)$. Because transitive closure requires a minimal set, which is not expressible in first-order logic, this approach still provides a looser set of integrity rules than we would like. However, it is still the case that whenever SPASS indicates that a formula is valid, it is indeed valid for the set of structures in which $t[n](v_1, v_2)$ is exactly $n^+(v_1, v_2)$.

An additional obstacle is that SPASS does not always terminate, because first-order logic is undecidable in general. In the examples described below, SPASS always terminated.

The rest of this chapter presents two examples. The first example, discussed in Section 6.2, demonstrates how supervaluational semantics allows us to obtain more precise information from a

3-valued structure than we would have otherwise. The second example, discussed in Section 6.3, demonstrates how to use the 3-valued structures obtained from a TVLA analysis to construct a loop invariant; this is then used to show that certain properties of a linked data structure hold on each loop iteration.

6.2 Querying Using Supervaluational Semantics

We implemented the supervaluational procedure described in Chapter 5, employing SPASS. The enhanced version of TVLA generates the formula $\widehat{\gamma}(S)$ and makes at most two calls to SPASS to compute the supervaluational value of a query φ in structure S.

Example 6.2.1 On the structure S from Fig. 2.2, the supervaluational value of the formula

$$\varphi_{\mathtt{x} \rightarrow \mathtt{next} \neq \mathtt{NULL}} \stackrel{\mathrm{def}}{=} \exists v_1, v_2 : x(v_1) \wedge n(v_1, v_2)$$

is 1. The reason for this is that S represents a list with at least two nodes; i.e., all concrete structures represented by S have at least two nodes. One node, $u_1^\natural$, corresponding to u_1 in S, is pointed to by program variable x. The other node, corresponding to the summary node u_2, must be reachable from x. Consider the sequence of nodes reachable from x, starting with $u_1^\natural$. Denote the first node in the sequence that embeds into u_2 by $u_2^\natural$. By the definition of reachability, there must be an n-link to $u_2^\natural$ from a node embedded into u_1. But the integrity rules guarantee that there is exactly one node that embeds into u_1, namely, $u_1^\natural$. Therefore, the formula $x(v_1) \wedge n(v_1, v_2)$ holds for $[v_1 \mapsto u_1^\natural, v_2 \mapsto u_2^\natural]$.

To compute this value, we applied SPASS to check the validity of $\widehat{\gamma}(S) \Rightarrow \varphi_{\mathtt{x} \rightarrow \mathtt{next} \neq \mathtt{NULL}}$; SPASS indicated that the formula is valid. This guarantees that the formula $\varphi_{\mathtt{x} \rightarrow \mathtt{next} \neq \mathtt{NULL}}$ evaluates to 1 on all the concrete structures that embed into S.

Note that this formula will evaluate to $1/2$ in TVLA, because $n(u_1, u_2) = 1/2$.

6.3 Generating and Querying a Loop Invariant

TVLA computes, for each program point p, a set X_p of bounded structures that overapproximate the set of stores that may occur at that point. We then used the enhanced version of TVLA to generate $\widehat{\gamma}(X_p)$. Because TVLA is sound, $\widehat{\gamma}(X_p)$ must be an invariant that holds at program point p. In particular, when p is a program point that begins a loop, $\widehat{\gamma}(X_p)$ is a loop invariant.

Example 6.3.1 Let $X = \{S_i \mid i = 1, \dots, 5\}$ denote the set of five 3-valued structures that TVLA found at the beginning of the loop in the `insert` program from Fig. 2.2. Appendix A shows the S_i and their characteristic formulas. The loop invariant is defined by

$$\widehat{\gamma}(X) = F_{List} \wedge (\bigvee_{i=1}^{5} \xi^{S_i})$$

27

We checked that in every structure that can occur at the beginning of the loop, x points to a valid list, i.e., one that is acyclic and unshared. This property is defined by the following formulas:

$$\text{acyc}_x \stackrel{\text{def}}{=} \forall v_1, v_2 : r_x(v_1) \wedge n^+(v_1, v_2) \Rightarrow \neg n^+(v_2, v_1)$$
$$\text{uns}_x \stackrel{\text{def}}{=} \forall v : r_x(v) \Rightarrow$$
$$\neg(\exists w_1, w_2 : \neg eq(w_1, w_2) \wedge n(w_1, v) \wedge n(w_2, v))$$
$$\text{list}_x \stackrel{\text{def}}{=} \text{acyc}_x \wedge \text{uns}_x$$

We applied SPASS to check the validity of $\widehat{\gamma}(S) \Rightarrow \text{list}_x$; SPASS indicated that the formula is valid.

It is interesting to note that the size of ξ^{S_2} is bigger than the size of ξ^{S_1}. This is natural because S_2 has more definite values, which imposes more restrictions than are imposed by S_1.

Chapter 7

Characterizing Canonical Abstraction by First-Order Formulas

This section defines an alternative abstract domain for use in shape analysis (and other logic-based analyses). The ordering relation in the abstract doman This section provides an alternative formulation of the abstraction which does not rely on embedding, as is the case for the abstraction defined in Section 2.4. Instead, the new abstract domain relies on a simple operation called *canonical abstraction*, which maps concrete structures into a limited subset of bounded structures.

7.1 Canonical Abstraction

Canonical abstraction was defined in [SRW99] as an abstraction with the following properties:

- Provides a uniform way to obtain 3-valued structures of a priori bounded size. This is important to automatically derive properties of programs with loops by employing iterative fixed-point algorithms. Canonical abstraction maps concrete individuals into abstract individuals according to the definite values of the unary predicates.

- The information loss is minimized when multiple individuals of S are mapped to the same individual in S',

This is formalized by the following definition:

Definition 7.1.1 *A structure $S' = \langle U^{S'}, \iota^{S'} \rangle$ is a* **canonical abstraction** *of a structure S, if there exists a surjective function $canonic \colon U^S \to U^{S'}$, induced by the following mapping:*

$$canonic_c(u) = u_{\{p \in \mathcal{P}_1 | \iota^S(p)(u)=1\}, \{p \in \mathcal{P}_1 | \iota^S(p)(u)=0\}} \tag{7.1}$$

such that, for every $p \in \mathcal{P}_k$ of arity k,

$$\iota^{S'}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{\substack{u_i \in U^S, \text{ s.t.} \\ canonic(u_i) = u'_i \in U^{S'}, \\ 1 \le i \le k}} \iota^S(p)(u_1, \ldots, u_k) \tag{7.2}$$

We say that $S' = canonic(S)$.

The name "$u_{\{p\in\mathcal{P}_1|\iota^S(p)(u)=1\},\{p\in\mathcal{P}_1|\iota^S(p)(u)=0\}}$" is known as the **canonical name** of individual u. The subscript on the canonical name of u involves two sets of unary predicate symbols: (i) those that are true at u, and (ii) those that are false at u.

Example 7.1.2 In structure S from Fig. 2.2, the canonical names of the individuals are as follows:

Individual	Canonical Name
u_1	$u_{\{x,r_x\},\{y,t,e,is,r_y,r_t,r_e\}}$
u_2	$u_{\{r_x\},\{x,y,t,e,is,r_y,r_t,r_e\}}$

Note that in context of the canonical abstraction, S represents lists with at least three nodes, pointed to by x, but it does not include a list with two nodes. The reason is that predicates n and eq have indefinite values in S; but a list with two nodes does not have both 0 and 1 values for the corresponding entries, as required for minimizing information loss as formalized in Eq. (7.2). In contrast, according to the abstraction that relies on embedding, defined in Section 2.4, S represents lists with two or more elements.

To characterize canonical abstraction, we have to redefine the subset of 3-valued structures of interest. We are interested only in 3-valued structures that that are "images of canonical abstraction" (*ICA*), i.e., results of applying canonical abstraction to 2-valued structures.

Definition 7.1.3 (Image of canonical abstraction (*ICA*)) *Structure S is an* ICA *if there exists a 2-valued structure $S^\natural$ such that S is the canonical abstraction of $S^\natural$.*

Concretization of 3-Valued Structures. Canonical abstraction allows us to define the (potentially infinite) set of concrete structures represented by a set of 3-valued structures, that are *ICA*

Definition 7.1.4 (Concretization of 3-Valued Structures) *For a set of structures $X \subseteq$ 3-STRUCT[$\mathcal{P}$], that are* ICA*, we denote by $\gamma_c(X)$ the set of 2-valued structures that X represents, i.e.,*

$$\gamma_c(X) = \{S^\natural \in \text{2-STRUCT}[\mathcal{P}] \mid \text{exists } S \in X \text{ such that } S \text{ is the canonical abstraction of } S^\natural \text{ and } S^\natural \models F\}$$
(7.3)

Also, for a singleton set $X = \{S\}$ we write $\gamma_c(S)$ instead of $\gamma_c(X)$.

The abstract domain is a powerset of *ICA* structures, where the order relation is set inclusion. This defines a Galois connection between sets of 2-valued structures and sets of *ICA* structures.

7.2 FO-Identifiable Structures

We have to redefine the notion of FO-identifiable nodes, given in Section 3.2, to use canonical abstraction rather than embedding, used in Definition 3.2.1.

Definition 7.2.1 *We say that a node u in a 3-valued structure S is* **FO-identifiable** *if there exists a formula $node_u^S(w)$ with designated free variable w, such that for every concrete 2-valued structure $S^\natural$, if S is the canonical abstraction of $S^\natural$, i.e., $S^\natural \in \gamma_c(S)$, then for every concrete node $u^\natural \in U^{S^\natural}$:*

$$canonic(u^\natural) = u \iff S^\natural, [w \mapsto u^\natural] \models node_u^S(w)$$
(7.4)

S is called FO-identifiable if all the nodes in S are FO-identifiable.

7.3 Characterizing Canonical Abstraction

An *ICA* structure is always a bounded structure, in which nullary and unary predicates always have definite values.[1] This is formalized by the following lemma:

Lemma 7.3.1 *If* 3*-valued structure* $S = \langle U^S, \iota^S \rangle$ *over vocabulary* $\mathcal{P}$ *is* ICA *then:*

(i) S *is a bounded structure.*

(ii) *For each nullary predicate* p, $\iota^S(p)() \in \{0, 1\}$.

(iii) *For each element* $u \in U$ *and each unary predicate* p, $\iota^S(p)(u) \in \{0, 1\}$.

The following lemma shows that *ICA* structures are FO-identifiable:

Lemma 7.3.2 *Every* 3*-valued structure* S *that is an* ICA *is FO-identifiable, where*

$$node^S_{u_i}(w) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_1} p^{\iota^S(p)(u_i)}(w) \tag{7.5}$$

Using this fact, we can define the formula τ^S that characterizes the set of 2-valued structures represented by S using canonical abstraction. The formula τ^S is merely ξ^S with additional conjuncts to ensure that the information loss is minimized, i.e., for every predicate p of arity $r > 1$ and every $1/2$ entry of p, the concrete structure has both a corresponding 1 entry and a corresponding 0 entry.

Definition 7.3.3 (First-Order Characteristic Formula for Canonical Abstraction) *Let* 3*-valued structure* $S = \langle U^S, \iota \rangle$ *be an ICA.*
For a predicate p *of arity* $r > 1$, *we define the closed formula for* p:

$$
\begin{aligned}
\tau^S[p] \quad &\stackrel{\text{def}}{=} \bigwedge_{\{u'_1,\ldots,u'_r\} \in U^S} \bigwedge_{\iota^S(p)(u'_1,\ldots,u'_r)=1/2} \\
&\wedge \quad \exists w_1,\ldots,w_r : \bigwedge_{j=1}^r node^S_{u'_j}(w_j) \wedge p(w_1,\ldots,w_r) \\
&\wedge \quad \exists w_1,\ldots,w_r : \bigwedge_{j=1}^r node^S_{u'_j}(w_j) \wedge \neg p(w_1,\ldots,w_r)
\end{aligned}
\tag{7.6}
$$

The formula of S *is defined by:*

$$\tau^S \stackrel{\text{def}}{=} \xi^S \wedge \bigwedge_{r=2}^{maxR} \bigwedge_{p \in \mathcal{P}_r} \tau^S[p] \tag{7.7}$$

The **characteristic formula for canonical abstraction of a set of *ICA* structures** $X \subseteq$ 3-STRUCT[$\mathcal{P}$] *is defined by*

$$\widehat{\gamma}_c(X) = F \wedge \left(\bigvee_{S \in X} \tau^S \right) \tag{7.8}$$

Also, for a singleton set $X = \{S\}$, *where* S *is an* ICA*structure, we write* $\widehat{\gamma}_c(S)$ *instead of* $\widehat{\gamma}_c(X)$.

---

[1]If not all unary predicates are defined as abstraction predicates, then the result may be a bounded structure of the less restrictive kind mentioned in Section 3.3. Also, unary predicates that are not abstraction predicates may have indefinite values.

Example 7.3.4 The characteristic formula for canonical abstraction of the structure S shown in Fig. 2.2 is:

$$
\begin{aligned}
\widehat{\gamma}_c(S) \;&=\; \widehat{\gamma}(S) \\
&\wedge\; \forall w_1, w_2 : \exists w_1, w_2 : \text{node}_{u_1}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \wedge n(w_1, w_2) \\
&\wedge\; \exists w_1, w_2 : \text{node}_{u_1}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \wedge \neg n(w_1, w_2) \\
&\wedge\; \exists w_1, w_2 : \text{node}_{u_2}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \wedge n(w_1, w_2) \\
&\wedge\; \exists w_1, w_2 : \text{node}_{u_2}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \wedge \neg n(w_1, w_2) \\
&\wedge\; \exists w_1, w_2 : \text{node}_{u_2}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \wedge eq(w_1, w_2) \\
&\wedge\; \exists w_1, w_2 : \text{node}_{u_2}^S(w_1) \wedge \text{node}_{u_2}^S(w_2) \wedge \neg eq(w_1, w_2)
\end{aligned}
\tag{7.9}
$$

where $\widehat{\gamma}(S)$ is given in Example 3.4.2. As stated in Example 7.1.2, S does not represent a list with two nodes. Indeed, a 2-valued structure that represents a list with two nodes does not satisfy this formula, because the last four lines of Eq. (7.9) can not be satisfied by any assignment within this structure.

Remark. Note that $\widehat{\gamma}_c$ is in Existential-Universal normal form (and thus decidable) whenever F is.

Theorem 7.3.5 *For every 3-valued structure S that is an* ICA *and a 2-valued structure $S^\natural$:*

$$
S^\natural \in \gamma_c(S) \text{ iff } S^\natural \models \widehat{\gamma}_c(S)
$$

Chapter 8

Related Work

There is a sizeable literature on *structure-description formalisms* for describing properties of linked data structures (see [BRS99, SRW02] for references). The motivation for the present work was to understand the expressive power of the shape abstractions defined in [SRW02].

In previous work, Benedikt et al. [BRS99] showed how to translate two kinds of shape descriptors, "path matrices" [Hen90, HN90] and the variant of shape graphs discussed in [SRW98], into a logic called L_r ("logic of reachability expressions"). The shape graphs from [SRW98] are also amenable to the techniques presented in this thesis: the characteristic formula defined in Eq. (3.7) is much simpler than the translation to L_r given in [BRS99]; moreover, Eq. (3.7) applies to a more general class of shape descriptors. However, the logic used in [BRS99] is decidable, which guarantees that terminating procedures can be given for problems that can be addressed using L_r.

The Pointer Analysis Logic Engine (PALE) [MS01] provides a structure-description formalism that serves as an assertion language; assertions are translated to second-order monadic logic and fed to MONA. PALE does not handle all data structures, but can handle all data structures describable as graph types [KS93]. Because the logic used by MONA is decidable, PALE is guaranteed to terminate.

One point of contrast between the shape abstractions based on 3-valued structures studied in this paper and both L_r and the PALE assertion language is that the powerset of 3-valued structures forms an abstract domain. This means that 3-valued structures can be used for program analysis by setting up an appropriate set of equations and finding its fixed point [SRW02, RSL03]. In contrast, when PALE is used for program analysis, an invariant must be supplied for each loop.

Other structure-description formalisms in the literature include ADDS [HHN92] and shape types [FM97].

The supervaluational semantics for first-order logic discussed in Chapter 5 is related to a number of other supervaluational semantics for partial logics and 3-valued logics discussed in the literature [vF66, Bla02, BG00, RLS02]. Compared to previous work, an innovation of Eq. (5.3) is the use of $\hat{\gamma}$ to translate a 3-valued structure to a formula. In fact, Eq. (5.3) is an example of a general reductionist strategy for providing a supervaluational evaluation procedure for abstract domains by using existing logics and theorem-provers/decision-procedures; as we discuss at greater length in [RSY03], a supervaluational evaluation procedure can be obtained whenever an appropriate logic, $\hat{\gamma}$ function, and theorem-prover/decision-procedure are available.

Chapter 9

Final Remarks

In [RSY03], we discuss how to perform all operations required for abstract interpretation in the most-precise way possible (relative to the abstraction in use), if certain primitive operations can be carried out, and if a sufficiently powerful theorem prover is at hand. Chief among the primitive operations that must be available is $\widehat{\gamma}$; thus, the material that has been presented in here shows how to fulfill the requirements of [RSY03] for a family of abstractions based on 3-valued structures (essentially those used in [SRW02] and in the TVLA system [LAS00]).

In ongoing work, we are investigating the feasibility of actually applying the techniques from [RSY03] to perform abstract interpretation for abstractions based on 3-valued structures. This approach could be more precise than TVLA because, for instance, it would take into account in a first-class way the integrity formula of the abstraction. In contrast, in TVLA some operations temporarily ignore the integrity formula, and rely on later clean-up steps to rectify matters.

We are also investigating the feasibility of using the results from this paper to develop a more precise and scalable version of TVLA by using *assume-guarantee* reasoning. The idea is to allow arbitrary first-order formulas with transitive closure to be used to express pre- and post-conditions, and to analyze the code for each procedure separately.

Finally, this work provides a way for TVLA to be used as a generator for loop invariants for existing verification systems, and to provide a "search space" for systems like [FQ02]. This system tries combinations of inferred or user-supplied formulas to see if it can identify loop invariants. However, currently it cannot infer the formulas required for list or tree data structures; in this case, TVLA can be used as a sub-procedure for generating such formulas.

References

[App98]   A. W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[BG00]   G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *Proc. CONCUR.* Springer-Verlag, 2000.

[Bla02]   S. Blamey. Partial logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Phil. Logic, 2nd. Ed., Vol. 5*, pages 261–353. Kluwer Academic Publishers, 2002.

[BRS99]   M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proceedings of the 1999 European Symposium On Programming*, pages 2–19, March 1999.

[CC77]   P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[CC79]   P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[CGL94]   E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *Trans. on Prog. Lang. and Syst.*, 16(5):1512–1542, 1994.

[Cou96]   B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In N. Immerman and P.G. Kolaitis, editors, *Descriptive Complexity and Finite Models: Proceedings of a DIAMCS Workshop*, chapter 2, pages 33–57. American Mathematical Society, 1996.

[CWZ90]   D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[Dam96]   D. Dams. *Abstract Interpretation and Partial Refinement for Model Checking.* PhD thesis, Technical Univ. of Eindhoven, Eindhoven, The Netherlands, July 1996.

[Deg95]   J. Degener. ANSI C yacc grammar. http://www.lysator.liu.se/c/ANSI-C-grammar-1.html, 1995.

[DRS03]   N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *To appear in SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2003.

[Fag75]   R. Fagin. Monadic generalized spectra. *Z. Math. Logik*, 21:89–96, 1975.

[FM97]   P. Fradet and D. Le Metayer. Shape types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1997. ACM Press.

[FQ02]   C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Princ. of Prog. Lang.*, 2002.

[Hen90]   L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.

[HHN92]   L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.

[HN90]   L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Par. and Dist. Syst.*, 1(1):35–47, January 1990.

[Imm99]   N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.

[JM81]   N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[KR78]   B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KS93]   N. Klarlund and M. Schwartzbach. Graph types. In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1993. ACM Press.

[LA00]   T. Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000.

[LAS00]   T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, pages 280–301, 2000.

[Leu02]   R. Leupers. An executable intermediate representation for retargetable compilation and high-level code optimization. Unpublished, 2002.

[MS01]   A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 221–231, 2001.

[NMRW02]   G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Int. Conf. on Comp. Construct.*, pages 213–228, 2002. See "http://manju.cs.berkeley.edu/cil/".

[NNH99]   F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[RLS02]   T. Reps, A. Loginov, and M. Sagiv. Semantic minimization of 3-valued propositional formulae. In *Proc. IEEE Symp. on Logic in Computer Science*, 2002.

[RSL03]   T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symp. On Programming*, 2003.

[RSY03]   T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. Submitted for publication, 2003.

[SRW98]   M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.

[SRW99]   M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, pages 105–118, New York, NY, January 1999. ACM Press.

[SRW02]   M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.

[TSBTG]   Microsoft Research The Semantics Based Tools Group. Microsoft ast toolkit. Licensed Software.

[vF66]   B. van Fraassen. Singular terms, truth-value gaps, and free logic. *J. Phil*, 63(17):481–495, 1966.

[Wei]   C. Weidenbach. SPASS: An automated theorem prover for first-order logic with equality. Available at "http://spass.mpi-sb.mpg.de/index.html".

[Yor02]   G. Yorsh. Translation rules from C to CoreC - informal pseudo-code description. Available at "http://www.math.tau.ac.il/$\sim$ gretay/GFC", 2002.

[YRSW03]   G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterization of heap abstractions. Submitted for publication, 2003.

List of Figures

List of Tables

Appendix A

Result of Example 6.3.1

Structure	CharacteristicFormula
$x, y \rightarrow (u_1) \xrightarrow{n} (u_2)$ with self-loop n on u_2; S_1; r_x, r_y under u_1, r_x, r_y under u_2	$\begin{aligned} \text{node}_{u_1}^{S_1}(w) &= x(w) \wedge y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_2}^{S_1}(w) &= \neg x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \end{aligned}$ <hr> $\begin{aligned} \xi^{S_1} &= \exists v_1, v_2 : \bigwedge_{i=1,2} \text{node}_{u_i}^{S_1}(v_i) \wedge \neg eq(v_1, v_2) \\ &\wedge\ \forall w : \bigvee_{i=1,2} \text{node}_{u_i}^{S_1}(w) \\ &\wedge\ \forall w_1, w_2 : \bigwedge_{i=1,2} \text{node}_{u_i}^{S_1}(w_i) \Rightarrow \\ &\quad \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1) \\ &\wedge\ \forall w_1, w_2 : \bigwedge_{i=1,2} \text{node}_{u_1}^{S_1}(w_i) \Rightarrow \\ &\quad \wedge eq(w_1, w_2) \wedge \neg n(w_1, w_2) \end{aligned}$
$x \rightarrow (u_1) \xrightarrow{n} (u_2)$; S_2; r_x under u_1, y, r_x, r_y under u_2	$\begin{aligned} \text{node}_{u_1}^{S_2}(w) &= x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_2}^{S_2}(w) &= \neg x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \end{aligned}$ <hr> $\begin{aligned} \xi^{S_2} &= \exists v_1, v_2 : \bigwedge_{i=1,2} \text{node}_{u_i}^{S_2}(v_i) \wedge \neg eq(v_1, v_2) \\ &\wedge\ \forall w : \bigvee_{i=1,2} \text{node}_{u_i}^{S_2}(w) \\ &\wedge\ \forall w_1, w_2 : \bigwedge_{i=1,2} \text{node}_{u_i}^{S_1}(w_i) \Rightarrow \\ &\quad \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1) \wedge n(w1, w2) \\ &\wedge\ \forall w_1, w_2 : \bigwedge_{i=1,2} \text{node}_{u_1}^{S_1}(w_i) \Rightarrow \\ &\quad \wedge eq(w_1, w_2) \wedge \neg n(w_1, w_2) \\ &\wedge\ \forall w_1, w_2 : \bigwedge_{i=1,2} \text{node}_{u_2}^{S_1}(w_i) \Rightarrow \\ &\quad \wedge eq(w_1, w_2) \wedge \neg n(w_1, w_2) \end{aligned}$
$x \rightarrow (u_1) \xrightarrow{n} (u_2) \xrightarrow{n} (u_3)$ with self-loop n on u_3; S_3; r_x under u_1, y, r_x, r_y under u_2, r_x, r_y under u_3	$\begin{aligned} \text{node}_{u_1}^{S_3}(w) &= x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_2}^{S_3}(w) &= \neg x(w) \wedge y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \\ \text{node}_{u_3}^{S_3}(w) &= \neg x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w) \\ &\wedge\ r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w) \end{aligned}$ <hr> $\begin{aligned} \xi^{S_3} &= \exists v_1, v_2, v_3 : \bigwedge_{i=1,2,3} \text{node}_{u_i}^{S_3}(v_i) \wedge \bigwedge_{k \neq j} \neg eq(v_k, v_j) \\ &\wedge\ \forall w : \bigvee_{i=1,2,3} \text{node}_{u_i}^{S_3}(w) \\ &\wedge\ \forall w_1, w_2 : \left(\bigwedge_{i=1,2} \text{node}_{u_1}^{S_3}(w_i) \Rightarrow \right. \\ &\quad \left. eq(w_1, w_2) \wedge \neg n(w_1, w_2) \right) \\ &\wedge\ \left(\bigwedge_{i=1,2} \text{node}_{u_2}^{S_3}(w_i) \Rightarrow \right. \\ &\quad \left. eq(w_1, w_2) \wedge \neg n(w_1, w_2) \right) \\ &\wedge\ \left(\text{node}_{u_1}^{S_3}(w_1) \wedge \text{node}_{u_2}^{S_3}(w_2) \Rightarrow \right. \\ &\quad \left. \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1) \wedge n(w_1, w_2) \right) \\ &\wedge\ \left(\text{node}_{u_2}^{S_3}(w_1) \wedge \text{node}_{u_3}^{S_3}(w_2) \Rightarrow \right. \\ &\quad \left. \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1) \right) \\ &\wedge\ \left(\text{node}_{u_1}^{S_3}(w_1) \wedge \text{node}_{u_3}^{S_3}(w_2) \Rightarrow \right. \\ &\quad \left. \neg eq(w_1, w_2) \wedge \neg n(w_2, w_1) \wedge \neg n(w_1, w_2) \right) \end{aligned}$

Table A.1: The left column shows the structures that arise at the beginning of the loop in the `insert` program from Fig. 2.1(b). The right column shows the characteristic formula for each structure. Note that we omit the redundant sub-formulas $\xi^S[p]$, for $p \in \mathcal{P}_1$, that are part of ξ_{total}^S and $\text{node}_{u_j}^{S_i}(w)$ definitions.

41

Structure	CharacteristicFormula

Structure	CharacteristicFormula		
	$\mathrm{node}^{S_4}_{u_1}(w)$ =		$x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_x(w) \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)$
	$\mathrm{node}^{S_4}_{u_1}(w)$ =		$\neg x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_x(w) \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)$
	$\mathrm{node}^{S_4}_{u_3}(w)$ =		$\neg x(w) \wedge y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)$
	$\mathrm{node}^{S_4}_{u_4}(w)$ =		$\neg x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)$
	ξ^{S_1} =		$\exists v_1,\dots,v_4 : \bigwedge_{i=1,\dots,4} \mathrm{node}^{S_4}_{u_i}(v_i) \wedge \bigwedge_{k\neq j} \neg eq(v_k, v_j)$
	$\wedge$		$\forall w : \bigvee_{i=1,\dots,4} \mathrm{node}^{S_4}_{u_i}(w)$
	$\wedge$	$\forall w_1, w_2 :$	
			$(\bigwedge_{i=1,2} \mathrm{node}^{S_4}_{u_1}(w_i) \Rightarrow eq(w_1,w_2) \wedge \neg n(w_1,w_2))$
	$\wedge$		$(\bigwedge_{i=1,2} \mathrm{node}^{S_4}_{u_3}(w_i) \Rightarrow eq(w_1,w_2) \wedge \neg n(w_1,w_2))$
	$\wedge$		$(\mathrm{node}^{S_4}_{u_1}(w_1) \wedge \mathrm{node}^{S_4}_{u_2}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1))$
	$\wedge$		$(\mathrm{node}^{S_4}_{u_2}(w_1) \wedge \mathrm{node}^{S_4}_{u_3}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1))$
	$\wedge$		$(\mathrm{node}^{S_4}_{u_1}(w_1) \wedge \mathrm{node}^{S_4}_{u_3}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1) \wedge \neg n(w_1,w_2))$
	$\wedge$		$(\mathrm{node}^{S_4}_{u_3}(w_1) \wedge \mathrm{node}^{S_4}_{u_4}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1))$
	$\wedge$		$(\mathrm{node}^{S_4}_{u_1}(w_1) \wedge \mathrm{node}^{S_4}_{u_4}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1) \wedge \neg n(w_1,w_2))$
	$\wedge$		$(\mathrm{node}^{S_4}_{u_2}(w_1) \wedge \mathrm{node}^{S_4}_{u_4}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1) \wedge \neg n(w_1,w_2))$

Structure S_4: $\mathtt{x} \to (u_1) \xrightarrow{n} (u_2) \xrightarrow{n} (u_3) \xrightarrow{n} (u_4)$, with labels r_x, r_x, y, r_x, r_y, r_x, r_y

Structure	CharacteristicFormula		
	$\mathrm{node}^{S_5}_{u_1}(w)$ =		$x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_x(w) \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge r_e(w) \wedge \neg is(w)$
	$\mathrm{node}^{S_5}_{u_2}(w)$ =		$\neg x(w) \wedge \neg y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_e(w) \wedge \neg r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)$
	$\mathrm{node}^{S_5}_{u_3}(w)$ =		$\neg x(w) \wedge y(w) \wedge \neg t(w) \wedge \neg e(w)$
	$\wedge$		$r_x(w) \wedge r_y(w) \wedge \neg r_t(w) \wedge \neg r_e(w) \wedge \neg is(w)$
	ξ^{S_3} =		$\exists v_1, v_2, v_3 : \bigwedge_{i=1,2,3} \mathrm{node}^{S_5}_{u_i}(v_i) \wedge \bigwedge_{k\neq j} \neg eq(v_k, v_j)$
	$\wedge$		$\forall w : \bigvee_{i=1,2,3} \mathrm{node}^{S_5}_{u_i}(w)$
	$\wedge$	$\forall w_1, w_2 :$	
			$(\bigwedge_{i=1,2} \mathrm{node}^{S_5}_{u_1}(w_i) \Rightarrow eq(w_1,w_2) \wedge \neg n(w_1,w_2))$
	$\wedge$		$(\bigwedge_{i=1,2} \mathrm{node}^{S_5}_{u_3}(w_i) \Rightarrow eq(w_1,w_2) \wedge \neg n(w_1,w_2))$
	$\wedge$		$(\mathrm{node}^{S_5}_{u_1}(w_1) \wedge \mathrm{node}^{S_5}_{u_2}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1))$
	$\wedge$		$(\mathrm{node}^{S_5}_{u_2}(w_2) \wedge \mathrm{node}^{S_5}_{u_3}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1))$
	$\wedge$		$(\mathrm{node}^{S_5}_{u_1}(w_1) \wedge \mathrm{node}^{S_5}_{u_3}(w_2) \Rightarrow \neg eq(w_1,w_2) \wedge \neg n(w_2,w_1) \wedge \neg n(w_1,w_2))$

Structure S_5: $\mathtt{x} \to (u_1) \xrightarrow{n} (u_2) \xrightarrow{n} (u_3)$, with labels r_x, r_x, y, r_x, r_y

Table A.2: Table A.1 continued.

Appendix B

Proofs

Lemma 3.2.3 *For every* 2*-valued structure* $S^\natural$ *and assignment* Z

$$S^\natural, Z \models p^B(v_1, v_2, \dots, v_k) \text{ iff } \iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) \sqsubseteq B$$

Proof of the if direction: Suppose that $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) \sqsubseteq B$. There are two cases to consider: (i) $B = 1/2$ or (ii) $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) = B$. If $B = 1/2$, then by Definition 3.2.2, $p^B(v_1, v_2, \dots, v_k) = 1$ and thus by Definition 2.1.3, $S^\natural, Z \models p^B(v_1, v_2, \dots, v_k)$ for all Z. If B = 1, then $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) = 1$ or by definition Definition 2.1.3, $S^\natural, Z \models p(v_1, v_2, \dots, v_k)$ which is $S^\natural, Z \models p^1(v_1, v_2, \dots, v_k)$ by Definition 3.2.2. Similarly, if B = 0, then $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) = 0$ implies that $S^\natural, Z \models \neg p(v_1, v_2, \dots, v_k) = p^0(v_1, v_2, \dots, v_k)$.

Proof of the only-if direction: Assume that $S^\natural, Z \models p^B(v_1, v_2, \dots, v_k)$. If $B = 1/2$, then $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) \sqsubseteq B$ trivially holds. If $B = 0$, apply Definition 3.2.2 to the assumption to get $S^\natural, Z \models \neg p(v_1, v_2, \dots, v_k)$, which implies $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) = 0 = B$, by Definition 2.1.3. Similarly, if $B = 1$, the assumption implies $\iota^{S^\natural}(p)(Z(v_1), Z(v_2), \dots, Z(v_k)) = 1 = B$.

Lemma 3.3.3 *Every bounded* 3*-valued structure* S *is FO-identifiable, where*

$$node_{u_i}^S(w) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_1} p^{\iota^S(p)(u_i)}(w)$$

Proof: Consider a bounded 3-valued structure $S = \{U, \iota^S\}$. We shall show that every element $u \in U$ is FO-identifiable using the formula defined in Eq. (3.2). Let $S^\natural$ be a 2-valued structure that embeds into S using a function f, and let $u^\natural$ be a concrete element in $U^{S^\natural}$. By Definition 3.2.1, we have to show that the following holds:

$$f(u^\natural) = u \iff S^\natural, [w \mapsto u^\natural] \models node_u^S(w)$$

Proof of the if direction: Suppose that $S^\natural, [w \mapsto u^\natural] \models node_u^S(w)$. In particular, each conjunct of

43

node_u^S must hold, i.e., for each predicate $p \in \mathcal{P}_1$, $S^{\natural}, [w \mapsto u^{\natural}] \models p^{\iota^S(p)(u)}(w)$. Using Lemma 3.2.3 we get that $\iota^{S^{\natural}}(p)(u^{\natural}) \sqsubseteq \iota^S(p)(u)$. In addition, the embedding condition in Eq. (2.2), requires, in particular, that for each unary predicate p $\iota^{S^{\natural}}(p)(u^{\natural}) \sqsubseteq \iota^S(p)(f(u^{\natural}))$ holds. Let $u_1 = f(u^{\natural})$. For the sake of argument, assume that $u_1 \neq u$. Recall that S is a bounded structure, in which every individual must have a unique combination of definite values of unary predicates. As a consequence, there must be a unary predicate p such that $\iota^S(p)(u_1) \neq \iota^S(p)(u)$ and the value of p on both u_1 and u is definite. This yields a contradiction, because $\sqsubseteq$ on definite values implies equality; however $\iota^{S^{\natural}}(p)(u^{\natural}) = \iota^S(p)(u)$ and $\iota^{S^{\natural}}(p)(u^{\natural}) = \iota^S(p)(f(u^{\natural})) = \iota^S(p)(u_1)$ can not hold simultaneously, by the assumption.

Proof of the only-if direction: Suppose that $f(u^{\natural}) = u$. Using Eq. (2.2), the embedding function f guarantees that for each unary predicate p, $\iota^{S^{\natural}}(p)(u^{\natural}) \sqsubseteq \iota^S(p)(f(u^{\natural}))$. This means that $S^{\natural}, [w \mapsto u^{\natural}] \models p^{\iota^S(p)(f(u^{\natural}))}(w)$ by Lemma 3.2.3, or $S^{\natural}, [w \mapsto u^{\natural}] \models p^{\iota^S(p)(u)}(w)$ by the assumption. This holds for all unary predicates, and thus holds for their conjunction as well, namely, for the formula node_u^S.

Theorem 3.4.3 *For every FO-identifiable 3-valued structure S, and a 2-valued structure $S^{\natural}$:*

$$S^{\natural} \in \gamma(S) \ iff \ S^{\natural} \models \widehat{\gamma}(S)$$

Proof: In Lemma B.0.6, we show that the if-direction holds, even when S is not FO-identifiable, i.e., every concrete structure satisfying the characteristic formula $\widehat{\gamma}(S)$ is indeed in $\gamma(S)$. In Lemma B.0.7 we show the only-if part, i.e., for an FO-identifiable structure, the other direction is also true.

Lemma B.0.6 *Let S be a first-order structure with set of individuals $U = \{u_1, u_2, \ldots, u_n\}$. Let $\text{node}_{u_i}^S(w)$ used in $\widehat{\gamma}(S)$ be an arbitrary first-order formula free in w. Then, for all $S^{\natural}$ such that $S^{\natural} \models \widehat{\gamma}(S)$, $S^{\natural} \in \gamma(S)$.*

Proof: Let $S^{\natural} = \langle U^{\natural}, \iota^{\natural} \rangle$ be a concrete structure such that $S^{\natural} \models \widehat{\gamma}(S)$. We shall construct a surjective function $f \colon U^{\natural} \to U$ such that $S^{\natural} \sqsubseteq^f S$. Let $Z^{\natural}$ be an assignment such that $S^{\natural}, Z^{\natural} \models \varphi$ where $\varphi \overset{\text{def}}{=} \bigwedge_{i=1}^n \text{node}_{u_i}^S(v_i) \wedge \bigwedge_{k \neq j} \neg eq(v_k, v_j)$ (the first line of Eq. (3.7) without the existential quantification). Define the function $f \colon U^{\natural} \to U$ by:

$$f(u^{\natural}) = \begin{cases} u_i & \text{if } Z^{\natural}(v_i) = u^{\natural} \\ u_j & \text{if for all } i, \ Z^{\natural}(v_i) \neq u^{\natural} \text{ and } u_j \text{ is an arbitrary element such that} \\ & S^{\natural}, [w \mapsto u^{\natural}] \models \text{node}_{u_j}^S(w) \end{cases} \qquad \text{(B.1)}$$

Let us show that every concrete element is mapped to some element in U. In the case that $Z(v_i) = u^{\natural}$, the concrete element $u^{\natural}$ is mapped to $u_i \in U$ by f. Otherwise, because $S^{\natural} \models \xi^S[total]$ holds, at least one of its disjuncts must be satisfied by each $u^{\natural}$, i.e. $S^{\natural}, [w \mapsto u^{\natural}]$ must satisfy $\text{node}_{u_j}^S(w)$ for some u_j; thus f's definition will map $u^{\natural}$ to this u_j. Therefore, $f(u^{\natural})$ is well-defined.

In addition, every element $u_i \in U$ is assigned by f to some concrete element $u_i^{\natural} \in U^{\natural}$ such that $Z(v_i) = u_i^{\natural}$. Since $S^{\natural}, Z^{\natural}$ satisfies the sub-formula $\bigwedge_{k \neq j} \neg eq(v_k, v_j)$, all such elements $u_i^{\natural}$ are different. Therefore, $f(u^{\natural})$ is surjective.

Let p be a nullary predicate. Because $S^{\natural}$ satisfies $\xi_{nullary}^S$, it must satisfy each conjunct, in particular $S^{\natural} \models p^{\iota^S(p)()}$. Using Lemma 3.2.3 we get that $\iota^{S^{\natural}}(p)() \sqsubseteq \iota^S(p)()$.

Let $p \in P$ be a predicate of arity $r \geq 1$. Let $u_1^{\natural}, u_2^{\natural}, \ldots, u_r^{\natural} \in U^{\natural}$ and let us show that

$$\iota^{S^{\natural}}(p)(u_1^{\natural}, u_2^{\natural}, \ldots, u_r^{\natural}) \sqsubseteq \iota^S(p)(f(u_1^{\natural}), f(u_2^{\natural}), \ldots, f(u_r^{\natural})) \tag{B.2}$$

Let Z be an assignment such that $Z(w_i) = u_i^{\natural}$ for $i = 1, \ldots, r$. Because $S^{\natural} \models \xi^S[p]$, we conclude that $S^{\natural}, Z$ satisfies the body of Eq. (3.6). Consider the conjunct of the body with premise $\bigwedge_{j=1}^r \text{node}_{f(u_j^{\natural})}^S(w_j)$. By definition of f, $S^{\natural}, w_j \mapsto u_j^{\natural}$ satisfies $\text{node}_{f(u_j^{\natural})}^S(w_j)$ for all $j = 1, \ldots, r$, which means that the premise is satisfied by $S^{\natural}, Z$. Therefore, the conclusion must hold: $S^{\natural}, Z \models p^{\iota^S(p)(f(u_1^{\natural}), \ldots, f(u_r^{\natural}))}(w_1, \ldots, w_r))$ and the result follows from Lemma 3.2.3.

Lemma B.0.7 *For every 3-valued FO-identifiable structure S, and 2-valued structure $S^{\natural}$ such that $S^{\natural} \models F$ and $S^{\natural} \sqsubseteq S$, $S^{\natural} \models \xi^S$.*

Proof: Let $f \colon S^{\natural} \to S$ be a surjective function such that $S^{\natural} \sqsubseteq^f S$. Let $u_i^{\natural}$ be an arbitrary element such that $f(u_i^{\natural}) = u_i$. Define an assignment $Z^{\natural}$ such that $Z^{\natural}(v_i) = u_i^{\natural}$; $u_i^{\natural}$ must exist because f is surjective. Because S is FO-identifiable, by Definition 3.2.1 we conclude that for every $1 \leq i \leq n$, $S^{\natural}, Z^{\natural} \models \text{node}_{u_i}^S(v_i)$. Because f is a function, all $u_i^{\natural}$ are distinct elements, meaning that $S^{\natural}, Z^{\natural}$ satisfies the sub-formula $\bigwedge_{k \neq j} \neg eq(v_k, v_j)$.

Because f is a function, for every $u^{\natural}$ there is u such that $f(u^{\natural}) = u$. Then, by Definition 3.2.1, $S^{\natural}, [w \mapsto u^{\natural}] \models \text{node}_u^S(w)$, i.e., every assignment to w in $S^{\natural}$ satisfies some disjunct of ξ_{total}^S. That is $S^{\natural}$ satisfies ξ_{total}^S.

For every nullary predicate $p \in \mathcal{P}_0$, using Eq. (2.2) and Lemma 3.2.3, we conclude that $S^{\natural}$ satisfies $p^{\iota^S(p)()}$. Therefore, $S^{\natural}$ satisfies $\xi_{nullary}^S$.

Let $p \in P$ be a predicate of arity r. Let $u_1^{\natural}, \ldots, u_r^{\natural} \in U^{\natural}$ and let $Z^{\natural}$ be an assignment such that $Z^{\natural}(w_i) = u_i^{\natural}$. We shall show that $S^{\natural}, Z^{\natural}$ satisfy the body of Eq. (3.6). If the premise of the implication is not satisfied then the formula vacuously holds. Otherwise, $S^{\natural}, Z^{\natural} \models \text{node}_{u_i}^S(w_i)$ for all $i = 1, \ldots, r$. Then, by Definition 3.2.1, $f(u_i^{\natural}) = u_i$. Using Eq. (2.2) on f, we get $\iota^{S^{\natural}}(p)(u_1^{\natural}, \ldots, u_r^{\natural}) \sqsubseteq \iota^S(p)(f(u_1^{\natural}), \ldots, f(u_r^{\natural}))$, which means that $\iota^{S^{\natural}}(p)(u_1^{\natural}, \ldots, u_r^{\natural}) \sqsubseteq \iota^S(p)(u_1, \ldots, u_r)$ holds. By Lemma 3.2.3, we conclude that $S^{\natural}, Z^{\natural}$ satisfies $p^{\iota^S(p)(u_1, \ldots, u_r)}(w_1, \ldots, w_r)$.

Theorem 4.2.3 *For every 3-valued structure S, and a 2-valued structure $S^{\natural}$:*

$$S^{\natural} \in \gamma(S) \text{ iff } S^{\natural} \models \widehat{\gamma_{NP}}(S)$$

Proof: In Lemma B.0.8, we show that the if-direction holds, i.e., every concrete structure satisfying the NP-characteristic formula $\widehat{\gamma_{NP}}$ is indeed in $\gamma(S)$. In Lemma B.0.9 we show the only-if part.

Lemma B.0.8 *Let S be a logical structure with set of individuals $U = \{u_1, u_2, \ldots, u_n\}$. Then, for all $S^{\natural}$ such that $S^{\natural} \models \widehat{\gamma_{NP}}(S)$, $S^{\natural} \in \gamma(S)$.*

Proof: Let $S^{\natural} = \langle U^{\natural}, \iota^{\natural} \rangle$ be a concrete structure such that $S^{\natural} \models \widehat{\gamma}(S)$. We shall construct a surjective function $f \colon U^{\natural} \to U$ such that $S^{\natural} \sqsubseteq^f S$. Let $Z^{\natural}$ be an assignment such that $S^{\natural}, Z^{\natural} \models \varphi$ where φ is the body of ν^S without the existential quantifiers on sets. Let $Z^{\natural}(V_i) = U_i \subseteq U^{\natural}$. Consider the following definition:

$$f(u^{\natural}) = \{u_i \mid u^{\natural} \in U_i\} \tag{B.3}$$

45

$f(u^\natural)$ is a set of size at most 1 because the pair $S^\natural$, $Z^\natural$ satisfies the sub-formula $\nu^S_{disjoint}$. This insures that the sets $U_1, \ldots, U_n$ are disjoint, i.e., each concrete element belongs to at most one set. For simplicity, we say that $f(u^\natural) = u_i$, whenever $f(u^\natural) = \{u_i\}$.

We shall show that every concrete element is mapped by f to some element in U. Because $S^\natural$, $Z^\natural$ satisfies ξ^S_{total}, we conclude that every concrete element satisfies the formula $node^S_{u_i}(w)$ for some u_i. Also, $node^S_{u_i}(w)$ given in Definition 4.2.1 is a membership test in the set V_i; therefore, every concrete element must be a member of some set U_i. Thus, $u^\natural$ is mapped to $u_i \in U$, by the definition of f in Eq. (B.3). This shows that f is well-defined.

Because $S^\natural$, $Z^\natural$ satisfies $\models \nu^S_{non\_empty}[i]$ for $i = 1, \ldots, n$, it must be that every U_i contains at least one element, say $u_i^\natural$, that is mapped to u_i by f. Because the sets are disjoint, all such elements $u_i^\natural$ are different. Therefore, f is surjective.

Let p be a nullary predicate. Because $S^\natural$ satisfies $\xi^S_{nullary}$, it must satisfy each conjunct, in particular $S^\natural \models p^{\iota^S(p)()}$. Using Lemma 3.2.3 we get that $\iota^{S^\natural}(p)() \sqsubseteq \iota^S(p)()$.

Let $p \in P$ be a predicate of arity $r \geq 1$. Let $u_1^\natural, u_2^\natural, \ldots, u_r^\natural \in U^\natural$ and let us show that

$$\iota^{S^\natural}(p)(u_1^\natural, u_2^\natural, \ldots, u_r^\natural) \sqsubseteq \iota^S(p)(f(u_1^\natural), f(u_2^\natural), \ldots, f(u_r^\natural)) \tag{B.4}$$

Let $Z_1^\natural$ be an extension of assignment $Z^\natural$ such that $Z_1^\natural(w_i) = u_i^\natural$ for $i = 1, \ldots, r$. Because $S^\natural$, $Z^\natural \models \xi^S[p]$, we conclude that $S^\natural$, $Z_1^\natural$ satisfies the body of Eq. (3.6). Consider the conjunct of the body with premise $\bigwedge_{j=1}^r node^S_{f(u_j^\natural)}(w_j)$. By definition of f, $S^\natural, w_j \mapsto u_j^\natural$ satisfies $node^S_{f(u_j^\natural)}(w_j)$ for all $j = 1, \ldots, r$, which means that the premise is satisfied by $S^\natural$, $Z_1^\natural$. Therefore, the conclusion must hold: $S^\natural$, $Z_1^\natural \models p^{\iota^S(p)(f(u_1^\natural), \ldots, f(u_r^\natural))}(w_1, \ldots, w_r))$ and the result follows from Lemma 3.2.3.

Lemma B.0.9 *For every 3-valued structure S, and 2-valued structure $S^\natural$ such that $S^\natural \models F$ and $S^\natural \sqsubseteq S$, $S^\natural \models \xi^S$.*
Proof: Let $f \colon S^\natural \to S$ be a surjective function such that $S^\natural \sqsubseteq^f S$. Define an assignment $Z^\natural$ such that $Z^\natural(V_i) = U_i \subseteq U^\natural$ and $U_i = \{u_i^\natural \mid f(u_i^\natural) = u_i\}$.

Because f is a surjective function, there must exist at least one concrete element that is mapped to u_i by f. This element belongs to the set U_i. Therefore, $S^\natural$, $Z^\natural \models \bigwedge_{i=1}^n \nu^S_{non\_empty}[i]$.

Because f is a well-defined function, it maps each concrete element to exactly one element $u_i \in U$, which induces the set U_i. Therefore, a concrete element cannot belong to more than one set; hence $S^\natural$, $Z^\natural \models \bigwedge_{k \neq j} \nu^S_{disjoint}[k, j]$.

Because f is a function, f maps every concrete element to some element in U. Therefore, every concrete element belongs to some set, i.e., satisfies some disjunct of ξ^S_{total}. That is $S^\natural$, $Z^\natural \models \xi^S_{total}$.

For every nullary predicate $p \in \mathcal{P}_0$, using Eq. (2.2) and Lemma 3.2.3, we conclude that $S^\natural$, $Z^\natural$ satisfies $p^{\iota^S(p)()}$. Therefore, $S^\natural$, $Z^\natural \models \xi^S_{nullary}$.

Let $p \in P$ be a predicate of arity r. Let $u_1^\natural, \ldots, u_r^\natural \in U^\natural$ and let $Z_1^\natural$ be an extension of assignment $Z^\natural$ such that $Z_1^\natural(w_i) = u_i^\natural$. We shall show that $S^\natural$, $Z_1^\natural$ satisfy the body of Eq. (3.6). If the premise of the implication is not satisfied, then the formula vacuously holds. Otherwise, $S^\natural$, $Z_1^\natural \models node^S_{u_i}(w_i)$ for all $i = 1, \ldots, r$. Then, by Definition 4.2.1, $u_i^\natural$ belongs to the set U_i. The definition of U_i implies that $f(u_i^\natural) = u_i$. Using Eq. (2.2), we get $\iota^{S^\natural}(p)(u_1^\natural, \ldots, u_r^\natural) \sqsubseteq \iota^S(p)(f(u_1^\natural), \ldots, f(u_r^\natural))$ which

means $\iota^{S^\natural}(p)(u_1^\natural, \ldots, u_r^\natural) \sqsubseteq \iota^S(p)(u_1, \ldots, u_r)$. By Lemma 3.2.3 we conclude that $S^\natural, Z^\natural$ satisfies $p^{\iota^S(p)(u_1,\ldots,u_r)}(w_1, \ldots, w_r)$.

Lemma 7.3.1 *If* 3*-valued structure* $S = \langle U, \iota^S \rangle$ *over vocabulary* $\mathcal{P}$ *is* ICA *then:*

(i) *S is a bounded structure.*

(ii) *For each nullary predicate p, $\iota^S(p)() \in \{0, 1\}$.*

(iii) *For each element $u \in U$, and each unary predicate p, $\iota^S(p)(u) \in \{0, 1\}$.*

Proof: Let $S^\natural = \{U^\natural, \iota^{S^\natural}\}$ be a 2-valued structure, such that S is the canonical abstraction of $S^\natural$. Let $canonic\colon U^\natural \to U$ be the mapping that identifies S as the canonical abstraction of $S^\natural$.

(i) Show that S is a bounded structure. By Eq. (7.1), every abstract element represents concrete elements with the same canonical name. Thus, for two distinct abstract elements $u_0, u_1 \in U^S$, the canonical name of concrete elements represented by u_0 is different from the canonical name of concrete elements represented by u_1. Without loss of generality, assume that the canonical names differ in a unary predicate p, such that p evaluates to 0 on all concrete elements represented by u_0, and p evaluates to 1 on all concrete elements represented by u_1. From the join operation in Eq. (7.2), it follows that the value of p on u_0 must be 0 and the value of p on u_1 must be 1. This shows that, in general, every pair of distinct elements in S differs in a definite value of some unary predicate, proving that S is a bounded structure.

(ii) Let p be a nullary predicate. Show that $\iota^S(p)() \in \{0, 1\}$. By Eq. (7.2), $\iota^S(p)() = \sqcup\{\iota^{S^\natural}(p)()\} = \iota^{S^\natural}(p)()$. This means that p has the same value in S and $S^\natural$. Because $S^\natural$ is a concrete structure, the value of p must be definite.

(iii) Let p be a unary predicate and let $u \in U$. Show that $\iota^S(p)(u) \in \{0, 1\}$. Suppose that the opposite holds: $\iota^S(p)(u) = 1/2$. By Eq. (7.2), there exist two concrete elements, denoted by u_0 and u_1, such that $canonic(u_0) = u$ and $canonic(u_0) = u$, and p evaluates to 0 on u_0 and to 1 on u_1. Hence, these concrete elements have different canonical names and by Eq. (7.1) they cannot be mapped by $canonic$ to the same abstract element; this contradicts the supposition and hence $\iota^S(p)(u) \in \{0, 1\}$.

Lemma 7.3.2 *Every* 3*-valued structure S that is an* ICA *is FO-identifiable, where*

$$node_{u_i}^S(w) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}_1} p^{\iota^S(p)(u_i)}(w) \tag{B.5}$$

Proof: Let $S = \{U, \iota^S\}$ be a 3-valued structure that is *ICA*. We shall show that every element $u \in U$ is FO-identifiable using the formula defined in Eq. (7.5). Let $S^\natural = \{U^\natural, \iota^{S^\natural}\}$ be a 2-valued structure, such that S is the canonical abstraction of $S^\natural$, induced by a function $canonic$, and let $u^\natural \in U^{S^\natural}$. By Definition 7.2.1, we have to show that the following holds:

$$canonic(u^\natural) = u \iff S^\natural, [w \mapsto u^\natural] \models node_u^S(w)$$

Proof of the if direction: Suppose that $S^\natural, [w \mapsto u^\natural] \models \text{node}_u^S(w)$. Let $u_1 = canonic(u^\natural)$. For the sake of argument, assume that $u_1 \neq u$. S is an *ICA* and using Lemma 7.3.1(i) we get that S is a bounded structure. By Definition 3.3.1, there exists a unary predicate p that evaluates to different definite values on u and u_1. Without loss of generality, suppose that p evaluates to 0 on u and to 1 on u_1. This implies the following two facts. First, from property Eq. (7.2) of the definition of canonical abstraction, p also evaluates to 1 on all concrete values mapped to u_1 by *canonic*; in particular, p must evaluate to 1 on $u^\natural$. Second, recall that by assumption, each conjunct of node_u^S must hold, i.e., for each predicate $p \in \mathcal{P}_1$, $S^\natural, [w \mapsto u^\natural] \models p^{\iota^S(p)(u)}(w)$. Because p evaluates to 0 on u, we get from Definition 3.2.2 that $S^\natural, [w \mapsto u^\natural] \models p^0(w)$, which means $\iota^{S^\natural}(p)(u^\natural) = 0$ and a contradiction is obtained.

Proof of the only-if direction: Suppose that $canonic(u^\natural) = u$. Because S is an *ICA* by Lemma 7.3.1(iii) we know that all unary predicates have definite values in S. Let p be a unary predicate. Let $B \in \{1, 0\}$ be such that $\iota^S(p)(u) = B$. Because p has definite value B on u in S, by Eq. (7.2) it must have the same definite value B on all concrete nodes in $S^\natural$ that are mapped to u by *canonic*; in particular, on $u^\natural$: $\iota^{S^\natural}(p)(u^\natural) = B$. Therefore, using Definition 3.2.2, $S^\natural, [w \mapsto u^\natural] \models p^B(w)$, in other words, $S^\natural, [w \mapsto u^\natural] \models p^{\iota^S(p)(u)}(w)$. This holds for all unary predicates, and thus holds for their conjunction as well, i.e., for the formula node_u^S.

Theorem 7.3.5 *For every 3-valued structure S that is an* ICA *and a 2-valued structure $S^\natural$:*

$$S^\natural \in \gamma_c(S) \ \textit{iff} \ S^\natural \models \widehat{\gamma}_c(S)$$

Proof: In Lemma B.0.10, we show that the if-direction holds, i.e., a 3-valued structure S is the canonical abstraction of every concrete structure satisfying the characteristic formula $\widehat{\gamma}_c(S)$; in Lemma B.0.11 we show the other direction.

Lemma B.0.10 *Let S be an* ICA *with set of individuals $U = \{u_1, u_2, \ldots, u_n\}$. Let $\text{node}_{u_i}^S(w)$ be an arbitrary formula free in w, used in $\widehat{\gamma}_c$. Then, for all $S^\natural$ such that $S^\natural \models \widehat{\gamma}_c(S)$, S is a canonical abstraction of $S^\natural$.*

Proof: Let $S^\natural = \langle U^\natural, \iota^\natural \rangle$ be a concrete structure such that $S^\natural \models \widehat{\gamma}_c(S)$. We shall construct a surjective function $canonic \colon U^\natural \to U$ such that $S^\natural$ is a canonical abstraction of S. From Definition 7.3.3 it follows, in particular, that $S^\natural \models \xi^S$. Let $Z^\natural$ be an assignment such that $S^\natural, Z^\natural \models \varphi$ where $\varphi \overset{\text{def}}{=} \bigwedge_{i=1}^n \text{node}_{u_i}^S(v_i) \wedge \bigwedge_{k \neq j} \neg eq(v_k, v_j)$ (the first line of Eq. (3.7) without the existential quantification). Define the function $canonic \colon U^\natural \to U$ by:

$$canonic(u^\natural) = \begin{cases} u_i & \text{if } Z^\natural(v_i) = u^\natural \\ u_j & \text{if for all } i, \ Z^\natural(v_i) \neq u^\natural \text{ and } u_j \text{ is an arbitrary element such that} \\ & S^\natural, [w \mapsto u^\natural] \models \text{node}_{u_j}^S(w) \end{cases} \quad \text{(B.6)}$$

Let us show that every concrete element is mapped to some element in U. In the case that $Z(v_i) = u^\natural$, the concrete element $u^\natural$ is mapped to $u_i \in U$ by *canonic*. Otherwise, because $S^\natural \models \xi^S[total]$ holds, at least one of its disjuncts must be satisfied by each $u^\natural$, i.e., $S^\natural, [w \mapsto u^\natural]$ must satisfy $\text{node}_{u_i}^S(w)$ for some u_i; thus *canonic*'s definition will map $u^\natural$ to this u_i. Therefore, $canonic(u^\natural)$ is well-defined.

In addition, every element $u_i \in U$ is assigned by *canonic* to some concrete element $u_i^\natural \in U^\natural$ such that $Z(v_i) = u_i^\natural$. Since $S^\natural, Z^\natural$ satisfies the sub-formula $\bigwedge_{k \neq j} \neg eq(v_k, v_j)$, all such elements $u_i^\natural$ are different. Therefore, *canonic*$(u^\natural)$ is surjective.

We shall show that *canonic* satisfies Eq. (7.1) and Eq. (7.2); that is, *canonic* identifies S as the canonical abstraction of $S^\natural$.

First, let us show that Eq. (7.2) holds for the abstraction imposed by *canonic*, namely that a predicate p in S has the most precise abstract value w.r.t. the concrete values that it represents, as is imposed by *canonic*.

Because S is an *ICA*, all nullary predicates in S must have definite values, by Lemma 7.3.1(ii). $S^\natural$ satisfies $\xi_{nullary}^S$; therefore, by Definition 3.2.2, nullary predicates in $S^\natural$ must have the same definite values as in S; this shows that Eq. (7.2) holds for nullary predicates.

Because S is an *ICA*, all unary predicates in S must have definite values, by Lemma 7.3.1(iii). Let p be a unary predicate and let $u \in U$ be an individual of S such that $\iota^S(p)(u) = b$. We shall show that p has the same definite value b on all concrete elements mapped to u by *canonic*. Because the join of these values is also b, we will get that Eq. (7.2) holds for p and u. Recall that $S^\natural$ satisfies formula $\xi^S[p]$, hence each assignment to w satisfies the conjunct $node_u^S(w) \Rightarrow p^b(w)$ of $\xi^S[p]$. Let $u^\natural \in U^\natural$ be an individual of $U^\natural$ such that *canonic*$(u^\natural) = u$ and consider an assignment in which w is mapped to $u^\natural$. By the definition of *canonic*, this assignment satisfies $node_u^S(w)$, the premise of the conjunct. Therefore, it satisfies the conclusion, i.e., $S^\natural, [w \mapsto u^\natural]$ satisfies $p^b(w)$. Using Definition 3.2.2 we get that $\iota^{S^\natural}(p)(u^\natural) = b$.

Let p be a predicate of arity $r > 1$. If p has a definite value b in S on a tuple $u_1, \ldots, u_r$, $\xi^S[p]$ requires that p evaluates to the same definite value b on every concrete tuple $u_1^\natural, \ldots, u_r^\natural$ such that *canonic*$(u_i^\natural) = u_i$ (by the same argument as for unary predicates). Therefore, the join operation returns b as the most precise abstract value of p for these concrete tuples. Otherwise, if p evaluates to $1/2$ on $u_1, \ldots, u_r \in U$, there must be two tuples of elements in $U^\natural$, say $u_{01}^\natural, \ldots, u_{0r}^\natural$ and $u_{11}^\natural, \ldots, u_{1r}^\natural$, such that $S^\natural, [w_1 \mapsto u_{01}^\natural, \ldots, w_r \mapsto u_{0r}^\natural] \models \neg p(w_1, \ldots, w_r)$ and $S^\natural, [w_1 \mapsto u_{11}^\natural, \ldots, w_1 \mapsto u_{1r}^\natural] \models p(w_1, \ldots, w_r)$, because $S^\natural \models \tau^S[p]$. Thus, p evaluates to 0 on the first tuple and to 1 on the second tuple of the concrete structure; therefore, the most precise value obtained by the join operation on these values is $1/2$.

We shall show that *canonic* satisfies Eq. (7.1), i.e., it maps elements according to their canonical names. This involves showing two directions:

1. For the sake of contradiction, assume that there are two distinct elements $u_0^\natural, u_1^\natural \in U^\natural$ that have the same canonical name (meaning that for all $p \in \mathcal{P}_1$, $\iota^{S^\natural}(p)(u_0^\natural) = \iota^{S^\natural}(p)(u_1^\natural)$), but *canonic*$(u_0^\natural) \neq$ *canonic*$(u_1^\natural)$. Because S is a bounded structure, there must be unary predicate p that evaluates to 0 on *canonic*$(u_0^\natural)$ and to 1 on *canonic*$(u_1^\natural)$. As shown above, p evaluates to the same definite values in the concrete structure $S^\natural$: $\iota^{S^\natural}(p)(u_0^\natural) = 0$, and $\iota^{S^\natural}(p)(u_1^\natural) = 1$ and a contradiction is obtained.

2. For the sake of contradiction, assume that two concrete elements, denoted by $u_0^\natural, u_1^\natural \in U^\natural$, have different canonical names, but are mapped by *canonic* to the same same element in U: *canonic*$(u_0^\natural) =$ *canonic*$(u_1^\natural)$, denoted by u. By definition of *canonic*, $S^\natural, [w \mapsto u_i^\natural]$ satisfies $node_{canonic(u_i^\natural)}^S(w)$, for $i = 0, 1$, in other words $S^\natural, [w \mapsto u_i^\natural]$ satisfies $node_u^S(w)$.

Therefore, it satisfies each conjunct of *node* formula, i.e., for all p, $S^\natural$, $[w \mapsto u_i^\natural]$ satisfies $p^{iota^S(p)(u)}(w)$. From this and the fact that all unary predicates in S have definite values because S is an *ICA*, we conclude by Definition 3.2.2, that $\iota^{S^\natural}(p)(u_i^\natural) = \iota^S(p)(u)$. Therefore, $\iota^{S^\natural}(p)(u_0^\natural) = \iota^S(p)(u)$ and $\iota^{S^\natural}(p)(u_1^\natural) = \iota^S(p)(u)$, for all $p \in \mathcal{P}_1$. Therefore, $u_0^\natural$ and $u_1^\natural$ have the same canonical name and a contradiction is obtained.

Lemma B.0.11 *For every 3-valued structure S that is an* ICA *and 2-valued structure $S^\natural$ such that $S^\natural \models F$, such that S is the canonical abstraction of $S^\natural$, $S^\natural \models \tau^S$.*
Proof: Let $canonic \colon U^\natural \to U$ be the mapping that identifies S as the canonical abstraction of $S^\natural$. $canonic$ is a surjective function and possesses the properties in Eq. (7.1) and Eq. (7.2).

First, we show that $S^\natural \models \xi^S$. Let $u_i^\natural$ be an arbitrary element such that $canonic(u_i^\natural) = u_i$. Define an assignment $Z^\natural$ such that $Z^\natural(v_i) = u_i^\natural$; $u_i^\natural$ must exist because $canonic$ is surjective. Because S is FO-identifiable, by Lemma 7.3.2 we conclude that for every $1 \le i \le n$, $S^\natural, Z^\natural \models node_{u_i}^S(v_i)$. Because $canonic$ is a function, all the $u_i^\natural$ are distinct elements, meaning that $S^\natural, Z^\natural$ satisfies the sub-formula $\bigwedge_{k \ne j} \neg eq(v_k, v_j)$.

Because $canonic$ is a function, for every $u^\natural$ there is a u such that $canonic(u^\natural) = u$. Then, by Definition 7.2.1, $S^\natural, [w \mapsto u^\natural] \models node_u^S(w)$, i.e., every assignment to w in $S^\natural$ satisfies some disjunct of ξ_{total}^S. That is, $S^\natural$ satisfies ξ_{total}^S.

Because S is an *ICA*, nullary predicates have the same definite values in S and in $S^\natural$, by Lemma 7.3.1(ii). Therefore, by Definition 3.2.2, $S^\natural$ satisfies $p^{\iota^S(p)()}$, for every nullary predicate $p \in \mathcal{P}_0$, which means that $S^\natural$ satisfies $\xi_{nullary}^S$.

Let $p \in P$ be a predicate of arity r. Let $u_1^\natural, \ldots, u_r^\natural \in U^\natural$ and let $Z^\natural$ be an assignment such that $Z^\natural(w_i) = u_i^\natural$. We shall show that $S^\natural, Z^\natural$ satisfies the body of Eq. (3.6). Consider a conjunct of the body. If the premise of the implication in this conjunct is not satisfied, then the conjunct vacuously holds. Otherwise, $S^\natural, Z^\natural \models node_{u_i}^S(w_i)$ for all $i = 1, \ldots, r$. Then, by Lemma 7.3.2, $canonic(u_i^\natural) = u_i$. We have two cases to consider: (i) if $\iota^S(p)(u_1, \ldots, u_r) = b \in \{1, 0\}$ then by Eq. (7.2) $\iota^{S^\natural}(p)(u_1^\natural, \ldots, u_r^\natural) = b$, in other words, $S^\natural, Z^\natural$ satisfies $p^b(w_1, \ldots, w_r)$. (ii) if $\iota^S(p)(u_1, \ldots, u_r) = 1/2$ then by Definition 3.2.2, $p^{\iota^S(p)(u_1,\ldots,u_r)}(w_1, \ldots, w_r) = p^{1/2}(w_1, \ldots, w_r) = \mathbf{1}$, which holds for any assignment.

To complete the proof, we show that for every $p \in \mathcal{P}_r$ of arity $r > 1$, $\tau^S[p]$ holds. Let p be a predicate that evaluates to $1/2$ on a tuple $u_1, \ldots, u_r \in S$. Because S is an *ICA* $\iota^S(p)(u_1, \ldots, u_r) = 1/2$ means that the join operation in Eq. (7.2) yields $1/2$. By the definition of join as the least upper bound, and using the information order in Definition 2.3.1, we conclude that (i) $S^\natural$ must contain at least two distinct tuples; denoted by $u_{01}^\natural, \ldots, u_{0r}^\natural$ and $u_{11}^\natural, \ldots, u_{1r}^\natural$. Because $canonic(u_{ij}^\natural) = u_j$ for $i = 0, 1$ and $j = 1, \ldots, r$, by Lemma 7.3.2 we get that $S^\natural, [w \mapsto u_{ij}^\natural] \models node_{u_j}^S(w)$. Therefore, each tuple satisfies $\bigwedge_{j=1}^r node_{u_j}^S(w_j)$. (ii) p evaluates to 0 on the first tuple and 1 on the second tuple. This shows that $S^\natural \models \tau^S[p]$.

Appendix C

The Design of CoreC

This chapter is intended as a reference manual for the C code *Simplifier*.

C.1   Introduction

In this chapter we address a practical aspect of program analysis — dealing with real applications coded in C. Towards this end, we develop *Simplifier*, a tool that transforms a C program into an equivalent C program that uses only a small subset of the C language, called *CoreC*. This tool enables faster development of source-code analyzers; it was used by CSSV [DRS03] to check real C programs for string errors. The novelty of our work is the design of *CoreC* and the meaning-preserving translation rules from C to *CoreC*.

The main features of *CoreC* are:

- The *CoreC* syntax is compact and simple: (i) the number of different constructs that may appear in a *CoreC* program is relatively small; (ii) each construct has only one operation and simple arguments. This syntax makes it easy to specify the semantics of *CoreC* programs, because each statement performs a single operation.

- Due to the this, *CoreC* makes it easy to define and implement static analyses and source-to-source transformations of C programs. All that is needed to handle the full C language is to specify the semantics for *CoreC* constructs. The reason is that the translation rules preserve the semantics of the original program.

- *CoreC* is an executable intermediate representation, which enables easy debugging of *CoreC* itself.

- As opposed to many other intermediate representations, *CoreC* is portable across different machines and compilers.

The rest of the chapter is organized as follows: Section C.2 defines a subset of C called *CoreC*. Source-to-source translation rules for converting C into *CoreC* are described in Section C.3. Related work is described in Section C.5. Finally, the limitations of the current translation are listed in Section C.4.

C.2 The *CoreC* Language Subset

The *CoreC* language is a subset of C with the following restrictions:

- The only control-flow constructs are `goto` and an if-statement with `goto`s in the then- and else-clauses.

- Expressions cannot have side effects. In particular, assignments are statements. Expressions such as `+=`, `++`, and `comma` expressions are not allowed.

- There are two types of expressions: (i) Boolean comparison expressions used in if-conditions, and (ii) "Value" expressions (arithmetic, dereferences, and address expressions), which are used as L-values and R-values in assignments. Both types of expressions are rather limited. They cannot include more than one operation. The field operator (`a.b`) is not considered as an operation, and therefore can appear more than once in an expression. Dereference (`*`) is considered as an operation. The only exception to this rule is the address operator, which can be applied to a dereference: `&((*a).b)`. Moreover, at most one side of an assignment can include an operation. Value expressions cannot include comparison operators. Finally, logical connectives `&&`, `||`, and `!` are not allowed at all.

- Variables can only be initialized through regular assignments; initializations in declarations are not allowed. As a consequence, variables defined with `const` type qualifiers are not allowed, because they cannot be initialized, except in a declaration. However, `const` qualifiers can be safely removed, because we assume that the original program passes compilation, which guarantees that the program contains no assignments to `const` variables.

The syntax of the *CoreC* language is given in Table C.1. The syntax is in extended BNF. IDENTIFIER and CONSTANT represent valid identifier and constant symbols, accepted by standard C compilers. The syntax follows the definitions given in [Deg95] and [KR78]. Section C.5 compares *CoreC* syntax to other representations.

```
<CoreC> ::= (<functionDefinition> | <declaration>) *
<declaration> ::= <declarationSpecifiers> [<declarator> ]
<functionDefinition> ::= [<declarationSpecifiers> ]<declarator>
                    <declaration> * <compoundStatement>
<declarationSpecifiers> ::= (<storageClassSpecifier> | <typeSpecifier>)*
<storageClassSpecifier> ::= TYPEDEF | EXTERN | STATIC | AUTO | REGISTER
<typeSpecifier> ::= VOID | CHAR | SHORT | INT | LONG | FLOAT | DOUBLE | SIGNED |
                UNSIGNED | <structOrUnionSpecifier> | <enumSpecifier> | typeName
<structOrUnionSpecifier> ::= (STRUCT | UNION) [IDENTIFIER]{<structDeclaration>+ }
<structDeclaration> ::= <specifierList> <structDeclarationList>
<specifierList> ::= (<typeSpecifier>)+
<structDeclarationList> ::= <structDeclarator> (, <structDeclarator>)*
<structDeclarator> ::= <declarator> |
                    <declarator> : <constantExpression>
<enumSpecifier> ::= ENUM [IDENTIFIER][{<enumeratorList> }]
<enumeratorList> ::= <enumerator> (, <enumerator>)*
<enumerator> ::= IDENTIFIER [= <constantExpression> ]
<declarator> ::= [<pointer> ]<directDeclarator>
<pointer> ::= (∗)+
<directDeclarator> ::= IDENTIFIER |
                    <directDeclarator> ( [<declaration> ]* , [...]) // function
                    | <directDeclarator> [CONSTANT ]// array
```

Table C.1: The *CoreC* syntax.

```
<compoundStatement> ::= {<declaration>* <statement>* }
<statement> ::= <compoundStatement>
            |   IDENTIFIER : <statement>
            |   <Assignment>;
            |   IF ( <booleanExpression> ) goto IDENTIFIER; else goto IDENTIFIER;
            |   GOTO IDENTIFIER ;
            |   RETURN [<primaryExpression> ];
            |   <Call>;
            |   ; // empty statement
<Assignment> ::= <name> = <simpleExpression>
            |    <deref> = <primaryExpression>
<deref> ::= (∗ <name>)(.FIELD_NAME)*
<simpleExpression> ::= <primaryExpression>
                   |   (typeName) <primaryExpression> // cast
                   |   <primaryExpressoin> <binop> <primaryExpressoin>
                   |   <unop> <primaryExpressoin>
                   |   <deref>
                   |   <address>
                   |   <Call>
<address> ::= &<name> | &<deref>
<Call> ::= IDENTIFIER ( [<argList> ])
<argList> ::= primaryExpression [, primaryExpression ]*
<booleanExpression> ::= <primaryExpression> <relop> <primaryExpression>
<name> ::= IDENTIFIER ( .FIELD_NAME )*
<primaryExpression> ::= <name> | CONSTANT
<relop> ::= == | ! = | <= | >= | < | >
<binop> ::= + | − | ∗ | / | % | ^ | << | >> | | | &
<unop> ::= + | − | ∼
```

Table C.1: The *CoreC* syntax.

C.3   From C into *CoreC*

A C program is converted into a *CoreC* program by translating complex expressions into expressions with at most one operation. In certain cases, new temporary variables are introduced to store intermediate values. The resulting *CoreC* program is equivalent to the original C program in the sense that if the C program is defined according to ANSI-C, both C and *CoreC* programs provide the same output.

Example C.3.1 Fig. C.1 shows an example program `SkipLine` and its translation to *CoreC*. Note that double pointer indirections are translated into two sequential pointer indirections and that the `for` loop is translated into an `if-goto` loop.

```c
# define SIZE 5 void SkipLine(int NbLine,
                              char** PtrEndText)
{
  int indice;
  for (indice=0; indice<NbLine;
              indice++)
  {
    **PtrEndText = '\n';
    (*PtrEndText)++;
  }
  **PtrEndText = '\0';
}

void main()
{
  char buf[SIZE]="a\nb;
  char *r, *s;

  r = buf;
  SkipLine(1,&r);
  fgets(r,SIZE-1,stdin);
  s = r + strlen(r);
  SkipLine(1,&s);
}
```

(a)

55

```c
void SkipLine(int NbLine,
                    char** PtrEndText)
{
    int indice; char *tmp4;

    indice=0;
    begin_loop:
      if (indice>=NbLine)
      goto end_loop;
      else goto loop_body;
      loop_body:{
        char *tmp1; char *tmp2; char *tmp3;
        tmp1 = *PtrEndText
        *tmp1 = '\n';
        tmp2 = *PtrEndText;
        tmp3 = tmp2 + 1;
        *PtrEndText = tmp3;
      }
      indice = indice + 1;
      goto begin_loop;
    end_loop:
      tmp4 = *PtrEndText
      *tmp4 = '\0';
}

void main()
{
    char buf[5];
    char *r; char *s;
    char **tmp4; char **tmp7;
    int tmp5; int tmp6;

    buf[0]='a';
    buf[1]='\n';
    buf[2]='b';
    buf[3]='\0';
    buf[4]='\0';

    r = buf;
    tmp4 = &r;
    SkipLine(1,tmp4);
    tmp5 = SIZE-1;
    fgets(r,tmp5,stdin);
    tmp6 = strlen(r);
    s = r + tmp6;
    tmp7 = &s;
    SkipLine(1,tmp7);
}
```

(b)

Figure C.1: Example: (a) SkipLine, a string-manipulation function from EADS Airbus with a toy main program; (b) the result of translation SkipLine program to CoreC.

This section provides a high-level description of the algorithm for converting C to *CoreC*. It is based on the algorithm for translating abstract syntax into abstract machine code, used in compilers, as described in Chapter 7 of [App98]. To provide a front-end-independent description of the translation process, we assume that an abstract syntax tree (AST) of the input C program is given in a form that allows top-down traversal and property retrieval. At the end of the traversal of the input AST, an equivalent AST is generated, containing only legal *CoreC* constructs. The translation process recursively traverses each sub-expression of an AST node, generates equivalent *CoreC* constructs for each subexpression, and combines them.

C.3.1 Internal structure

To define the translation inductively, the algorithm uses an internal structure named *CoreC* Unit (CCU). CCU accumulates declarations and statements created during traversal. CCU also contains sub-expressions to be combined in a higher level. The CCU structure has a similar functionality to the Tr_exp structure in [App98]. The differences are due to the complexity of C language, compared to Tiger. As in Tr_exp, there are several kinds of CCU structures (see Table C.2):

CCUExp - *CoreC* expression.

CCUBool - Boolean expression, contains only one relational or equality operator, applied to primary_expressions.

CCUArgList - list of *CoreC* expressions, to be used as arguments to a *CoreC* function call.

CCUCond - if a Boolean expression can be used either as a value or in a conditional jump (i.e. the translation depends on the context), the decision is delayed and conditional unit CCUCond is generated. The Boolean expression is used as a condition in if-statement. This if-statement is added to the statement list in the basic part of CCU. Two new labels are generated for true and false clauses of this if-statement. Each label is added to the suitable list of labels in CCUCond. Now, it is the responsibility of the caller to "patch" the labels at the right places (either where if-branches start or where the value of the Boolean expression is assigned to a new temporary, for future use in an expression).

CCUStmt - holds two list of labels, referenced by the statements of the unit, but not yet located. When the translation of current unit is finished, the labels will be added before and after the statements of this unit. This "patch" is handy for translating "break" and "continue" statements used within loops and switch.

C.3.2 Value/Address computations

Sometimes we will have CCU of one kind and we will need to convert it to an equivalent unit of another kind. For example, $f = (a > b) || (c < d)$ requires the conversion of CCUCond into CCUExp, so that the "true" or "false" results of computing $||$ can be stored into "f". In general, the translation process may convert a value computation into an address computation and visa versa. To convert a "conditional" into a value, we invent a new temporary variable and assign "1" to it. This assignment is added to the statement list of current unit, preceded by labels from the "true" list of CCUCond. We handle the "false" branch similarly. The result is just the temporary.

| CCU { CoreDeclarations d; CoreStatements s; } |
| CCUStmt extends CCU { LabelList start_labels, end_labels; } |
| CCUExp extends CCU { CoreExpression exp; } |
| CCUBool extends CCU { CoreBoolean exp; } |
| CCUArgList extends CCU { CorePrimaryList list; } |

Table C.2: CCU (*CoreC* Unit) is a structure used for translation

C.3.3 L-value and R-value

Translation of an expression depends on whether it is an L-value or an R-value. When an expression is an R-value it denotes a value fetched from a location (i.e., the contents). When an expression is an L-value it denotes a location (i.e., an address) in which a value can be stored (for example, when the expression appears on the left-hand side of an assignment). Replacing an L-value expression by a temporary variable may change the meaning of the program (for example, see the use of $a.f$ in $a.f = b + c$, $b = \&(a.f)$). Therefore, we need special translation rules for expressions that serve as L-values.

C.3.4 Translation rules

Main modules of the algorithm are listed in Table C.3. Each module takes an arbitrary C construct and produces a *CoreC* unit, which is semantically equivalent to the input construct. This *CoreC* unit contains *CoreC* declarations and statements generated during the simplification process of the input construct, including new variables and their initializations.

Informal pseudo-code description of the translation algorithm can be found in [Yor02]. In that document, some informal naming conventions are used to save space: *exp* denotes an expression, *stmt* denotes a statement, *"e.f"* denotes symbolic structure of expression, + denotes concatenation of elements of the same type into a list. Also note the function IsCore(exp, core_type), that checks whether the expression exp is a legal *CoreC* expression of type core_type, as defined in Table C.1.

CCUStmt simplify_stmt(stmt); *Input:* C statement.
CCUExp simplify_lvalue(exp); *Input:* Arbitrary C expression denoting an L-value Generates an error when exp does not denote an L-value.
CCU simplify_exp(exp); *Input:* Arbitrary C expression denoting an R-value
CCU simplify_bool_exp(exp); *Input:* C expression representing a Boolean value Generates either CCUCond or CCUExp.

Table C.3: Main modules of the translation algorithm.

CCUStmt simplify_decl(d); *Input:* an arbitrary C declaration with/without initialization. *Output: CoreC* declaration equivalent to the input declaration, with the following exceptions: (a) removing "const" qualifiers; (b) initialization is converted into an assignment, which is added to the statement list.
CCUExp simplify_post(exp, bin_op); *CCUExp simplify_pre(expbin_op);* *Input:* C-expression denoting post-inc (x++) or post-dec(x–) operation and a suitable binary operator (+ or -). Generates an error when exp does not denote an l-value.
*CCUExp simplify_unary(exp, un_op, SymbolTable *scope);* *Input:* C-expression denoting the operand of a unary operator un_op from the following list: $\sim$e $-$e $+$e
CCUExp simplify_binary(e1, e2, bin_op); *Input:* C-expressions e1, e2 denoting operands of a binary operation: $+ - */\%|\& <<>>$ according to bin_op param.
CCUExp simplify_op_assign(e1, e2, bin_op); *Input:* C expressions e1, e2 of and assignment "e1 op e2", where op is one of the following: +=, -=, *=, /=, bin_op denotes binary operator suitable for the assignment. Output: Expression equivalent to the value returned by the assignment after the evaluation, equivalent *CoreC* assignment, and additional declarations and statements created in the simplifier process.

Table C.3: Main modules of the translation algorithm.

C.4 Limitations of the Current Translation

We implemented the translation algorithm described above, using Microsoft AST Toolkit [TSBTG] as a frontend and a backend. This tool builds an abstract syntax tree of a program and allows traversal and modification of the tree. It also produces an output of the (modified) abstract syntax tree in the form of a C program.

Explicit casting. Signed/unsigned warning in the simplified code that is not produced by the original code. For example, consider the program fragment on Fig. C.2(a) and it's translation on Fig. C.2(b). The last line of the translation produces a signed/unsigned mismatch.

Initialization lists - limited treatment for (i) casting of list elements when types do not match; (ii) nested initialization lists with missing elements. Simplifier may fail in some cases even if the initialization is correct according to ANSI C. It is recommended to correct the original code by adding explicit casting, missing elements, and nested lists for initialization of structure fields.

```
                              int n;  unsigned int a, b, tmp;
unsigned int a, b;            tmp = a/b;
if (n < a/b) ...              if (n < tmp) ...


          (a)                               (b)
```

Figure C.2: Example: (a) a program fragment and (b) its translation to *CoreC*.

Erroneous output might be produced for the following features:

- variable-length parameter lists
- `setjmp`/`longjmp`
- signals
- system calls such as `system`, `exec`, `abort`, etc.

sizeof expression is replaced by a constant, which may be machine dependent.

Optimizations: The current translation rules do not produce minimal size *CoreC* code. It is possible to minimize the number of new variables, labels, and statements introduced during translation, by reusing temporaries, for example.

C.5   Related Work

There are numerous tools that convert C to an intermediate representation (IR), many of which are machine independent. Most compilers convert C code to IR before generating assembly. Program-analysis tools usually use a front-end to translate the program to some IR suitable for analysis and optimizations. Such tools also contain a back-end to convert IR back to C. The output is often generated in a canonical form, more limited than general C syntax (for example, SUIF, SimpleC).

C.5.1   IR-C

Similar to *CoreC*, IR-C (see [Leu02]) retains the executablity of the output by representing the IR in C syntax. Another similarity is the attempt to simplify C expressions into three-address instructions. IR-C produces lower-level code than *CoreC*.

C.5.2   CIL

The CIL (C Intermediate Language) in [NMRW02] is a high-level intermediate representation for C which has many features common to *CoreC*. However, *CoreC* conceptually defers from CIL. In general, *CoreC* is more structured and performs deeper simplification of the input program. The main differences between *CoreC* and CIL are listed in Table C.4.

CoreC	CIL
Converts initialization of a variable in its declaration to a regular assignment, which requires removing *const* modifiers from all types. This conversion is applied to global variables as well as locals. Therefore, initialization of global variables is performed in a meta-main function implanted in the beginning of the original main function.	Applies the same conversion only for local variables, whereas global variables can be initialized at declaration.
Converts initialization lists and array initializations to assignments, including assignments to missing elements.	Applies these conversions only for local variables.
Always removes operators ? → [].	Index operator is removed only for pointers, not for arrays.
Converts static variables in functions into global variables.	—
—	Adds missing function declarations.
Supports only *goto*-statement and *if*-statement with goto's on its branches.	Supports additional control flow constructs: *while*, *break*, *continue*, *switch*.
Transforms expressions with side-effects into sequences of statements.	Retains ++ operator.
Supplies original code line	—
Does not allow nesting, except for fields.	Allows nested expressions (both fields and operators), which reduces the number of temporary variables and new statements introduced by the translation.
Retains scopes.	Removes scopes inside functions (requires renaming)
—	Removes declarations for unused entities (types, variables).
Always converts expressions with &&, ‖ into explicit if-statements.	Treatment of boolean expressions and conditions: (i) convert expression with &&, ‖ operators into explicit if-statement. (ii) duplicate expressions and statements that depend on boolean operation. (iii) to avoid excessive duplication use a *goto* for statements that have more than 5 instructions.

Table C.4: Differences between the features supported by CIL and *CoreC*.

60

CoreC	CIL
Has only a canonic form of if-statement: if () goto L1 else goto L2.	Allows arbitrary statements as if-clauses.
Implemented under Windows with Microsoft Visual Studio compiler.	Implemented for UNIX systems with GCC and has support for special GCC options.

Table C.4: Differences between the features supported by CIL and *CoreC*.

C.6   Complexity

The number of new variables introduced during the translation process is linear in the number of operations in the input program. For each new variable, exactly one assignment is performed. The only exception is in the translation of $\&\&$ and $\|$ operators to an expression, where the new variable that denotes the value of the expression is assigned in both branches of an if-statement. The number of new labels is linear in the number of loop (while, do-while, for) and conditional (if, case) constructs in the program. The translation of initialization lists and array initializations is linear in the number of initializers.