

COUNTEREXAMPLE-GUIDED VERIFICATION OF IMPERATIVE PROGRAMS AGAINST FUNCTIONAL SPECIFICATION

INDRAJIT BANERJEE



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
JUNE 2023

COUNTEREXAMPLE-GUIDED VERIFICATION OF IMPERATIVE PROGRAMS AGAINST FUNCTIONAL SPECIFICATION

by

INDRAJIT BANERJEE

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of
Master of Science (Research)

to the



Indian Institute of Technology Delhi

JUNE 2023

Certificate

This is to certify that the thesis titled **Counterexample-Guided Verification of Imperative Programs Against Functional Specification** being submitted by **Mr. Indrajit Banerjee** for the award of **Master of Science (Research) in Computer Science and Engineering** is a record of bona fide work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Sorav Bansal
Microsoft Chair Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi - 110016

Acknowledgements

I would like to sincerely thank my thesis supervisor Prof. Sorav Bansal for his continuous support during my study and research. His guidance, patience, motivation and long discussions provided a strong platform with clear visibility and research direction.

Besides my advisor, I would like to thank the following members of my Student Research Committee (SRC) for their insightful comments and encouragement that helped me to widen my research from various perspectives:

Prof. Sanjiva Prasad (Dept. of CSE, IIT Delhi)

Prof. Kumar Madhukar (Dept. of CSE, IIT Delhi)

Mr. Akash Lal (Microsoft Research Lab, India)

I am grateful to our research group members: Abhishek Rose, Shubhani at IIT Delhi for their help and motivating discussions on various topics related to my research.

Indrajit Banerjee

Abstract

We describe an algorithm capable of checking equivalence of two programs that manipulate recursive data structures such as linked lists, strings, trees and matrices. The first program, called specification, is written in a succinct and safe functional language with algebraic data types (ADT). The second program, called implementation, is written in C using arrays and pointers. Our algorithm, based on prior work on counterexample guided equivalence checking, automatically searches for a sound equivalence proof between the two programs.

We formulate an algorithm for discharging proof obligations containing relations between recursive data structure values across the two diverse syntaxes, which forms our first contribution. Our proof discharge algorithm is capable of generating falsifying counterexamples in case of a proof failure. These counterexamples help guide the search for a sound equivalence proof and aid in inference of invariants. As part of our proof discharge algorithm, we formulate a program representation of values. This allows us to reformulate proof obligations due to the top-level equivalence check into smaller nested equivalence checks. Based on this algorithm, we implement an automatic (push-button) equivalence checker tool named S2C, which forms our second contribution.

S2C is evaluated on implementations of common string library functions taken from popular C library implementations, as well as implementations of common list, tree and matrix programs. These implementations differ in data layout of recursive data structures as well as algorithmic strategies. We demonstrate that S2C is able to establish equivalence between a single specification and its diverse C implementations.

Keywords: *Equivalence checking; Bisimulation; Recursive Data Structures; Algebraic Data Types;*

Contents

Abstract	v
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 A Motivating Example	2
1.2 Our Contributions	7
1.3 Outline of the Thesis	8
2 Preliminaries	11
2.1 The Specification Language : Spec	11
2.2 Abstract Representation of Programs	13
2.2.1 Conversion of Programs to Intermediate Representation	13
2.2.2 IR Instructions	16
2.2.3 Control-Flow Graph Representation	18
2.3 Equivalence Definition	20

2.3.1	Constraining Inputs to C	21
2.4	Bisimulation Relation	22
2.4.1	Well-formedness of Product-CFG	23
2.5	Recursive Relation	24
2.6	Proof Obligations	25
3	Proof Discharge Algorithm through Illustative Examples	27
3.1	Properties of Proof Discharge Algorithm	27
3.1.1	Soundness of Proof Discharge Algorithm	27
3.1.2	Conjunctive Recursive Relation Property of Proof Obligations	28
3.2	Iterative Unification and Rewriting Procedure	29
3.2.1	Atomic Expression	29
3.2.2	Expression Trees	30
3.2.3	Unification Procedure	31
3.2.4	Unification under Rewriting	32
3.2.5	Decomposition of Recursive Relations	33
3.3	Categorization of Proof Obligations	34
3.3.1	k -unrolling of Recursive Relations	34
3.3.2	Categorization Procedure	35
3.4	Handling Type I Proof Obligations	35
3.5	Handling Type II Proof Obligations	37
3.5.1	Depth of ADT Values	39
3.5.2	Approximation of Recursive Relations	39
3.5.3	Reduction of Approximate Recursive Relations	40

3.5.4	Summary of Type II Proof Discharge Algorithm	41
3.6	Handling Type III Proof Obligations	43
3.6.1	LHS-to-RHS Substitution and RHS Decomposition	43
3.6.2	Deconstruction Programs for Lifted Values	45
3.6.3	Checking Bisimulation between Deconstruction Programs	47
3.6.4	Points-to Analysis	49
3.6.5	Transferring Points-to Information to Decons-PCFG	50
3.6.6	Summary of Type III Proof Discharge Algorithm	52
3.7	Overview of Proof Discharge Algorithm	52
4	Spec-to-C Equivalence Checker	57
4.1	Points-to Analysis	58
4.2	Counterexample-guided Product-CFG Construction	60
4.2.1	Correlating Pathsets	62
4.2.2	Best-First Ranking of Partial Product-CFGs	63
4.2.3	Correlation in the Presence of Function Calls	64
4.3	Invariant Inference and Counterexample Generation	65
4.4	More on Proof Discharge Algorithm	67
4.4.1	Handling Proof Obligations On Pathsets	67
4.4.2	Canonicalization Procedure	68
4.4.3	Unification Procedure	70
4.4.4	Iterative Unification and Rewriting Procedure	71
4.4.5	Decomposition Procedure for Recursive Relations	72
4.4.6	Reduction Procedures for Approximate Recursive Relations	73

4.4.7	SMT Encoding of First Order Logic Formula	77
4.4.8	Reconciliation of Counterexamples	79
4.4.9	Value Tree Representation	80
4.4.10	Conversion of Expressions to their Value Trees	84
4.4.11	Applications of Value Trees	91
5	Evaluation	97
5.1	Experiments	98
5.1.1	String	99
5.1.2	List	104
5.1.3	Tree	105
5.1.4	Matrix	106
5.2	Results	107
5.3	Limitations	107
6	Conclusion	111
	Bibliography	113
	Biography	121

List of Figures

1.1	Spec and C programs each constructing a Linked List.	3
1.2	IRs for the Spec and C <code>mk_list</code> procedures in figs. 1.1a and 1.1b respectively. . .	4
1.3	CFG representation of Spec and C IRs shown in figs. 1.2a and 1.2b for the <code>mk_list</code> procedures in figs. 1.1a and 1.1b respectively.	5
1.4	Product-CFG between the CFGs in figs. 1.3a and 1.3b. Figure 1.4b contains the corresponding node invariants for the product-CFG.	6
2.1	Spec and C programs along with corresponding IRs for the <code>is_empty</code> procedures.	12
2.2	Simplified expression grammar of Spec language	13
2.3	Spec and C programs for traversing a Linked List along with corresponding IRs for the <code>sum_list</code> procedures.	14
2.4	CFG representation of Spec and C IRs in figs. 2.1c and 2.1d for the <code>is_empty</code> procedures in figs. 2.1a and 2.1b respectively.	18

2.5	CFG representation of Spec and C IRs in figs. 2.3b and 2.3d for the <code>sum_list</code> procedures in figs. 2.3a and 2.3c respectively.	18
2.6	Product-CFG and its associated node invariants for CFGs in figs. 2.4a and 2.4b. .	21
2.7	Product-CFG between the CFGs in figs. 2.5a and 2.5b. Figure 2.7b contains the corresponding node invariants for the product-CFG.	22
3.1	Expression trees corresponding to three canonical <code>List</code> expressions used in the context of unification.	30
3.2	Product-CFG between the CFGs in figs. 1.3a and 1.3b. Figure 1.4b contains the corresponding node invariants for the product-CFG.	36
3.3	Product-CFG between the CFGs in figs. 2.5a and 2.5b. Figure 2.7b contains the corresponding node invariants for the product-CFG.	37
3.4	Expression trees of three values, each of type <code>List</code> , <code>Tree</code> and <code>Matrix</code> respectively. The depths are shown as superscripts for each node in the trees.	38
3.5	IR and CFG representation of deconstruction program based on the lifting constructor $\text{Clist}_m^{\text{lnode}}$ defined in eq. (2.2). The edge $D5 \rightarrow D6$ contains a recursive function call. In fig. 3.5b, the square boxes show the transfer functions for the deconstruction program. The dashed edges represent the recursive function call in the CFG representation as shown in fig. 3.5b.	45

3.6	Decons-PCFG and its associated node invariants for the deconstruction programs based on $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)$ and $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(l_C)$ respectively.	46
3.7	Points-to invariants at $\mathbf{c5}$ of \mathcal{C} in fig. 1.2b. Figure 3.7b shows the graphical representation of the relations in fig. 3.7a. Each node represents a \mathcal{C} pseudo-register or a memory region in \mathfrak{m} . An edge $A \rightarrow R$ represents the condition that A (or objects in A in case of a memory region) may point to the memory region R	50
3.8	Pseudocode of the algorithm responsible for discharging proof obligations containing recursive relations	53
4.1	Pseudocode of the best-first search algorithm responsible for incremental construction of a product-CFG between \mathcal{S} and \mathcal{C}	61
4.2	Transfer function f_e and predicate grammar \mathbb{G} for invariant inference dataflow analysis in table 4.2. Given invariants ϕ_s and counterexamples Γ_s at node s , f_e returns the updated invariants ϕ_d and counterexamples Γ_d at node d . strongestInvCover (Γ) computes the strongest invariant cover for counterexamples Γ . exec_e (Γ) (concretely) executes counterexamples Γ over edge e . prove (P) (in fig. 3.8) discharges a proof obligation P , and returns either True or False (Γ).	66
4.3	Type trees for the ADTs List , Tree and Matrix respectively.	81

4.4	Three type trees for List ADT. Figure 4.4a shows the type tree for the canonical form of List . Figure 4.4b is obtained by peeling the back-edge $[1 \rightarrow \text{tail}]$ in fig. 4.4a. Figure 4.4c is obtained by unrolling the back-edge $[1 \rightarrow \text{tail}]$ in fig. 4.4a or by peeling the back-edge $[2 \rightarrow \text{LCons}]$ in fig. 4.4b respectively.	82
4.5	Value trees of three List -typed expressions	83
4.6	Construction of $\mathcal{V}(e_1 \odot e_2)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$. \odot represents an arbitrary scalar operator.	85
4.7	Construction of $\mathcal{V}(\text{LCons}(e_1, e_2))$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$. $\mathcal{R}_{\text{LNil}}$ represents an arbitrary value tree corresponding to the product-type (in \mathbb{T}) associated with LNil . . .	85
4.8	Construction of $\mathcal{V}(e \text{ is } \text{MCons})$ from $\mathcal{V}(e)$. The dashed edge represents the (possibly empty) set of backedges originating in v_2 that terminates at 0.	86
4.9	Construction of $\mathcal{V}(e.\text{cols})$ from $\mathcal{V}(e)$. Similar to type trees, unroll $[1 \rightarrow \text{cols}]$ represents the operation of hoisting one iteration of the cycle $0 \rightarrow 1 \rightarrow 0$	87
4.10	Construction of $\mathcal{V}(\text{if } c \text{ then } \text{MNil} \text{ else } \text{MCons}(e_1, e_2))$ from $\mathcal{V}(c)$, $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$	88
4.11	Construction of $\mathcal{V}(\text{Clist}_{\text{m}}^{\text{lnode}}(p))$ from $\mathcal{V}(p)$. The process involves assuming $v_{\omega_1}^{\text{Clist}}$ to be the value tree of $\text{Clist}_{\text{m}}^{\text{lnode}}(\omega_1)$, followed by expansion of $v_{\omega_1}^{\text{Clist}}$ using the definition of $\text{Clist}_{\text{m}}^{\text{lnode}}$ (in eq. (2.2)) and, finally folding the tree-edge $[1 \rightarrow \text{tail}]$ incident on the self-referential subtree $v_{\omega_1}^{\text{Clist}}$ into a backedge.	89
4.12	Construction of $\mathcal{V}(m)$ for a Matrix variable m . The process is identical to the construction of value trees for lifted expressions as shown in fig. 4.11.	90

4.13	Value trees for a <code>List</code> variable l and a lifted expression $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)$ respectively.	92
4.14	Value trees of $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(1_C)$ and $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(1_C)$ along with the invariants table. .	95
5.1	Figures 5.1a and 5.1b show the (abstracted) IRs for the Spec specification and a generic nul-terminated array based C implementation of <code>strchr</code> . Figure 5.1c shows the product-CFG representing a bisimulation relation between figs. 5.1a and 5.1b. The node invariants for the product-CFG in fig. 5.1c are given in fig. 5.1d.	100
5.2	Figure 5.2a shows the (abstracted) IR for the Spec specification of <code>strlen</code> . Figures 5.2b and 5.2c show the (abstracted) IRs for two C implementations of <code>strlen</code> . Figure 5.2b is a generic implementation using a nul-terminated array to represent a string, whereas fig. 5.2c is an optimized implementation with a chunked linked list memory layout for a string.	102
5.3	Product-CFGs and their node invariants representing bisimulation relations between the specification fig. 5.2a and its two implementations in figs. 5.2b and 5.2c respectively.	103

List of Tables

4.1	Dataflow Formulation of the Points-to Analysis	58
4.2	Dataflow Formulation of the Invariant Inference Algorithm	65
5.1	String lifting constructors and their definitions.	98
5.2	List lifting constructors and their definitions.	104
5.3	Tree lifting constructors and their definitions.	105
5.4	Matrix and auxiliary List lifting constructors and their definitions.	106
5.5	Time taken by S2C for successful equivalence checks between specifications and their C implementations. Additionally, we show the minimum under- and over-approximation depths at which these equivalence checks succeed.	108

Chapter 1

Introduction

The problem of equivalence checking between a functional specification and an implementation written in a low level imperative language such as C has been of major research interest. On one side, the functional programming model closely resembles mathematical functions, which allows for comparatively easier verification of algorithmic correctness. On the other hand, a low level imperative language such as C trades the safer abstractions of a functional language for proximity to the machine language resulting in (usually) significantly faster executables, albeit at the cost of a substantially higher possibility of algorithmic errors. Being able to establish equivalence between the two abstractions would allow the user to take advantage of both worlds – (a) easier proof of functional correctness and (b) more efficient executables. The applications of such an equivalence checker would include (a) program verification, where the equivalence checker is used to verify that the C implementation behaves according to the specification and (b) translation validation, where the equivalence checker attempts to generate a proof of equivalence across the transformations (and translations) performed by an optimizing compiler.

The verification of a C implementation against its manually written functional specification through manually-coded refinement proofs has been performed extensively in the seL4 micro-kernel [31]. Frameworks for program equivalence proofs have been developed in interactive theorem provers like Coq [20] where correlations and invariants are manually identified during proof

codification. On the other hand, programming languages like Dafny [33] offer automated program reasoning for imperative languages with abstract data types such as sets and arrays. Such languages perform automatic compile-time checks for manually-specified correctness predicates through SMT solvers. Additionally, there exists significant prior work on translation validation [39, 52, 49, 51, 32, 54, 55, 44, 53, 34, 30, 35, 12, 47, 18, 27, 46, 38] across multiple programming languages with similar models of computation. In most of these applications, soundness is critical, i.e., if the equivalence checker determines the programs to be equivalent, then the programs are indeed equivalent and evidently have equivalent runtime observable behaviour. On the other hand, a sound equivalence checker may be incomplete and fail to prove equivalence of a program pair, even if they were equivalent.

In this work, we present S2C, a *sound* algorithm to automatically search for a proof of equivalence between a functional specification and its optimized C implementations. We will demonstrate how S2C is capable of proving equivalence of multiple equivalent C implementations with vastly different (a) data layouts (e.g. array, linked list representations for a *list*) and (b) algorithmic strategies (e.g. alternate algorithms, optimizations) against a *single* functional specification. This opens the possibility of regression verification [50, 24], where S2C can be used to automate verification across software updates that change memory layouts of data structures.

1.1 A Motivating Example

We start by restricting our attention to programs that construct, read, and write to recursive data structures. In languages like C, pointer and array based implementations of these data-structures are prone to safety and liveness bugs. Similar recursive data structures are also available in safer functional languages like Haskell [36], where algebraic data types (ADTs) [15] ensure several safety properties. We define a minimal functional language, called Spec, that enables the safe and succinct specification of programs manipulating and traversing recursive data structures. Spec is equipped with ADTs as well as boolean (`bool`) and fixed-width bitvector (`i<N>`) types.

We motivate our work by considering example Spec and C programs. The major hurdles of our


```

A0: type List = LNil | LCons (val:i32, tail:List).
A1:
A2: fn mk_list_impl (n:i32) (i:i32) (l:List) : List =
A3:   if i ≥u n then l
A4:   else make_list_impl(n, i+1i32, LCons(i, l)).
A5:
A6: fn mk_list (n:i32) : List = mk_list_impl(n, 0i32, LNil).

```

(a) Spec program

```

B0: typedef struct lnode {
B1:   unsigned val; struct lnode* next;
B2: } lnode;
B3:
B4: lnode* mk_list(unsigned n) {
B5:   lnode* l = NULL;
B6:   for (unsigned i = 0; i < n; ++i) {
B7:     lnode* p = malloc(sizeof lnode);
B8:     p->val = i; p->next = l; l = p;
B9:   }
B10:  return l;
B11: }

```

(b) C program with malloc()

Figure 1.1: Spec and C programs each constructing a Linked List.

approach are listed alongside an informal discussion of our proposed solutions. We state our primary contributions in section 1.2 and finish with the organization of the rest of the thesis in section 1.3.

Figures 1.1a and 1.1b show the construction of lists in Spec and C respectively. The `List` ADT in the Spec program is defined at line A0 in fig. 1.1a. An empty `List` is represented by the *data constructor* `LNil`, whereas a non-empty list uses the `LCons` constructor to combine its first value (`val:i32`) and the remaining list (`tail:List`). The inputs to a Spec procedure (aka function) are its well-typed arguments, which may include recursive data structure (i.e. ADT) values. The inputs to a C procedure are its explicit arguments and the implicit state of program memory at

<pre> S0: List mk_list (i32 n) { S1: List l := LNil; S2: i32 i := 0_{i32}; S3: while ¬(i ≥_u n): S4: l := LCons(i, l); S5: i := i + 1_{i32}; S6: return l; SE: }</pre>	<pre> C0: i32 mk_list (i32 n) { C1: i32 l := 0_{i32}; C2: i32 i := 0_{i32}; C3: while i <_u n: C4: i32 p := malloc_{C4}(sizeof(lnode)); C5: m := m[addrof(p $\xrightarrow{m}_{\text{lnode}}$ val) ← i]_{i32}; C6: m := m[addrof(p $\xrightarrow{m}_{\text{lnode}}$ next) ← l]_{i32}; C7: l := p; C8: i := i + 1_{i32}; C9: return l; CE: }</pre>
--	---

(a) (Abstracted) Spec IR

(b) (Abstracted) C IR

Figure 1.2: IRs for the Spec and C `mk_list` procedures in figs. 1.1a and 1.1b respectively.

procedure entry. Similarly, the output of a C procedure consists of its explicit return value and the state of program memory at procedure exit.

The Spec function `mk_list` (defined at line A6 in fig. 1.1a), takes a bitvector of size 32 ($n:i32$). It returns a `List` value representing the list $[(n-1), (n-2), \dots, 1, 0]$. On the other hand, the C function `mk_list` (defined at line B4 in Figure 1.1b) constructs a *pointer based* linked list representing a list identical to the Spec function. Unlike Spec, the construction of the linked list in C requires explicit allocation of memory through calls to `malloc` in addition to stores to the memory. We are interested in showing that the Spec and C `mk_list` procedures are ‘equivalent’ i.e., given equal n inputs, they both construct lists that are ‘equal’.

For comparison, we represent both programs in a common abstract framework. This involves converting both `mk_list` functions to a common logical representation (intermediate representation or IR for short). Figures 1.2a and 1.2b show the IRs of the Spec and C `mk_list` procedures in figs. 1.1a and 1.1b respectively. For the Spec program, the tail-recursive function `mk_list_impl` is converted to a loop and inlined in the top-level function `mk_list` in the IR. In case of the C program in fig. 1.1b, the memory state is made explicit (represented by m), and the size and memory layout of each type is concretized in the IR. For example, the `unsigned` and `pointer` types are encoded as the `i32` bitvector type. A comprehensive description of the logical model is

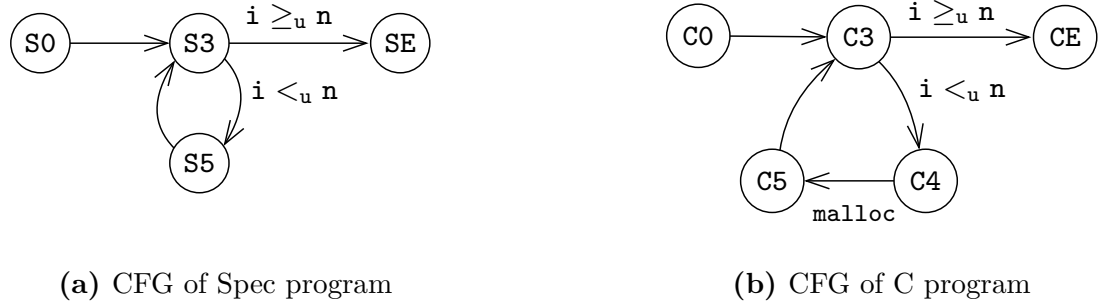
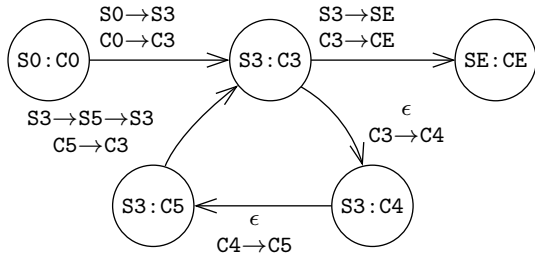


Figure 1.3: CFG representation of Spec and C IRs shown in figs. 1.2a and 1.2b for the `mk_list` procedures in figs. 1.1a and 1.1b respectively.

given in section 2.2.

To reiterate, we are interested in showing equivalence of the Spec and C IRs. Since the argument `n` to both procedures have identical types (i.e. `i32`), their equality is trivially expressible as: $\mathbf{n}_S = \mathbf{n}_C$ ¹. However, Spec uses the `List` ADT to represent a list, whereas the C procedure represents its list using a collection of `lnode` objects linked through their `next` fields in the memory \mathfrak{m} , and simply returns a value of type `i32` (`lnode*` in the original C program) pointing to the first `lnode` in the list (or the null value in case of an empty list). In order to express equality between these two list values (of types `List` and `i32`), we would like to ‘adapt’ one of the values so as to match their types. We choose to lift the C linked list (represented by the `i32` value and the C memory state) to a `List` value using an operator called a *lifting constructor*. Let us call this lifting constructor $\mathbf{Clist}_{\mathfrak{m}}^{\mathbf{lnode}}$, where the expression $\mathbf{Clist}_{\mathfrak{m}}^{\mathbf{lnode}}(p:\mathbf{i32})$ represents a `List` value constructed from a C pointer p (pointing to an `lnode` object) in memory state \mathfrak{m} . We will formally define $\mathbf{Clist}_{\mathfrak{m}}^{\mathbf{lnode}}$ subsequently in section 2.5. For now, such an operator allows us to express equality between the outputs of the Spec and C procedures as $\mathbf{ret}_S = \mathbf{Clist}_{\mathfrak{m}}^{\mathbf{lnode}}(\mathbf{ret}_C)$, where \mathbf{ret}_S and \mathbf{ret}_C represents the values returned by the respective Spec and C procedures in figs. 1.2a and 1.2b. To further emphasize the fact that we are comparing (a) a Spec ADT value with (b) an ADT value lifted from C values using a lifting constructor, we use ‘ \sim ’ instead of ‘ $=$ ’ and call it a recursive relation: $\mathbf{ret}_S \sim \mathbf{Clist}_{\mathfrak{m}}^{\mathbf{lnode}}(\mathbf{ret}_C)$.

¹We use S and C subscripts to refer to variables in the Spec and C procedures respectively.



(a) Product-CFG between CFGs
in figs. 1.3a and 1.3b

PC-Pair	Invariants	
(S0:C0)	(P)	$n_S = n_C$
(S3:C3)	(I1)	$n_S = n_C$
	(I3)	$i_S \leq_u n_S$
(S3:C4)	(I2)	$i_S = i_C$
	(I4)	$l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(S3:C5)	(I5)	$n_S = n_C$
	(I7)	$i_S <_u n_S$
(SE:CE)	(I6)	$i_S = i_C$
	(I8)	$l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(SE:CE)	(E)	$\text{ret}_S \sim \text{Clist}_m^{\text{lnode}}(\text{ret}_C)$

(b) Node invariants for product-CFG in fig. 1.4a

Figure 1.4: Product-CFG between the CFGs in figs. 1.3a and 1.3b. Figure 1.4b contains the corresponding node invariants for the product-CFG.

Consequently, we are interested in proving that given $n_S = n_C$ at the procedure entries, $\text{ret}_S \sim \text{Clist}_m^{\text{lnode}}(\text{ret}_C)$ holds at the exits of both procedures. Before going into the proof method, we first introduce an alternate representation of IR, called the Control-Flow Graph (CFG for short). Figures 1.3a and 1.3b show the CFG representation of the Spec and C IRs in figs. 1.2a and 1.2b respectively. The CFG representation is fundamentally a labeled transition system representation of the corresponding IR, and is further explored in section 2.2. In essence, each node represents a PC location of its IR, and each edge represents (possibly conditional) transition between PCs through instruction execution. For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 1.3b, the edge $C5 \rightarrow C3$ represents the path $C5 \rightarrow C6 \rightarrow C7 \rightarrow C8 \rightarrow C3$ in fig. 1.2b.

A common approach for showing equivalence between a pair of procedures involve finding a bisimulation relation across the said procedure-pair. Intuitively, a bisimulation relation (a) correlates program transitions across the specification and implementation procedures, and (b) asserts inductively-provable invariants between machine states of the two procedures at the endpoints of each correlated transition [43]. A bisimulation relation itself can be represented as a program, called a *product program* [53] and its CFG representation is called a *product-CFG*. Figure 1.4a shows a product-CFG between the Spec and C `mk_list` procedures in figs. 1.3a and 1.3b respectively.

At each node of the product-CFG, *invariants* relate the states of the Spec and C procedures respectively. Figure 1.4b lists the node invariants for the product-CFG in fig. 1.4a. At the start node ($S0:C0$) of the product-CFG, the precondition (labeled \textcircled{P}) ensures equality of input arguments n_S and n_C at the procedure entries. Inductive invariants (labeled \textcircled{I}) need to be inferred at each intermediate product-CFG node (e.g., ($S3:C3$)) relating both programs' states. For example, at node ($S3:C5$), $\textcircled{I6} \ i_S = i_C$ is an inductive invariant. The inductive invariant $\textcircled{I4} \ l_S \sim \text{Clist}_m^{\text{Inode}}(l_C)$ is another example of a recursive relation and asserts equality between the intermediate Spec and C lists at their respective loop heads. Assuming that the precondition (\textcircled{P}) holds at the entry node ($S0:C0$), a bisimulation check involves checking that the inductive invariants hold too, and consequently the postcondition (\textcircled{E}) holds at the exit node ($SE:CE$). Checking correctness of a bisimulation relation involves checking whether an invariant holds (among other things). These checks result in proof queries which must be discharged by a solver (aka theorem prover).

1.2 Our Contributions

As previously summarized in section 1.1, an algorithm to find a bisimulation based proof of equivalence between a Spec and C procedure involves three major algorithms: $\textcircled{A1}$ An algorithm for construction of a product-CFG by correlating program executions across the Spec and C procedures respectively. $\textcircled{A2}$ An algorithm for identification of inductively-provable invariants at intermediate correlated PCs. $\textcircled{A3}$ An algorithm for solving proof obligations generated by $\textcircled{A1}$ and $\textcircled{A2}$ algorithms. Next we list our major contributions.

Proof Discharge Algorithm

Solving proof obligations ($\textcircled{A3}$) involving recursive relations (generated by $\textcircled{A1}$ and $\textcircled{A2}$) is quite interesting and forms our primary contribution. We describe a *sound* proof discharge algorithm capable of tackling proof obligations containing recursive relations using off-the-shelf SMT solvers. Our proof discharge algorithm is also capable of reconstruction of counterexamples for the original

proof query from models returned by the individual SMT queries. These counterexamples form the foundation of counterexample-guided heuristics for $\textcircled{\text{A1}}$ and $\textcircled{\text{A2}}$ algorithms as we will discuss shortly. As part of our proof discharge algorithm, we reformulate equality of ADT values (i.e. recursive relations) as equivalence of programs and discharge these proof queries using a nested (albeit much simpler) equivalence check.

Spec-to-C Automatic Equivalence Checker Tool

Our second contribution is S2C, a *sound* equivalence checker tool capable of proving equivalence between a Spec and a C program automatically. S2C either successfully finds a bisimulation relation implying equivalence or it provides a (sound but incomplete) unknown verdict. S2C builds upon the Counter tool[27] and uses specialized versions of (a) counterexample-guided correlation algorithm for incremental construction of a product-CFG ($\textcircled{\text{A1}}$) and (b) counterexample-guided invariant inference algorithm for inference of inductive invariants at correlated PCs in the (partially constructed) product-CFG ($\textcircled{\text{A2}}$) based on prior work on counterexample-guided equivalence checking [48]. S2C discharges the required verification conditions (i.e. proof obligations) using our Proof Discharge Algorithm. The counterexamples generated by the proof discharge algorithm help steer the search algorithms $\textcircled{\text{A1}}$ and $\textcircled{\text{A2}}$.

1.3 Outline of the Thesis

The rest of the thesis is divided into five chapters. We begin with a thorough presentation of the topics introduced thus far in chapter 2. The major topics discussed include our specification language, logical representation of programs, and bisimulation in the context of Spec-C equivalence. We finish chapter 2 with a logical encoding of proof obligations generated during an equivalence check.

The next chapter, chapter 3 illustrates our proof discharge algorithm ($\textcircled{\text{A1}}$) through proof obligations originating from equivalence checks between previously shown program-pairs. In the context

of our proof discharge algorithm, we introduce a decomposition procedure based on unification to reduce a recursive relation to an equivalent set of equalities. The proof discharge algorithm categorizes a proof obligation into three types, each of which is illustrated with the help of examples. We finish with a summary and pseudocode of the algorithm.

Chapter 4 gives an exhaustive overview of the major components of our Spec-to-C equivalence checker tool, S2C. We begin with a dataflow formulation of the points-to analysis introduced as part of our proof discharge algorithm in chapter 3. Next, we summarize the counterexample driven algorithms for product-CFG construction ($\overline{\text{A2}}$) and invariant inference ($\overline{\text{A3}}$) based on prior work [48]. Chapter 4 presents the pseudocode for the major subprocedures used as part of our proof discharge algorithm, in addition to details of SMT encoding of proof obligations and recovery of counterexamples from SMT models. We finish with a new representation of expressions, called ‘Value Trees’ which enables the consolidation of multiple subprocedures utilized as a component of our proof discharge algorithm. We also present an algorithm for converting an expression to its value tree representation along with illustrative examples of its applications.

We provide a comprehensive evaluation of S2C in chapter 5, followed by its limitations. Chapter 6 concludes our work by reiterating the key ideas presented, alongside additional related work.

Chapter 2

Preliminaries

2.1 The Specification Language : Spec

We start with an introduction to our specification language, called Spec. Spec supports recursive algebraic data types (ADT) [15] similar to the ones available in functional languages such as Haskell [36] and SML [45]. Spec supports mutually recursive ADTs but does *not* support universal types. Additionally, Spec is equipped with the following *scalar* types: `unit`, `bool` (boolean) and `i<N>` (fixed-width bitvectors). ADTs can be thought of as ‘sum of product’ types where each *data constructor* represents a variant (of the sum-type) and the arguments to each data constructor represents its *fields* (of the product-type). For example, the `List` type (defined at A0 in fig. 1.1a) has two variants `LNil` and `LCons`. `LNil` has no fields while `LCons` has two fields `val` and `tail` of types `i32` and `List` respectively. Additionally, Spec follows *equirecursive* typing rules i.e. a `List` value l and `LCons`(l_{i32}, l) have definitionally *equal* types. Later in section 4.4.9, we further expand on ADTs in the context of a graphical representation of values used as part of our proof discharge algorithm. The language also borrows its expression grammar heavily from functional languages. This includes `let-in`, `if-then-else`, `match` and function application. Pattern matching (i.e. deconstruction) of ADT values is achieved through `match`. Unlike functional languages, Spec only supports first order functions. Also, Spec does not support partial function application. Hence,

```

A0: type Str = SInvalid | SNil
A1:       | SCons (ch:i8, tail:Str).
A2:
A3: fn is_empty (s : Str) : bool =
A4:   assuming  $\neg$ (s is SInvalid) do
A5:     s is SNil.

```

(a) Spec program

```

B0: #include <stdbool.h>
B1:
B2: bool is_empty(char* s) {
B3:   if (!s) return false;
B4:   return *s == 0;
B5: }

```

(b) C program

```

S0: bool is_empty(Str s) {
S1:   assume  $\neg$ (s is SInvalid);
S2:   return s is SNil;
SE: }

```

(c) (Abstracted) Spec IR

```

C0: bool is_empty(i32 s) {
C1:   if (s == 0_i32): return false;
C2:   return m[0_i32]_i8 == 0_i8;
CE: }

```

(d) (Abstracted) C IR

Figure 2.1: Spec and C programs along with corresponding IRs for the `is_empty` procedures.

we limit our attention to C programs containing only first order functions. Spec is equipped with a special `assuming-do` construct for explicitly providing assertions. Figure 2.1a shows a Spec program with the `assuming-do` construct, with its C equivalent shown in fig. 2.1b. The corresponding IRs are shown in figs. 2.1c and 2.1d respectively. The significance of explicit assertions in Spec is further discussed throughout this chapter. Spec also provides intrinsic scalar operators for expressing computation in C succinctly yet explicitly. Examples of scalar operators include (a) logical operators (e.g., `and`), (b) bitvector arithmetic operators (e.g., `bvadd(+)`), and (c) relational operators for comparing bitvectors interpreted as unsigned or signed integers (e.g., $\leq_{u,s}$). The equality operator (`=`) is only supported for scalar types.

Figure 2.2 shows the simplified expression grammar for Spec language. `<data-cons>` represents an ADT data constructor. The `<expr> is <data-cons>` construct returns a value of `bool` type and is used to test whether the value `<expr>` is of kind `<data-cons>`. `<scalar-op>` includes the logical, arithmetic and relational operators supported by Spec.

$\langle \text{expr} \rangle$	\rightarrow	$\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$ $ $ $\text{let } \langle \text{id} \rangle = \langle \text{expr} \rangle \text{ in } \langle \text{expr} \rangle$ $ $ $\text{match } \langle \text{expr} \rangle \text{ with } \langle \text{match-clause-list} \rangle$ $ $ $\text{assuming } \langle \text{expr} \rangle \text{ do } \langle \text{expr} \rangle$ $ $ $\langle \text{id} \rangle (\langle \text{expr-list} \rangle)$ $ $ $\langle \text{data-cons} \rangle (\langle \text{expr-list} \rangle)$ $ $ $\langle \text{expr} \rangle \text{ is } \langle \text{data-cons} \rangle$ $ $ $\langle \text{expr} \rangle \langle \text{scalar-op} \rangle \langle \text{expr} \rangle$ $ $ $\langle \text{literal}_{\text{unit}} \rangle \langle \text{literal}_{\text{bool}} \rangle \langle \text{literal}_{\text{iN}} \rangle$
$\langle \text{match-clause-list} \rangle$	\rightarrow	$\langle \text{match-clause} \rangle^*$
$\langle \text{match-clause} \rangle$	\rightarrow	$ \langle \text{data-cons} \rangle (\langle \text{id-list} \rangle) \Rightarrow \langle \text{expr} \rangle$
$\langle \text{expr-list} \rangle$	\rightarrow	$\epsilon \langle \text{expr} \rangle , \langle \text{expr-list} \rangle$
$\langle \text{id-list} \rangle$	\rightarrow	$\epsilon \langle \text{id} \rangle , \langle \text{id-list} \rangle$
$\langle \text{literal}_{\text{unit}} \rangle$	\rightarrow	$()$
$\langle \text{literal}_{\text{bool}} \rangle$	\rightarrow	$\text{false} \text{true}$
$\langle \text{literal}_{\text{iN}} \rangle$	\rightarrow	$[0 \dots 2^N - 1]$

Figure 2.2: Simplified expression grammar of Spec language

2.2 Abstract Representation of Programs

As outlined in section 1.1, we convert both Spec and C programs to a common abstract representation called the *Control-Flow Graph* (CFG for short). This process involves first converting both programs to a linear representation called the IR. In this section, we present both IR and CFG representations of Spec and C procedures.

2.2.1 Conversion of Programs to Intermediate Representation

IR is a Three-Address-Code (3AC) style intermediate representation. We often omit intermediate registers in the IR for brevity, and refer to this as the *abstracted* IR. We have already seen the IRs (in figs. 1.2a and 1.2b) for the Spec and C programs that construct lists in figs. 1.1a and 1.1b. Figures 2.3a and 2.3c show Spec and C programs that traverse a list of integers and compute their sum. The corresponding IR programs for the above are shown in figs. 2.3b and 2.3d respectively.

```

A0: type List = LNil | LCons (val:i32, tail:List).
A1:
A2: fn sum_list_impl (l:List) (sum:i32) :i32 =
A3:   match l with
A4:   | LNil => sum
A5:   | LCons(x, rest) =>
A6:       sum_list_impl(rest, sum + x).
A7:
A8: fn sum_list (l:List) : i32 =
A9:   sum_list_impl(l, 0i32).

```

(a) Spec program

```

S0: i32 sum_list (List l) {
S1:   i32 sum := 0i32;
S2:   while ¬(l is LNil):
S3:     // (l is LCons);
S4:     sum := sum + l.val;
S5:     l := l.next;
S6:   return sum;
SE: }

```

(b) (Abstracted) Spec IR

```

B0: typedef struct lnode {
B1:   unsigned val; struct lnode* next; } lnode;
B2:
B3: unsigned sum_list(lnode* l) {
B4:   unsigned sum = 0;
B5:   while (l) {
B6:     sum += l->val;
B7:     l = l->next;
B8:   }
B9:   return sum;
B10: }

```

(c) C program

```

C0: i32 sum_list (i32 l) {
C1:   i32 sum := 0i32;
C2:   while l ≠ 0i32:
C3:     sum := sum + l  $\xrightarrow{m}_{\text{lnode}}$  val;
C4:     l := l  $\xrightarrow{m}_{\text{lnode}}$  next;
C5:   return sum;
CE: }

```

(d) (Abstracted) C IR

Figure 2.3: Spec and C programs for traversing a Linked List along with corresponding IRs for the `sum_list` procedures.

The following major steps are performed during conversion of a Spec source to its IR representation:

1. **match** statements are converted to explicit **if-else** conditionals where each branch is associated with a **match** branch. The *sum-is* operator is used to query the top-level data constructor of an expression. The fields of the data constructor are bound to variables using the *product-access* or *accessor* operator. For example, the **match** statement at A3 (in fig. 2.3a) is lowered to **if-else** in fig. 2.3b, where ‘**l** is LCons’ is used to test whether **l** is of kind LCons and ‘**l.val**’ is used to extract the **val** field of LCons data constructor. Importantly, the expression ‘**e.fi**’ is well-formed iff ‘**e** is V_{fi} ’, where V_{fi} represents the data constructor containing the field **fi**. The construction of the IR guarantees the well-formedness of all *accessor* expressions.
2. All tail-recursive calls are converted to loops in the IR. However, all non-tail procedure calls are preserved as is. This transformation enables direct correlation (during equivalence checking) of tail-calls in Spec with native loops in C. For example, the tail recursive function **sum_list_impl** at A2 (in fig. 2.3a) is converted to a non-recursive function with a loop.
3. All *helper* functions¹ are inlined at their call-site. We are only interested in proving equivalence of non-helper functions in Spec with their C counterparts. For example, the helper function **sum_list_impl** (now non-recursive due to previous step), is inlined at call-site A7 in fig. 2.3a.
4. **assuming-do** statements are converted to their equivalent **assume** instruction in the IR. A Spec program is *well-defined* iff it satisfies all **assume** clauses encountered during its execution. These conditions are called *undefined behaviour assumes* or *UB assumes* for short. For example, the **assuming-do** statement at A4 in fig. 2.1a is converted to an **assume** instruction at S1 in fig. 2.1c.

Similarly, the following transformations are carried out during conversion of a C source to its IR:

¹We use a special marker to designate a function as ‘helper’ in Spec. For simplicity, this marker is omitted and instead helper function names are ended with the ‘_impl’ suffix.

1. Non-determinism in the original C program is determinized in the IR. This includes concretizing the size and memory layouts of both scalar (e.g. `int`) and compound (e.g., `struct`) types, along with fixing the order of evaluation in case it is unspecified. For example, during conversion of C program in fig. 1.1b to IR (in fig. 1.2b), the size of pointer types and `unsigned` is fixed to 32 bits (i.e. `i32`). Similarly, the memory layout (including alignment and offset) of `lnode` struct defined at B0 (in fig. 1.1b) is chosen. The implications of determinizing the C program behaviour are further discussed in chapter 4. For now, it is sufficient to note that we are interested in equivalence between Spec and this determinized version of C.
2. The memory state of the C program is made explicit, represented using the byte-addressable array '`m`'. Memory loads and stores are represented using explicit operations on `m`, e.g., (a) memory loads at C3 and C4 in fig. 2.3d, and (b) memory stores at C5 and C6 in fig. 1.2b. The memory load and store operators are defined promptly in section 2.2.2
3. We annotate calls to memory allocation functions (e.g., `malloc`) with their call-site, i.e., IR PC. For example, `mallocC4` is annotated with its call-site C4 in fig. 1.2b. These annotations are used by a points-to analysis done as part of our equivalence checking procedure, and defined subsequently in section 4.1.

2.2.2 IR Instructions

Note that both Spec and C programs are converted to the common IR. For a Spec procedure, IR supports scalar types, as well as ADTs defined in its Spec source. The IR also inherits the scalar operators available as part of the Spec language. Each ADT value can be thought of as a key-value dictionary that maps each of its field names to their respective values. These key-value pairs are accessed using the previously introduced *accessor* operator, e.g., `l.val` and `l.next` represents the first and second fields of the `LCons` data constructor in fig. 2.3b. Recall that, the IR also allows querying the top-level variant of an ADT value using the *sum-is* operator, e.g., `l` is `LNil` at S2 in fig. 2.3b. The `val` field is associated with the `LCons` data constructor and evidently, `l.val` (and `l.next`) is only *well-formed* under `l` is `LCons`. As previously stated, the well-formedness of all

accessor expressions are preserved during construction of IR for a Spec procedure. Using *accessor* and *sum-is* operators, a **List** value l can be expanded as:

$$U_S : l = \underline{\text{if}} \ l \text{ is LNil } \underline{\text{then}} \ \text{LNil} \ \underline{\text{else}} \ \text{LCons}(l.\text{val}, l.\text{next}) \quad (2.1)$$

In this expanded representation of l , the *sum-deconstruction* operator ‘if-then-else’ conditionally deconstructs the sum type into its variants **LNil** and **LCons**. The *underlined if-then-else* operator is a stricter version of **if-then-else**, and is reserved for ADT values. An if-then-else expression e (for an ADT T) must satisfy the following properties: (a) e has exactly one branch for each data constructor of T (in the order they are defined), and (b) the branch associated with the data constructor V has the form $V(e_1, e_2, \dots)$ i.e. its top-level operator is V . For example, an if-then-else expression for the **List** type must be of the form: ‘if e_1 then **LNil** else **LCons**(e_2, e_3)’ for some expressions e_1, e_2, e_3 . Equation (2.1) is called the *unrolling procedure* for the **List** variable l . We can similarly define the unrolling procedure for any ADT variable (based on the definition of the ADT).

On the C side, the size of a pointer is fixed² and the memory state is modeled as a byte-addressable array over bitvectors (represented by \mathbb{m}). “ $\mathbb{m}[p]_{\text{T}}$ ” represents a memory load operation and is equal to the bytes at addresses $[p, p + \text{sizeof}(\text{T})]$ in \mathbb{m} , interpreted as a value of type T . Similarly, “ $\mathbb{m}[p \leftarrow v]_{\text{T}}$ ” represents a memory store operation and is equal to \mathbb{m} everywhere except at addresses $[p, p + \text{sizeof}(\text{T})]$ which contains the value v of type T (e.g., **C5** in fig. 1.2b). We use the following two C-like syntaxes to represent more intricate memory loads succinctly:

1. “ $p \xrightarrow{\mathbb{m}}_{\text{T}} \text{fi}$ ” is equivalent to “ $\mathbb{m}[p + \text{offsetof}(\text{T}, \text{fi})]_{\text{typeof}(\text{T}, \text{fi})}$ ” i.e., it returns the bytes in the memory array \mathbb{m} starting at address ‘ $p + \text{offsetof}(\text{T}, \text{fi})$ ’ and interpreted as a value of type $\text{typeof}(\text{T}, \text{fi})$.
2. “ $p[i]_{\mathbb{m}}^{\text{T}}$ ” is equivalent to “ $\mathbb{m}[p + i \times \text{sizeof}(\text{T})]_{\text{T}}$ ” i.e., it returns the bytes in the memory array \mathbb{m} starting at address ‘ $p + i \times \text{sizeof}(\text{T})$ ’ and interpreted as a value of type T . Interestingly, $\mathbb{m}[p]_{\text{T}} = p[0]_{\mathbb{m}}^{\text{T}}$.

²We choose an address width of 4 bytes or 32 bits throughout this thesis.



Figure 2.4: CFG representation of Spec and C IRs in figs. 2.1c and 2.1d for the `is_empty` procedures in figs. 2.1a and 2.1b respectively.



Figure 2.5: CFG representation of Spec and C IRs in figs. 2.3b and 2.3d for the `sum_list` procedures in figs. 2.3a and 2.3c respectively.

Recall that the size and layout of each type in C is concretized in the IR, and hence the values ‘`offsetof(T,f)`’ and ‘`sizeof(T)`’ are constants. We use the ‘`addrof()`’ operator to extract the address of a memory load expression: “`addrof($\mathfrak{m}[p]_T$)`” is equivalent to p . For example, at PC C5 in fig. 1.2b, $\text{addrof}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}) \Leftrightarrow p + \text{offsetof}(\text{lnode}, \text{val})$. Additionally, given a bitvector expression e , “ $e[\text{ub} : \text{lb}]$ ” represents a bitvector of size $(\text{ub} - \text{lb} + 1)$, such that the i^{th} bit is equal to the $(i + \text{lb})^{\text{th}}$ bit in e .

2.2.3 Control-Flow Graph Representation

Figures 2.5a and 2.5b show the Control-Flow Graph (CFG) representation of the Spec and C IRs in figs. 2.3b and 2.3d respectively. Additionally, the CFG representation of IRs in figs. 2.1c

and 2.1d are shown in figs. 2.4a and 2.4b respectively. The Control-Flow Graph is an alternate graphical representation of an IR program that emphasizes the control flow structures of the static program. Each CFG node represents a program point (i.e. IR PC) and is denoted by n . The CFG representation is analogous to a deterministic labeled transition systems and uses a symbolic state Ω_n to represent the machine state at node n . An edge from n to n' (denoted by $\omega[n \rightarrow n']$) represents transition from n to n' through execution of instructions and is associated with:

1. An *edge condition* representing the condition that must be satisfied by Ω_n to trigger the edge ω .
2. A *transfer function* representing the symbolic state at n' ($\Omega_{n'}$) as a function of Ω_n i.e. how the machine state is mutated along the edge ω .
3. An *UB assume* representing the condition that must be satisfied by Ω_n for the transition ω to be well-defined. For a Spec procedure, the **assume** clauses form its UB assumes. Recall that, a C procedure is determinized during conversion to CFG and thus do not require UB assumes.

For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 2.5a, the edge $S2 \rightarrow S5$ represents the path $S2 \rightarrow S3 \rightarrow S4 \rightarrow S5$ in fig. 2.3b. In such a case, the transfer function of the edge is the composition of the sequence of instructions. A CFG must contain exactly one entry node (representing the entry to the function) and (possibly) multiple exit nodes (each representing an exit from the function). For example, the CFG in fig. 2.4b contains an entry node $C0$ and two exit nodes CE_1 and CE_2 representing exits through the IR PCs $C1$ and $C2$ in fig. 2.1d respectively. An edge incident on an exit node is called an *exit edge*, and is associated with an *action* representing the returned value as a function of the symbolic state at the source node. Actions form the observable behaviour of a CFG while transition through non-exit edges are internal to the program. For a C CFG, the action includes both the returned value (if non-void) and the memory state. We restrict ourselves to programs without calls to *external procedures* (except for **malloc** in \mathcal{C}), and thus the only observable action of a CFG are its returned values along exit edges. We often omit the transfer functions in the CFG figures (if

they are shown in their corresponding IR) and only show the edge conditions (unless they are *true*). The UB assumes are shown inside curved rectangles (e.g., fig. 2.4b), unless they are *true*. Henceforth, we refer to the CFGs of Spec and C procedures as \mathcal{S} and \mathcal{C} respectively.

2.3 Equivalence Definition

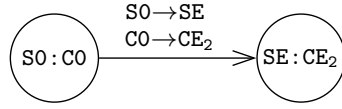
Given (1) a Spec function specification \mathcal{S} , (2) a C implementation \mathcal{C} , (3) a precondition Pre that relates the initial inputs $\text{Input}_{\mathcal{S}}$ and $\text{Input}_{\mathcal{C}}$ to \mathcal{S} and \mathcal{C} respectively, and (4) a postcondition $Post$ that relates the final outputs $\text{Output}_{\mathcal{S}}$ and $\text{Output}_{\mathcal{C}}$ of \mathcal{S} and \mathcal{C} respectively³: \mathcal{S} and \mathcal{C} are *equivalent* if for all possible inputs $\text{Input}_{\mathcal{S}}$ and $\text{Input}_{\mathcal{C}}$ such that $Pre(\text{Input}_{\mathcal{S}}, \text{Input}_{\mathcal{C}})$ holds, \mathcal{S} 's execution is well-defined on $\text{Input}_{\mathcal{S}}$, and \mathcal{C} 's memory allocation requests during its execution on $\text{Input}_{\mathcal{C}}$ are successful, then both \mathcal{S} and \mathcal{C} produce outputs such that $Post(\text{Output}_{\mathcal{S}}, \text{Output}_{\mathcal{C}})$ holds.

$$Pre(\text{Input}_{\mathcal{S}}, \text{Input}_{\mathcal{C}}) \wedge (\mathcal{S} \text{ def}) \wedge (\mathcal{C} \text{ fits}) \Rightarrow Post(\text{Output}_{\mathcal{S}}, \text{Output}_{\mathcal{C}})$$

The $(\mathcal{S} \text{ def})$ antecedent states that we are only interested in proving equivalence for well-defined executions of \mathcal{S} , i.e., executions that satisfy all assertions expressed using the **assuming-do** statement. The $(\mathcal{C} \text{ fits})$ antecedent states that we prove equivalence under the assumption that \mathcal{C} 's memory requirements fit within the available system memory i.e., only for those executions of \mathcal{C} in which all memory allocation requests (through **malloc** calls) are successful.

Recall that the observables of \mathcal{S} and \mathcal{C} are the actions associated with their exit edges (i.e. returned values). For \mathcal{S} , observables include the explicit value returned. For \mathcal{C} , observables include the returned value (if non-void) along with the memory state at procedure exit. The postcondition $Post$ relates these outputs of the two programs. The pair $(Pre, Post)$ represents the input-output behaviour of \mathcal{C} in terms of the specification \mathcal{S} , and is called the *input-output specification*. In general, Spec and C sources may contain multiple top-level procedures, with calls to each other. In this case, we are interested in finding equivalence between the CFGs of each pair of \mathcal{S} and \mathcal{C} procedures with respect to their input-output specification.

³ $\text{Input}_{\mathcal{C}}$ and $\text{Output}_{\mathcal{C}}$ include the initial and final memory state of \mathcal{C} respectively.



PC-Pair	Invariants
$(S0:C0)$	$(\textcircled{P}) \ s_S \sim \text{Cstr}_m^{\text{u8}}(s_C)$
$(SE:CE_2)$	$(\textcircled{E}) \ \text{ret}_S = \text{ret}_C$

(a) Product-CFG between figs. 2.4a and 2.4b (b) Note invariants for product-CFG in fig. 2.6a

Figure 2.6: Product-CFG and its associated node invariants for CFGs in figs. 2.4a and 2.4b.

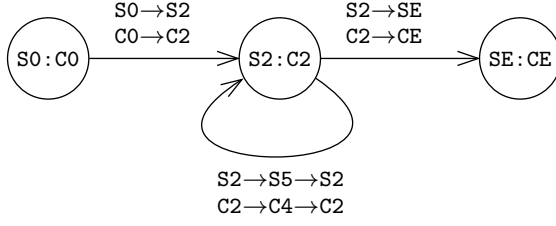
2.3.1 Constraining Inputs to C

Sometimes, the user may be interested in constraining the nature of inputs to \mathcal{C} for the purpose of checking equivalence only for *well-formed* inputs. In those circumstances, we use a combination of $(\mathcal{S} \text{ def})$ and Pre to constrain the execution of \mathcal{C} to inputs for which we are interested in proving equivalence. For example, consider the function `is_empty` (shown in figs. 2.1a and 2.1b) that accepts a string and checks if the string is empty. A string is represented as a list of characters (i.e. `i8`) in the Spec procedure. On the other hand, its C analogue expects a standard null-character⁴ terminated string represented by a pointer to its first character, say s_C . A *well-formed* nul-terminated string must point to an allocated region of memory terminating with a nul (i.e. *zero*) character and thus s_C must be non-null. Observe that the well-formedness of s_C is an entry assumption that must be ensured by the caller. Evidently, we are only interested in verifying the behaviour of the C procedure (against its specification) for well-formed inputs. Note that at B3 in fig. 2.1b, we handle the case of s_C being null for safety, even though a well-formed input would never trigger it.

Since Spec has no notion of pointers, we expose this conditional well-formedness of C input s_C through an explicit data constructor `SInvalid` for the `Str` ADT defined at A0 in fig. 2.1a. Additionally, $(\mathcal{S} \text{ def})$ asserts $\neg(s_S \text{ is } \text{SInvalid})$ (at A4 and $S1 \rightarrow S2$ in fig. 2.1c) and the precondition Pre (labeled (\textcircled{P}) in fig. 2.6b) relates $(s_S \text{ is } \text{SInvalid}) \Leftrightarrow (s_C = 0_{\text{i32}})$ ⁵. Combined, $(\mathcal{S} \text{ def})$ and Pre ensure that we constrain the inputs of \mathcal{S} and in turn, \mathcal{C} to only well-formed values during

⁴We use *nul* and *null* to denote the zero character and the null pointer respectively.

⁵The relation is implied by the recursive relation $1_S \sim \text{Cstr}_m^{\text{u8}}(1_C)$ as part of Pre shown in fig. 2.6b. The lifting constructor $\text{Cstr}_m^{\text{u8}}$ is defined subsequently in section 2.5.



PC-Pair	Invariants
(S0:C0)	$\textcircled{P} \ 1_S \sim \text{Clist}_{\text{rm}}^{\text{inode}}(1_C)$
(S2:C2)	$\textcircled{I1} \ 1_S \sim \text{Clist}_{\text{rm}}^{\text{inode}}(1_C)$ $\textcircled{I2} \ \text{sum}_S = \text{sum}_C$
(SE:CE)	$\textcircled{E} \ \text{ret}_S = \text{ret}_C$

(a) Product-CFG between figs. 2.5a and 2.5b (b) Node invariants for product-CFG in fig. 2.7a

Figure 2.7: Product-CFG between the CFGs in figs. 2.5a and 2.5b. Figure 2.7b contains the corresponding node invariants for the product-CFG.

equivalence check. A similar strategy is employed for string functions from the standard library (e.g. `strchr`) and is explored in detail during evaluation in section 5.1.1.

2.4 Bisimulation Relation

Recall that, we construct a *bisimulation relation* to identify equivalence between the CFGs of Spec and C procedures. A bisimulation relation correlates the transitions of \mathcal{S} and \mathcal{C} in lockstep, such that the lockstep execution ensures identical observable behaviour. A bisimulation relation between two programs can be represented using a *product program* [53] and the CFG representation of a product program is called a *product-CFG*. Figure 2.7a shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 2.5a and 2.5b.

A node in the product-CFG is formed by pairing nodes of \mathcal{S} and \mathcal{C} , e.g., $(S2:C2)$ is formed by pairing S2 and C2. If the lockstep execution of both programs is at node $(S2:C2)$ in the product-CFG, then \mathcal{S} 's execution is at S2 and \mathcal{C} 's execution is at C2. The start node $(S0:C0)$ of the product-CFG correlates the start nodes of CFGs of \mathcal{S} and \mathcal{C} . Similarly, an exit node $(SE:CE)$ correlates exit nodes of both programs.

An edge in the product-CFG is formed by pairing a *path* (a sequence of edges) in \mathcal{S} with a path

in \mathcal{C} ⁶. A product-CFG edge encodes the lockstep execution of its correlated paths. For example, the product-CFG edge $(S2:C2) \rightarrow (S2:C2)$ is formed by pairing $S2 \rightarrow S5 \rightarrow S2$ and $C2 \rightarrow C4 \rightarrow C2$ in figs. 2.5a and 2.5b respectively, and represents that when \mathcal{S} makes the transition $S2 \rightarrow S5 \rightarrow S2$, \mathcal{C} makes the transition $C2 \rightarrow C4 \rightarrow C2$ in lockstep. In general, a product-CFG edge e may correlate a finite path ρ_S in \mathcal{S} with a finite path ρ_C in \mathcal{C} , written $e = (\rho_S, \rho_C)$. The empty path ϵ in \mathcal{S} may be correlated with a finite path in \mathcal{C} , effectively simulating a *stuttering bisimulation* relation.

At the start node $(S0:C0)$ of the product-CFG in fig. 2.7a, the precondition Pre (labeled \textcircled{P}) ensures equality of input lists l_S and l_C at procedure entries. *Inductive invariants* (labeled \textcircled{I}) are inferred at each intermediate product-CFG node (e.g., $(S2:C2)$) that relate the values of \mathcal{S} with values and memory state of \mathcal{C} . At an exit node $(SE:CE)$ of the product-CFG, the postcondition $Post$ (labeled \textcircled{E}) represents equality of observable outputs and forms our top-level proof obligation. Assuming that the precondition Pre (\textcircled{P}) holds at the entry node $(S0:C0)$, a bisimulation check involves checking that the inductive invariants (\textcircled{I}) hold too, and consequently the postcondition $Post$ (\textcircled{E}) holds at each exit node $(SE:CE)$. The input-output specification (i.e. $(Pre, Post)$) is manually provided by the user while all inductive invariants are identified by an invariant inference algorithm described in section 4.3.

2.4.1 Well-formedness of Product-CFG

A product-CFG is well-formed (i.e. represents a valid bisimulation relation) if it correlates every *possibly* executable edge (as part of a path) in \mathcal{C} with a path in \mathcal{S} . An edge deemed to be unreachable (due to an unsatisfiable path condition) represents *dead code* and can be safely ignored and remain uncorrelated in the final product-CFG. For example, the $(\mathcal{S} \text{ def})$ and Pre conditions assert unreachability of the edge $C0 \rightarrow CE_1$ in fig. 2.4b, and remain uncorrelated in the product-CFG in fig. 2.6a. Without $(\mathcal{S} \text{ def})$ and Pre , our equivalence checker would fail because the edge $C0 \rightarrow CE_1$ is only triggered for ill-formed inputs and has no correlated paths in its specification.

⁶For ease of exposition, we present product-CFG in the context of *path* correlations. However, a more general approach of pathset is used based on prior work [27] for improved completeness of our algorithm. We will explore pathsets and its consequences on the algorithm in more detail in section 4.2.1.

Additionally, a well-formed product-CFG may not correlate a loop in \mathcal{C} with the empty path ϵ in \mathcal{S} . For example, fig. 1.4a shows the product-CFG between the programs in figs. 1.3a and 1.3b respectively. The edges $(S3:C3) \rightarrow (S3:C4)$ and $(S3:C4) \rightarrow (S3:C5)$ correlate the empty path ϵ with the non-empty paths $C3 \rightarrow C4$ and $C4 \rightarrow C5$ respectively. However, the only loop path $C3 \rightarrow C4 \rightarrow C5 \rightarrow C3$ in \mathcal{C} is still correlated with the non-empty path $S3 \rightarrow S5 \rightarrow S3$ in \mathcal{S} and thus, the product-CFG in fig. 1.4a satisfies this well-formedness criterion. This well-formedness condition ensures *divergence*, i.e. either both programs terminate or both continue indefinitely.

2.5 Recursive Relation

In section 1.1, we briefly introduced a lifting constructor ($\text{Clist}_{\mathfrak{m}}^{\text{lnode}}$) and associated recursive relations. In fig. 2.7b, the precondition $(\textcircled{\text{P}})$ is another instance of a recursive relation: “ $l_S \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)$ ” where l_S and l_C represent the input arguments to the Spec and C procedures respectively, **lnode** is the C **struct** type that contains the **val** and **next** fields (defined at B0 in fig. 2.3c), and \mathfrak{m} is the byte-addressable array representing the current memory state of the C program. $l_1 \sim l_2$ is read l_1 is recursively equal to l_2 and is semantically equivalent to $l_1 = l_2$. The ‘ \sim ’ simply emphasizes that l_1 and l_2 are (possibly recursive) ADT values. The lifting constructor $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}$ ‘lifts’ a C pointer value p (pointing to an object of type **struct lnode**) and memory state \mathfrak{m} to a (possibly infinite in case of a circular list) **List** value, and is defined through its *unrolling procedure* as follows:

$$U_C : \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p : i32) = \underline{\text{if}} \ p = 0 \ \underline{\text{then}} \ \text{LNil} \quad (2.2)$$

$$\quad \underline{\text{else}} \ \text{LCons}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}, \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next}))$$

Note the recursive nature of the lifting constructor $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}$: if the pointer p is zero (i.e. p is the null pointer), then it represents the empty list **LNil**; otherwise it represents the list formed by **LCons**-ing the value stored at $p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}$ in memory \mathfrak{m} and the list formed by recursively lifting $p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next}$ through $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}$. $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)$ allows us to adapt a C linked list (formed by chasing pointers in the memory \mathfrak{m}) to a **List** value and compare it with a Spec **List** value

for equality.

Also, consider the **Str** lifting constructor $\text{Cstr}_m^{\text{u8}\square}$ (in fig. 2.6b) used to lift a nul-terminated C string into a **Str** value. Similar to the **List** lifting constructor, $\text{Cstr}_m^{\text{u8}\square}$ is also defined through its unrolling procedure as follows:

$$\begin{aligned} \text{Cstr}_m^{\text{u8}\square}(p:\text{i32}) = & \text{if } p = 0 \text{ then } \text{SInvalid} \\ & \text{elif } p[0]_m^{\text{i8}} = 0 \text{ then } \text{SNil} \\ & \text{else } \text{SCons}(p[0]_m^{\text{i8}}, \text{Cstr}_m^{\text{u8}\square}(p + 1)) \end{aligned} \quad (2.3)$$

Observe that an ill-formed string is related to the pointer being null, whereas the nul character represents the empty string.

2.6 Proof Obligations

As previously discussed, algorithms for (a) incremental construction of a Product-CFG and (b) inference of invariants at intermediate PCs in the (partially constructed) product-CFG, are based on prior work[48] and discussed subsequently in sections 4.2 and 4.3. We discuss the proof obligations that arise from a given product-CFG. Recall that a bisimulation check involves checking that all inductive invariants (and the postcondition *Post*) hold at their associated product-CFG nodes.

We use relational Hoare triples to express these proof obligations [13, 28]. If ϕ denotes a predicate relating the machine states of \mathcal{S} and \mathcal{C} , then for a product-CFG edge $e = (\rho_S, \rho_C)$, $\{\phi_s\}(e)\{\phi_d\}$ denotes the condition: if any machine states σ_S and σ_C of programs \mathcal{S} and \mathcal{C} are related through precondition $\phi_s(\sigma_S, \sigma_C)$ and the finite paths ρ_S and ρ_C are executed in \mathcal{S} and \mathcal{C} respectively, then execution terminates normally in states σ'_S (for \mathcal{S}) and σ'_C (for \mathcal{C}) and postcondition $\phi_d(\sigma'_S, \sigma'_C)$ holds.

For every product-CFG edge $e[s \rightarrow d] = (\rho_S, \rho_C)$, we are interested in proving: $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$,

where ϕ_s and ϕ_d are the node invariants at the product-CFG nodes s and d respectively. The weakest-precondition transformer is used to translate a Hoare triple $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$ to the following first-order logic formula:

$$(\phi_s \wedge \text{pathcond}_{\rho_S} \wedge \text{pathcond}_{\rho_C} \wedge \text{ubfree}_{\rho_S}) \Rightarrow \text{WP}_{\rho_S, \rho_C}(\phi_d) \quad (2.4)$$

Here, pathcond_{ρ_X} represents the condition that path ρ is taken in program X and ubfree_{ρ_S} represents the condition that execution of \mathcal{S} along path ρ_S is free of undefined behaviour. $\text{WP}_{\rho_S, \rho_C}(\phi_d)$ represents the weakest-precondition of the predicate ϕ_d across the product-CFG edge $e = (\rho_S, \rho_C)$. From now on, we will use ‘LHS’ and ‘RHS’ to refer to the antecedent and consequent of the implication operator ‘ \Rightarrow ’ in eq. (2.4).

For example, checking that the loop invariant $\textcircled{\text{I1}} \ 1_S \sim \text{Clist}_m^{\text{lnode}}(1_C)$ holds at (S2:C2) in fig. 2.7a requires that $\textcircled{\text{I1}}$ is provable along the two product-CFG edges $(\text{S0:C0}) \rightarrow (\text{S2:C2})$ and $(\text{S2:C2}) \rightarrow (\text{S2:C2})$ terminating at (S2:C2) , which further reduces to the following two proof obligations:

- ① $\{\phi_{\text{S0:C0}}\}(\text{S0} \rightarrow \text{S2}, \text{C0} \rightarrow \text{C2})\{1_S \sim \text{Clist}_m^{\text{lnode}}(1_C)\}$
- ② $\{\phi_{\text{S2:C2}}\}(\text{S2} \rightarrow \text{S5} \rightarrow \text{S2}, \text{C2} \rightarrow \text{C4} \rightarrow \text{C2})\{1_S \sim \text{Clist}_m^{\text{lnode}}(1_C)\}$

Using weakest precondition predicate transformer, the proof obligation ② reduces to the following first-order logic formula:

$$\begin{aligned} 1_S \sim \text{Clist}_m^{\text{lnode}}(1_C) \wedge \text{sum}_S = \text{sum}_C \wedge (1_S \text{ is LCons}) \wedge (1_C \neq 0) \\ \Rightarrow 1_S.\text{next} \sim \text{Clist}_m^{\text{lnode}}(1_C \xrightarrow{\text{lnode}} \text{next}) \end{aligned} \quad (2.5)$$

Due to the presence of recursive relations, these proof queries (e.g., eq. (2.5)) cannot be solved directly by off-the-shelf solvers and require special handling. The next chapter illustrates our proof discharge algorithm for solving proof queries involving recursive relations.

Chapter 3

Proof Discharge Algorithm through Illustrative Examples

This chapter demonstrates our proof discharge algorithm through examples. We consider proof obligations generated due to product-CFG invariants shown in figs. 1.4 and 2.7 representing bisimulation relations for the `mk_list` and `sum_list` procedures respectively. We start by describing the properties of the proof discharge algorithm. We also list the properties of the proof obligations generated by our equivalence checker; these properties are essential for the correctness of our proof discharge algorithm. Next, the proof discharge algorithm is explored using instances of proof obligations, and we finish with an overview of the algorithm.

3.1 Properties of Proof Discharge Algorithm

3.1.1 Soundness of Proof Discharge Algorithm

An algorithm that evaluates the truth value of a proof obligation is called a *proof discharge algorithm*. In case a proof discharge algorithm deems a proof obligation to be unprovable, it

is expected to return *false* with a set of counterexamples that falsify the proof obligation. A proof discharge algorithm is *precise* if for all proof obligations, the truth value evaluated by the algorithm is identical to the proof obligation's *actual* truth value. A proof discharge algorithm is *sound* if: (a) whenever it evaluates a proof obligation to true, the actual truth value of that proof obligation is also true, and (b) whenever it generates a counterexample, that counterexample must falsify the proof obligation. However, it is possible for a sound proof discharge algorithm to return false (without counterexamples) when the proof obligation was actually provable.

For proof obligations generated by our equivalence checker, it is always safe for a proof discharge algorithm to return false (without counterexamples). Keeping this in mind, our proof discharge algorithm is designed to be *sound*. Conservatively evaluating a proof obligation to false (when it was actually provable) may prevent the equivalence proof from completing successfully. However, importantly, the overall equivalence procedure remains sound i.e. (a) either it successfully finds a valid proof of equivalence (bisimulation relation) or (b) it conservatively returns *unknown*.

3.1.2 Conjunctive Recursive Relation Property of Proof Obligations

Resolving the truth value of a proof obligation that contains a recursive relation such as $1_S \sim \text{Clist}_m^{\text{lnode}}(1_C)$ is unclear. Fortunately, the shapes of the proof obligations generated by our equivalence checker are restricted. Our equivalence checking algorithm ensures that, for an invariant $\phi_s = (\phi_s^1 \wedge \phi_s^2 \wedge \dots \wedge \phi_s^k)$, at any node s of a product-CFG, if a recursive relation appears in ϕ_s , it must be one of $\phi_s^1, \phi_s^2, \dots$, or ϕ_s^k . We call this the *conjunctive recursive relation* property of an invariant ϕ_s .

A proof obligation $\{\phi_s\}(e)\{\phi_d\}$, where $e[s \rightarrow d] = (\rho_S, \rho_C)$, gets lowered using $\text{WP}_e(\phi_d)$ (as shown in eq. (2.4)) to a first-order logic formula of the following form:

$$(\eta_1^s \wedge \eta_2^s \wedge \dots \wedge \eta_m^s) \Rightarrow (\eta_1^d \wedge \eta_2^d \wedge \dots \wedge \eta_n^d) \quad (3.1)$$

Thus, due to the conjunctive recursive relation property of ϕ_s and ϕ_d , any recursive relation in

eq. (3.1) must appear as one of η_i^s or η_j^d . To simplify proof obligation discharge, we break a first-order logic proof obligation P of the form in eq. (3.1) into multiple smaller proof obligations of the form $P_j : (\text{LHS} \Rightarrow \eta_j^d)$, for $j = 1..n$. Each proof obligation P_j is then discharged separately. We call this conversion from a bigger query to multiple smaller queries, *RHS-breaking*.

We provide a sound (but imprecise) proof discharge algorithm that converts a proof obligation generated by our equivalence checker into a series of SMT queries. Our algorithm begins by categorizing a proof obligation into one of three types; each type is discussed separately in subsequent sections. The categorization is based on a specialized unification procedure, which we describe next.

3.2 Iterative Unification and Rewriting Procedure

In this section, we introduce a unification procedure capable of decomposing an equality (e.g., a recursive relation) into an equivalent set of equalities (including scalar equalities and recursive relations).

3.2.1 Atomic Expression

We begin with some definitions. An expression e whose top-level constructor is a lifting constructor, e.g., $e = \text{Clist}_{\mathbf{m}}^{\text{lnode}}(\mathbf{l}_C)$, is called a *lifted expression*. An expression e of the form $v.\mathbf{a}_1.\mathbf{a}_2...\mathbf{a}_n$ i.e. a variable nested within *zero* or more *accessors*, is called a *pseudo-variable*. Note that, a variable is itself a pseudo-variable. Examples of pseudo-variable include the variable l and $l.\text{val}$.

Scalar expressions, pseudo-variables and lifted expressions are collectively called *atomic expressions*, or *atoms* for short. Hence, a **List** variable l , a bitvector expression $e_{\mathbf{i}32}$ and a lifted expression $\text{Clist}_{\mathbf{m}}^{\text{lnode}}(p)$ are all examples of atoms. With each atom of an ADT type, we associate an *unrolling procedure*. By definition, an ADT atom is either a pseudo-variable or a lifted expression. Each (pseudo-)variable is associated with its unrolling procedure governed by its type. For example, the unrolling procedure for a **List** variable l is given by U_S (eq. (2.1)). For

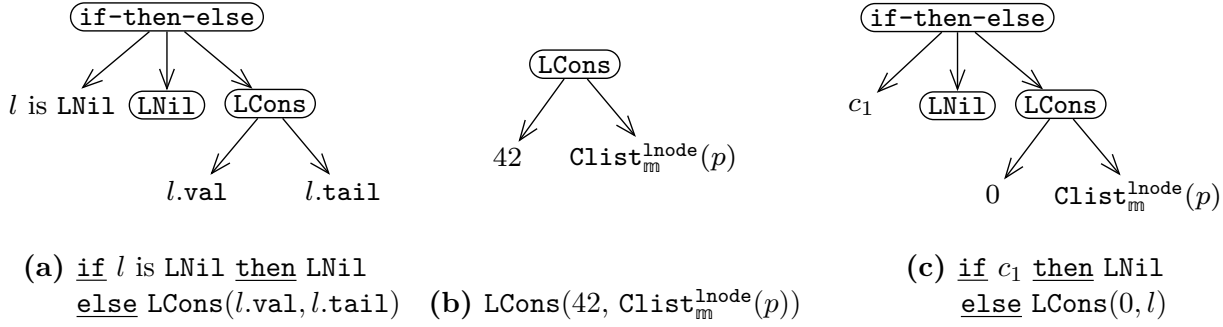


Figure 3.1: Expression trees corresponding to three canonical List expressions used in the context of unification.

lifted expressions, the unrolling procedure is given by its definition, e.g., U_C (eq. (2.2)) for the lifting constructor $Clist_m^{lnode}$.

3.2.2 Expression Trees

An expression e in which (a) each *accessor* (e.g., ‘ $_.tail$ ’) and (b) each *sum-is* operator (e.g., ‘ $_ \text{ is LCons}$ ’) operate on a pseudo-variable, is called a *canonical expression*. It is possible to convert any expression e into its canonical form \hat{e} . For example, the canonical form of $a + LCons(b, l).tail.val$ is given by $a + l.val$, where $l.val$ is a pseudo-variable. The pseudocode for the canonicalization procedure is given in section 4.4.2.

Consider the expression tree of a canonical expression \hat{e} , as shown in fig. 3.1. The internal nodes of \hat{e} represents ADT data constructors and the if-then-else sum-deconstruction operator. The leaves of \hat{e} are the atomic subexpressions of \hat{e} . For example, in fig. 3.1a, (if-then-else), (LNil) and (LCons) are the internal nodes, whereas $l \text{ is LNil}$, $l.val$ and $l.tail$ are the (atomic) leaf expressions. Note that nullary data constructors (e.g. LNil) are considered internal nodes even through they have no children.

The *expression path* to a node v in \hat{e} ’s tree is the path from the root of \hat{e} to the node v . The *expression path condition* represents the conjunction of all the if conditions (if the then branch

of taken along the path), or their negation (if the else branch is taken along the path) for each if-then-else along the path. Consider the expression if c_1 then LNil else $\text{LCons}(0, l)$ and its associated expression tree in fig. 3.1c. The expression path condition of c_1 is **true**, of LNil is c_1 and of $\text{Clist}_{\text{m}}^{\text{lnode}}(p)$ is $\neg c_1$.

3.2.3 Unification Procedure

For two expressions e_1 and e_2 under expression path conditions p_1 and p_2 respectively, a unification procedure $\theta(p_1, e_1, p_2, e_2)$ attempts to unify the canonical expression trees of e_1 and e_2 created by data constructors and the if-then-else operator (as shown in fig. 3.1). The unification procedure either fails to unify, or it returns *correlation tuples* $\langle p_1, a_1, p_2, e_2 \rangle$ where atom a_1 at expression path condition p_1 in one expression is correlated with expression e_2 at expression path condition p_2 in the other expression.

If *at least* one of e_1 and e_2 (say e_2) is atomic, unification always succeeds and returns a single correlation tuple: $\langle p_2, e_2, p_1, e_1 \rangle$. Recall that a variable is an atom and evidently, l and $\text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p))$ successfully unifies and yields $\langle p_1, l, p_2, \text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p)) \rangle$.

For two non-atomic expressions e_1 and e_2 to unify successfully, it must be true that either the top-level operator in e_1 and e_2 is the same data constructor (in which case an unification is attempted for each of their children), *or* the top-level operator in *at least* one of e_1 and e_2 is if-then-else. For example, unification of $\text{LCons}(e_1, e_2)$ and $\text{LCons}(e'_1, e'_2)$ proceeds by recursive attempts to unify e_1 with e'_1 , and e_2 with e'_2 respectively. On the other hand, unification of $\text{LCons}(e_1, e_2)$ and LNil fails due to mismatched top-level data constructors.

If the top-level operator in *exactly one* of e_1 and e_2 (say e_2) is if-then-else, then e_1 must have a data constructor at its root. Given $e_2 = \text{if } c \text{ then } e_2^{\text{th}} \text{ else } e_2^{\text{el}}$, we first attempt to unify e_1 with the if branch e_2^{th} — if unification succeeds, we also unify c (then condition) with **true**. Otherwise, we unify e_1 with the else branch e_2^{el} and $\neg c$ (else condition) with **true**. Consider the unification of a **List** variable l expanded through its unrolling procedure (i.e. if l is LNil then LNil else $\text{LCons}(l.\text{val}, l.\text{tail})$), and the expression $\text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p))$. The first unification

attempt (then branch) unifying LNil and $\text{LCons}(42, \text{Clist}_m^{\text{lnode}}(p))$ fails due to mismatched data constructors. The second unification attempt (else branch) results in the unification of – (a) $\neg l$ is LNil with *true*, and (b) $\text{LCons}(l.\text{val}, l.\text{tail})$ with $\text{LCons}(42, \text{Clist}_m^{\text{lnode}}(p))$, which eventually succeeds and returns correlation tuples.

If the top-level operator in both e_1 and e_2 is if-then-else, we unify each child (condition and branch expressions) of the corresponding if-then-else operators. Recall that the if-then-else operator (introduced in section 2.2) for an ADT T must have exactly one branch for each data constructor of T , and the branch associated with the data constructor V has V in its top-level. Whenever we descend down an if-then-else operator, we conjunct the if condition (if then branch is taken) or its negation (if else branch is taken) with its associated expression path condition. This allows us to keep track of the expression path conditions for both expressions during recursive descent into their children. For example, consider the unification of if c_1 then LNil else $\text{LCons}(0, l)$ and if c_2 then LNil else $\text{LCons}(i, \text{Clist}_m^{\text{lnode}}(p))$ under expression path conditions p_1 and p_2 respectively. The above results in the following three recursive calls to the unification procedure θ – (a) $\theta(p_1, c_1, p_2, c_2)$, (b) $\theta(p_1 \wedge c_1, \text{LNil}, p_2 \wedge c_2, \text{LNil})$, (c) $\theta(p_1 \wedge \neg c_1, \text{LCons}(0, l), p_2 \wedge \neg c_2, \text{LCons}(i, \text{Clist}_m^{\text{lnode}}(p)))$. The pseudocode for the unification procedure is given in section 4.4.3.

3.2.4 Unification under Rewriting

Recall that our unification procedure θ either fails or successfully unifies two expressions yielding correlation tuples relating atoms to (possibly) non-atomic expressions. Given two *canonical* expressions e_a and e_b at expression path conditions p_a and p_b respectively, an *iterative unification and rewriting procedure* $\Theta(p_a, e_a, p_b, e_b)$ is used to identify a set of correlation tuples relating *only* atoms in the two expressions. This iterative procedure begins with an attempt to unify e_a and e_b . If this unification fails, we return a failure for the original expressions e_a and e_b . Else, we obtain correlation tuples between atoms and expressions (with their expression path conditions). If the unification correlates an atom a_1 at expression path condition p_1 with another atom a_2 at expression path condition p_2 , we add $\langle p_1, a_1, p_2, a_2 \rangle$ to the final output. Otherwise, if the

unification correlates an atom a_1 at expression path condition p_1 to a non-atomic expression e_2 at expression path condition p_2 , we *rewrite* a_1 using its unrolling procedure to obtain expression e_1 . The unification algorithm then proceeds by unifying e_1 and e_2 through a recursive call to $\Theta(p_1, e_1, p_2, e_2)$.

For example, consider the unification of a `List` variable l with $\text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p))$ through θ , yielding the only correlation tuple $\langle p_1, l, p_2, \text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p)) \rangle$. The above tuple relates an atom with a non-atom, and hence Θ rewrites l using its unrolling procedure (in eq. (2.1)) to `if l is LNil then LNil else LCons($l.\text{val}$, $l.\text{tail}$)`. This is followed by another unification attempt which returns the correlation tuples: $\langle \text{true}, \neg l \text{ is LNil}, \text{true}, \text{true} \rangle$, $\langle \neg l \text{ is LNil}, l.\text{val}, \text{true}, 42 \rangle$ and $\langle \neg l \text{ is LNil}, l.\text{tail}, \text{true}, \text{Clist}_{\text{m}}^{\text{lnode}}(p) \rangle$. The tuples relate only atomic expressions and thus Θ terminates successfully, yielding the same correlation tuples. In general, the maximum number of rewrites performed by $\Theta(p_a, e_a, p_b, e_b)$ (before termination) is bounded by the sum of number of ADT data constructors in e_a and e_b . The pseudocode for iterative unification and rewriting is given in section 4.4.4.

3.2.5 Decomposition of Recursive Relations

For a recursive relation $l_1 \sim l_2$, we unify (canonicalized) l_1 and l_2 through a call to $\Theta(\text{true}, l_1, \text{true}, l_2)$. If the n tuples obtained after a successful unification are $\langle p_1^i, a_1^i, p_2^i, a_2^i \rangle$ (for $i = 1 \dots n$), then the *decomposition* of $l_1 \sim l_2$ is defined as:

$$l_1 \sim l_2 \Leftrightarrow \bigwedge_{i=1}^n ((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i)) \quad (3.2)$$

Recall that the unification of l and $\text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p))$ yields the correlation tuples: $\langle \text{true}, \neg l \text{ is LNil}, \text{true}, \text{true} \rangle$, $\langle \neg l \text{ is LNil}, l.\text{val}, \text{true}, 42 \rangle$ and $\langle \neg l \text{ is LNil}, l.\text{tail}, \text{true}, \text{Clist}_{\text{m}}^{\text{lnode}}(p) \rangle$. Consequently, the recursive relation $l \sim \text{LCons}(42, \text{Clist}_{\text{m}}^{\text{lnode}}(p))$ decomposes into¹:

¹ $(l \text{ is LCons})$ is equivalent to $\neg(l \text{ is LNil})$. In general, for an ADT value v of type T (with data constructors V_1, V_2, \dots, V_k), exactly one of $(v \text{ is } V_i)$ is true.

$$(l \text{ is LCons}) \wedge (l \text{ is LCons} \rightarrow l.\text{val} = 42) \wedge (l \text{ is LCons} \rightarrow l.\text{next} \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p))$$

In case of a failed unification, the *decomposition* is defined to be *false*, e.g., $\text{LNil} \sim \text{LCons}(0, l)$ decomposes into *false*.

Each conjunctive clause of the form $((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i))^2$ in the decomposition is called a *decomposition clause*. A decomposition clause may relate only atomic values, i.e., it may relate (a) two scalars or (b) two pseudo-variable(s) and/or lifted expressions. However, we restrict the shapes of recursive relation invariants such that each recursive relation in its decomposition *strictly* relates ADT pseudo-variables to lifted expressions. The invariant shapes along with the invariant inference procedure is presented in section 4.3. We *decompose* a recursive relation by replacing it with its decomposition. We *decompose* a proof obligation by decomposing all recursive relations in it.

3.3 Categorization of Proof Obligations

3.3.1 k -unrolling of Recursive Relations

Consider a recursive relation $l_1 \sim l_2$. We *unroll* $l_1 \sim l_2$ by rewriting the top-level lifted expressions in l_1 and l_2 using their respective unrolling procedures *and* decomposing the new recursive relation. We *unroll an expression* e by unrolling each recursive relation in e . More generally, the k -unrolling of e is found by unrolling the $(k - 1)$ -unrolling of e recursively. For example, the one-unrolling of $\text{LHS} \Rightarrow l \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)$ is given by the decomposition of $\text{LHS} \Rightarrow l \sim \text{if } p = 0 \text{ then LNil else LCons}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}, \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next}))$, which evaluates to:

$$\begin{aligned} \text{LHS} \Rightarrow (l \text{ is LNil}) &= (p = 0) \wedge (l \text{ is LCons} \wedge p \neq 0) \rightarrow (l.\text{val} = p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}) \\ &\quad \wedge (l \text{ is LCons} \wedge p \neq 0) \rightarrow (l.\text{tail} \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next})) \end{aligned}$$

²If a_1^i and a_2^i are ADT values, we replace $a_1^i = a_2^i$ with $a_1^i \sim a_2^i$.

3.3.2 Categorization Procedure

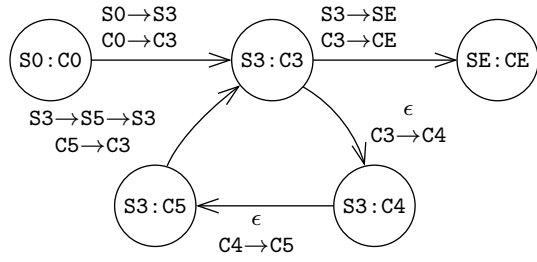
For a decomposed proof obligation $P_D : \text{LHS} \Rightarrow \text{RHS}$, we identify its k -unrolling (say P_K), where k is a fixed parameter of the algorithm called the *unrolling parameter*. We use a small value for k e.g., five. After k -unrolling, we *eliminate* those decomposition clauses $(p_1 \wedge p_2) \rightarrow (a_1 = a_2)$ in P_K whose antecedent $(p_1 \wedge p_2)$ evaluates to false under LHS (*ignoring* all recursive relations), yielding an equivalent proof obligation, say P_E . For example, the one-unrolling of $P : \text{LHS} \Rightarrow l \sim \text{Clist}_m^{\text{Inode}}(0)$, after elimination, yields $P_E : \text{LHS} \Rightarrow l$ is LNil. We categorize a proof obligation $P : \text{LHS} \Rightarrow \text{RHS}$ based on the k -unrolled form of its decomposition (i.e. P_E) as follows:

- Type I: P_E does not contain recursive relations
- Type II: P_E contains recursive relations *only* in the LHS
- Type III: P_E contains recursive relations in the RHS

Observe that the process of elimination of decomposition clauses (i.e. $P_K \rightarrow P_E$) itself requires a precise solver to identify *all* decomposition clauses that may be eliminated. An imprecise (but sound) procedure may only identify a subset of these decomposition clauses and evidently classify a type I (and II) proof obligation as type II (and III). Our proof discharge algorithm ensures that it is sound as long as the conversion of P_K to P_E is sound, i.e. we *only* remove those decomposition clauses whose antecedent definitely evaluates to false. Henceforth, we will simply use k -unrolling of P to refer to P_E directly. Next, we describe the algorithm for each type of proof obligations in sections 3.4 to 3.6.

3.4 Handling Type I Proof Obligations

Recall the product-CFG and its associated node invariants shown in fig. 3.2, representing a bisimulation relation for the `mk_list` procedure pair introduced in chapter 1. In fig. 3.2a, consider a proof obligation generated across the product-CFG edge $(S0:C0) \rightarrow (S3:C3)$ while checking if the



(a) Product-CFG between CFGs
in figs. 1.3a and 1.3b

PC-Pair	Invariants	
(S0:C0)	(P) $n_S = n_C$	
(S3:C3)	(I1) $n_S = n_C$	(I2) $i_S = i_C$
	(I3) $i_S \leq_u n_S$	(I4) $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(S3:C4)	(I5) $n_S = n_C$	(I6) $i_S = i_C$
(S3:C5)	(I7) $i_S <_u n_S$	(I8) $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(SE:CE)	(E) $\text{ret}_S \sim \text{Clist}_m^{\text{lnode}}(\text{ret}_C)$	

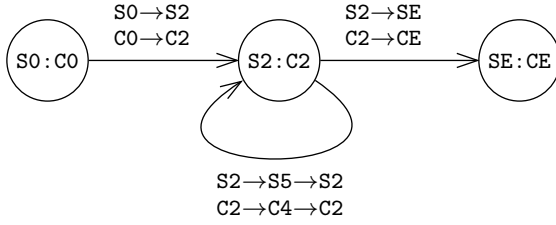
(b) Node invariants for product-CFG in fig. 1.4a

Figure 3.2: Product-CFG between the CFGs in figs. 1.3a and 1.3b. Figure 1.4b contains the corresponding node invariants for the product-CFG.

(I4) invariant in fig. 3.2b, $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$ holds at (S3:C3): $\{\phi_{S0:C0}\}(S0 \rightarrow S3, C0 \rightarrow C3)\{l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)\}$. The precondition $\phi_{S0:C0} \equiv (n_S = n_C)$ does not contain a recursive relation. When lowered to first-order logic through $\text{WP}_{S0 \rightarrow S3, C0 \rightarrow C3}$, this translates to $n_S = n_C \Rightarrow \text{LNil} \sim \text{Clist}_m^{\text{lnode}}(0)$. Here, LNil is obtained for l_S and 0 (null) is obtained for l_C . The one-unrolled form of this proof obligation yields $n_S = n_C \Rightarrow \text{true}$ which trivially resolves to true.

Consider the following example of a proof obligation: $\{\phi_{S0:C0}\}(S0 \rightarrow S3 \rightarrow S5 \rightarrow S3, C0 \rightarrow C3)\{l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)\}$. Notice, we have changed the path in \mathcal{S} (with CFG fig. 1.3a) to $S0 \rightarrow S3 \rightarrow S5 \rightarrow S3$ here. In this case, the corresponding first-order logic formula evaluates to: $(n_S = n_C) \wedge (0 <_u n_S) \Rightarrow \text{LCons}(0, \text{LNil}) \sim \text{Clist}_m^{\text{lnode}}(0)$, where $(0 <_u n_S)$ is the path condition for the path $S0 \rightarrow S3 \rightarrow S5 \rightarrow S3$. One-unrolling of this proof obligation decomposes RHS into false due to failed unification of LCons and LNil. The proof obligation is further discharged using an SMT solver which provides a counterexample (model) that evaluates the formula to false. For example, the counterexample $\{n_S \mapsto 42, n_C \mapsto 42\}$ evaluates this formula to false. These counterexamples assist in faster convergence of our correlation search and invariant inference procedures (as we will discuss later in sections 4.2 and 4.3).

Thus for type I queries, k -unrolling reduces all (if any) recursive relations in the original proof obligation into scalar equalities. The resulting query is further discharged using an SMT solver. Section 4.4 contains a deeper analysis of the following aspects of our proof discharge algorithm:



PC-Pair	Invariants
(S0:C0)	(P) $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(S2:C2)	(I1) $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$ (I2) $\text{sum}_S = \text{sum}_C$
(SE:CE)	(E) $\text{ret}_S = \text{ret}_C$

(a) Product-CFG between figs. 2.5a and 2.5b (b) Node invariants for product-CFG in fig. 2.7a

Figure 3.3: Product-CFG between the CFGs in figs. 2.5a and 2.5b. Figure 2.7b contains the corresponding node invariants for the product-CFG.

(a) translation of formula to SMT logic (section 4.4.7), and (b) reconstruction of counterexamples from models returned by the SMT solver (section 4.4.8). Assuming a capable enough SMT solver, all proof obligations in type I can be discharged precisely, i.e., we can always decide whether the proof obligation evaluates to true or false. If it evaluates to false, we also obtain counterexamples.

3.5 Handling Type II Proof Obligations

Recall the product-CFG and its node invariants shown in fig. 3.3, representing a bisimulation relation for the `sum_list` procedure pair introduced in chapter 2. Consider the proof obligation for (I2) invariant $\text{sum}_S = \text{sum}_C$ across edge $(S2:C2) \rightarrow (S2:C2)$ in fig. 3.3a: $\{\phi_{S2:C2}\}(S2 \rightarrow S5 \rightarrow S2, C2 \rightarrow C4 \rightarrow C2)\{\text{sum}_S = \text{sum}_C\}$, where the node invariant $\phi_{S2:C2}$ contains the recursive relation $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$. The corresponding (simplified) first-order logic formula for this proof obligation is: $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C) \wedge (\text{sum}_S = \text{sum}_C) \wedge (l_S \text{ is LCons}) \wedge (l_C \neq 0) \Rightarrow (\text{sum}_S + l_S.\text{val}) = (\text{sum}_C + l_C \xrightarrow{\text{lnode}} \text{val})$. We fail to remove the recursive relation on the LHS even after k -unrolling for any finite unrolling parameter k because both sides of \sim represent list values of arbitrary length. In such a scenario, we do not know of an efficient SMT encoding for the recursive relation $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$. Ignoring this recursive relation will incorrectly (although soundly) evaluate the proof obligation to false; however, for a successful equivalence proof, we need the proof discharge algorithm to evaluate it to true. Let's call this requirement (R1).

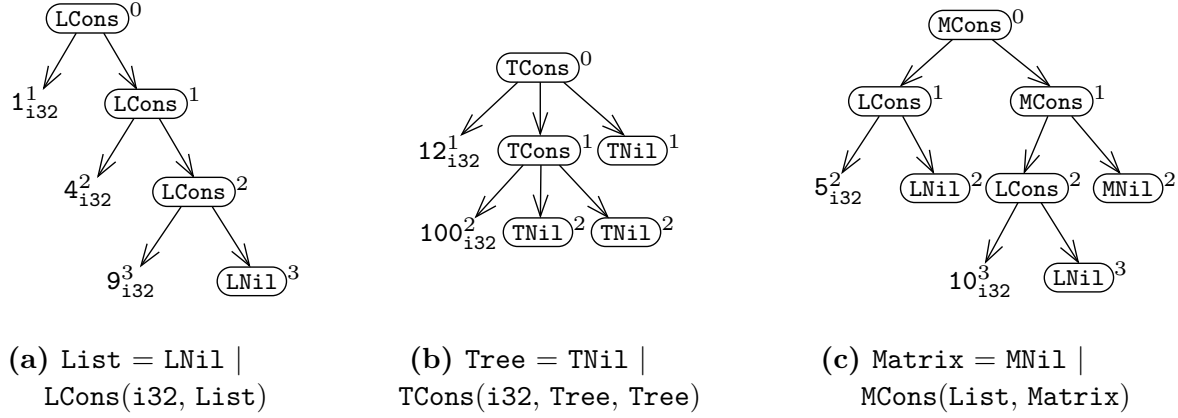


Figure 3.4: Expression trees of three values, each of type **List**, **Tree** and **Matrix** respectively. The depths are shown as superscripts for each node in the trees.

Now, consider the proof obligation formed by correlating two iterations of the loop in \mathcal{S} (with CFG fig. 2.5a) with one iteration of the loop in \mathcal{C} (with CFG fig. 2.5b): $\{\phi_{s2:c2}\}(S2 \rightarrow S5 \rightarrow S2 \rightarrow S5 \rightarrow S2, C2 \rightarrow C4 \rightarrow C2)\{\text{sum}_S = \text{sum}_C\}$. The equivalent first-order logic formula is: $l_S \sim \text{Clist}_{\text{m}}^{\text{lnode}}(l_C) \wedge (\text{sum}_S = \text{sum}_C) \wedge (l_S \text{ is LCons}) \wedge (l_S.\text{tail} \text{ is LCons}) \Rightarrow (\text{sum}_S + l_S.\text{val} + l_S.\text{tail.val}) = (\text{sum}_C + l_C \xrightarrow{\text{m}}_{\text{lnode}} \text{val})$. Similar to the prior proof obligation, its equivalent first-order logic formula contains a recursive relation in the LHS. Clearly, this proof obligation should evaluate to false. Whenever a proof obligation evaluates to false, we expect an ideal proof discharge algorithm to generate counterexamples that falsify the proof obligation. Let's call this requirement $\textcircled{\text{R2}}$. Recall that these counterexamples help in faster convergence of our correlation search and invariant inference procedures.

To tackle requirements $\textcircled{\text{R1}}$ and $\textcircled{\text{R2}}$, our proof discharge algorithm converts the original proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$ into two approximated proof obligations ($P_{\text{pre-o}} : \{\phi_s^{o_{d1}}\}(e)\{\phi_d\}$) and ($P_{\text{pre-u}} : \{\phi_s^{u_{d2}}\}(e)\{\phi_d\}$). Here $\phi_s^{o_{d1}}$ and $\phi_s^{u_{d2}}$ represent the over- and under-approximated versions of precondition ϕ_s respectively, and d_1 and d_2 represent *depth parameters* that indicate the degree of over- and under-approximation. To explain our over- and under-approximation scheme, we first introduce the notion of *depth* for an ADT value.

3.5.1 Depth of ADT Values

To define depth of an ADT value v , we view the value as its expression tree. The internal nodes represent ADT data constructors and the leafs (also called *terminals*) represent scalar constants (e.g. bitvector literals). The depth of a data constructor or a scalar literal in v is simply the depth of its associated node in the expression tree representation of v . The *depth* of ADT value v is defined as the depth of its expression tree. For example, the depth of $\text{LCons}(1, \text{LCons}(4, \text{LNil}))$ is 2. Figure 3.4 shows the expression tree and their corresponding node depths for three values, each of type **List**, **Tree** and **Matrix** respectively.

3.5.2 Approximation of Recursive Relations

The d -depth overapproximation of a recursive relation $l_1 \sim l_2$, denoted by $l_1 \sim_d l_2$, represents the condition that l_1 and l_2 are *recursively equal up to depth d* . i.e., l_1 and l_2 have identical structures and all *terminals* at depths $\leq d$ in the trees of both values are equal (under the precondition that the terminals exist); however, terminals at depths $> d$ may have different values. $l_1 \sim_d l_2$ (for finite d) is a weaker condition than $l_1 \sim l_2$ (i.e. overapproximation). The true equality i.e. $l_1 \sim l_2$ can be thought of as equality of structures and all terminals up to an unbounded depth i.e. $l_1 \sim_\infty l_2$.

The d -depth underapproximation of a recursive relation $l_1 \sim l_2$ is written as $l_1 \approx_d l_2$, where \approx_d represents the condition that l_1 and l_2 are *recursively equal and bounded to depth d* , i.e., l_1 and l_2 have a maximum depth $\leq d$ and they are recursively equal up to depth d . Thus, $l_1 \approx_d l_2$ is equivalent to $\Gamma_d(l_1) \wedge \Gamma_d(l_2) \wedge l_1 \sim_d l_2$, where $\Gamma_d(l)$ represents the condition that the maximum depth of l is d . $l_1 \approx_d l_2$ (for finite d) is a stronger condition than $l_1 \sim l_2$ (i.e. underapproximation) as it bounds the depth to d while also ensuring equality till depth d . For arbitrary depths a and b ($a \leq b$), the approximations of $l_1 \sim l_2$ are related as follows:

$$l_1 \approx_a l_2 \Rightarrow l_1 \approx_b l_2 \Rightarrow l_1 \sim l_2 \Rightarrow l_1 \sim_b l_2 \Rightarrow l_1 \sim_a l_2 \quad (3.3)$$

3.5.3 Reduction of Approximate Recursive Relations

Unlike the original recursive relation $l_1 \sim l_2$, its approximations $l_1 \sim_d l_2$ and $l_1 \approx_d l_2$ can be reduced into equivalent conditions absent of recursive relations. Hence, these approximations can be encoded in and subsequently discharged by a SMT solver.

- $l_1 \sim_d l_2$ is equivalent to the condition that the tree structures of l_1 and l_2 are identical till depth d and the corresponding terminal values in both d -depth identical structures are also equal. Note that these conditions only require scalar equalities. $l_1 \sim_d l_2$ can be identified through unification of l_1 and l_2 till depth d . This algorithm is similar to the ‘iterative unification and rewriting procedure’ in section 3.2 and further described in section 4.4.6. In this modified unification algorithm, we eagerly expand atomic ADT expressions till depth d , in contrast to the ‘iterative unification and rewriting procedure’ which terminates whenever a correlation tuple relates (possibly ADT) atomic expressions. Finally, we only keep those correlation tuples at depth $\leq d$ that relate scalar values and discard the recursive relations.

For example, the condition $l \sim_1 \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)$ is computed through iterative unification and rewriting till depth one; yielding the correlation tuples: $\langle \text{true}, l \text{ is LNil}, \text{true}, p = 0 \rangle$, $\langle l \text{ is LCons}, l.\text{val}, p \neq 0, p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val} \rangle$ and $\langle l \text{ is LCons}, l.\text{tail}, p \neq 0, \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next}) \rangle$. Keeping only those correlation tuples that relate scalar expressions, the above condition reduces to the SMT-encodable predicate:

$$((l \text{ is LNil}) = (p = 0)) \wedge ((l \text{ is LCons}) \wedge (p \neq 0) \rightarrow l.\text{val} = p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val})$$

- Recall that $l_1 \approx_d l_2 \Leftrightarrow \Gamma_d(l_1) \wedge \Gamma_d(l_2) \wedge l_1 \sim_d l_2$. $\Gamma_d(l)$ is equivalent to the condition that the tree nodes at depths $> d$ are unreachable. This is achieved through expanding (canonicalized) l through rewriting till depth d and asserting the unreachability of if-then-else paths that reach nodes with depths $> d$ (i.e. asserting the negation of their expression path conditions). For example, for a **List** variable l , the condition $\Gamma_2(l)$ is equivalent to $(l \text{ is LNil}) \vee ((l \text{ is LCons}) \wedge (l.\text{tail} \text{ is LNil}))$. Similarly, $\Gamma_2(\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p))$ is equivalent to $(p = 0) \vee ((p \neq 0) \wedge (p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next} = 0))$. Finally,

$l \approx_2 \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p) \Leftrightarrow \Gamma_2(l) \wedge \Gamma_2(\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)) \wedge l \sim_2 \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)$. The algorithms for reduction of over- and under-approximate recursive relations are given in section 4.4.6.

3.5.4 Summary of Type II Proof Discharge Algorithm

We over- (under-) approximate a precondition ϕ till depth d by d -depth over- (under-) approximating each recursive relation occurring in ϕ . Due to the conjunctive recursive relation property (defined in section 3.1.2), the over- and under-approximation of ϕ are also weaker and stronger conditions compared to ϕ respectively. For a type II proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$, we first submit the proof obligation $(P_{pre-o} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\})$ to the SMT solver. Recall that the precondition $\phi_s^{o_{d_1}}$ is the d_1 -depth overapproximated version of ϕ_s . If the SMT solver evaluates P_{pre-o} to true, then we return true for the original proof obligation P — if the Hoare triple with an overapproximate precondition holds, then the original Hoare triple also holds.

If the SMT solver evaluates P_{pre-o} to false, then we submit the proof obligation $(P_{pre-u} : \{\phi_s^{u_{d_2}}\}(e)\{\phi_d\})$ to the SMT solver. Recall that the precondition $\phi_s^{u_{d_2}}$ is the d_2 -depth underapproximated version of ϕ_s . If the SMT solver evaluates P_{pre-u} to false, then we return false for the original proof obligation P — if the Hoare triple with an underapproximate precondition does not hold, then the original Hoare triple also does not hold. Further, a counterexample that falsifies P_{pre-u} would also falsify P , and is thus a valid counterexample for use in our correlation search and invariant inference procedures.

Finally, if the SMT solver evaluates P_{pre-u} to true, then we have neither proven nor disproven P . In this case, we imprecisely (but soundly) return false for the original proof obligation P (without counterexamples). Note that both approximations of P strictly fall in type I and are discharged as discussed in section 3.4.

Revisiting our examples, the proof obligation $\{\phi_{s2:c2}\}(\text{S2} \rightarrow \text{S5} \rightarrow \text{S2}, \text{C2} \rightarrow \text{C4} \rightarrow \text{C2})\{\text{sum}_S = \text{sum}_C\}$ is provable using a depth 1 overapproximation of the precondition $\phi_{s2:c2}$ — the depth 1 overapproximation retains the information that the first value in lists l_S and $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(\text{l}_C)$ are equal, and that is sufficient to prove that the new values of sum_S and sum_C are also equal (given that

the old values are equal, as encoded in $\phi_{S2:C2}$).

Similarly, the proof obligation $\{\phi_{S2:C2}\}(S2 \rightarrow S5 \rightarrow S2 \rightarrow S5 \rightarrow S2, C2 \rightarrow C4 \rightarrow C2)\{\text{sum}_S = \text{sum}_C\}$ successfully evaluates to false using a depth 2 underapproximation of the precondition $\phi_{S2:C2}$. In the depth 2 underapproximate version, we try to prove that if the equal lists l_S and $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)$ have exactly two nodes³, then the sum of the two values in l_S is equal to the value stored in the first node in l_C . This proof obligation will return counterexample(s) that map program variables to their concrete values. The following is a possible counterexample to the depth 2 underapproximate proof obligation.

```
{  sumS ↦ 3,
   sumC ↦ 3,
   lS   ↦ LCons(42, LCons(43, LNil)),
   lC   ↦ 0x123,
   m     ↦ { 0x123 ↦lnode (.val ↦ 42, .next ↦ 0x456),
              0x456 ↦lnode (.val ↦ 43, .next ↦ 0),
              ()   ↦ 77
            }
}
```

This counterexample maps variables to values (e.g., sum_S maps to an i32 value 3 and l_S maps to a List value $\text{LCons}(42, \text{LCons}(43, \text{LNil}))$). Additionally, It maps the C program's memory state \mathfrak{m} to an array that maps the regions starting at addresses 0x123 and 0x456 (regions of size 'sizeof(lnode)') to memory objects of type `lnode` (with the `val` and `next` fields shown for each object). All other addresses (except the ones for which an explicit mapping is available), \mathfrak{m} provides a default byte-value 77 (shown as $() \mapsto 77$) in this counterexample.

This counterexample satisfies the preconditions $l_S \approx_2 \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)$, $\text{sum}_S = \text{sum}_C$ and the path conditions. Further, when the paths $S2 \rightarrow S5 \rightarrow S2 \rightarrow S5 \rightarrow S2$ and $C2 \rightarrow C4 \rightarrow C2$ are executed starting at the machine state represented by this counterexample, the resulting values of sum_S and sum_C are $3+42+43=88$ and $3+42=45$ respectively. Evidently, the counterexample falsifies the proof condition because these values are not equal (as required by the postcondition).

³The underapproximation restricts both lists to have at most two nodes; the path condition for $S2 \rightarrow S5 \rightarrow S2 \rightarrow S5 \rightarrow S2$ additionally restricts l_S to have at least two nodes. Together, this is equivalent to the list having exactly two nodes

3.6 Handling Type III Proof Obligations

In fig. 3.2a, consider a proof obligation generated across the product-CFG edge $(S3:C5) \rightarrow (S3:C3)$ while checking if the $(I4)$ invariant, $l_S \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)$, holds at $(S3:C3)$: $\{\phi_{S3:C5}\}(S3 \rightarrow S5 \rightarrow S3, C5 \rightarrow C3)\{l_S \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)\}$. Here, a recursive relation is present both in the precondition $\phi_{S3:C5}$ ($(I8)$) and in the postcondition $(I4)$ and we are unable to remove them after k -unrolling. When lowered to first-order logic through $\text{WP}_{S3 \rightarrow S5 \rightarrow S3, C5 \rightarrow C3}$, this translates to (showing only relevant relations):

$$(i_S = i_C \wedge p_C = \text{malloc}() \wedge l_S \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)) \Rightarrow (\text{LCons}(i_S, l_S) \sim \text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(p_C)) \quad (3.4)$$

On the RHS of this first-order logic formula, $\text{LCons}(i_S, l_S)$ is compared for equality with the lifted expression $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(p_C)$; here p_C represents the address of the newly allocated `lnode` object (through `malloc`) and \mathfrak{m}' represents the C memory state after executing the writes at lines C5 and C6 on the path $C5 \rightarrow C3$, i.e.,

$$\mathfrak{m}' \Leftrightarrow \mathfrak{m}[\text{addrof}(p_C \rightarrow \text{lnode } \text{val}) \leftarrow i_C]_{i32}[\text{addrof}(p_C \rightarrow \text{lnode } \text{next}) \leftarrow l_C]_{i32} \quad (3.5)$$

Recall that “ $\mathfrak{m}[a \leftarrow v]_T$ ” represents an array that is equal to \mathfrak{m} everywhere except at addresses $[a, a + \text{sizeof}(T))$ which contains the value v of type ‘T’. Consequently, \mathfrak{m}' is equal to \mathfrak{m} everywhere except at the `val` and `next` fields of the `lnode` object pointed to by p_C . We refer to these memory writes that distinguish \mathfrak{m} and \mathfrak{m}' , as the *distinguishing writes*.

3.6.1 LHS-to-RHS Substitution and RHS Decomposition

We start by utilizing the \sim relationships in the LHS (antecedent) of ‘ \Rightarrow ’ to rewrite eq. (3.4) so that the ADT variables (e.g., l_S) in its RHS (consequent) are substituted with the lifted \mathcal{C} values

(e.g., $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)$). Thus, we rewrite eq. (3.4) to:

$$\begin{aligned} & (i_S = i_C \wedge p_C = \text{malloc}() \wedge l_S \sim \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)) \\ & \Rightarrow (\text{LCons}(i_S, \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C)) \sim \text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(p_C)) \end{aligned} \quad (3.6)$$

Next, we decompose the RHS by decomposing the recursive relation in the RHS followed by RHS-breaking. This process reduces eq. (3.6) into the following smaller proof obligations (LHS denotes the antecedent of the proof obligation in eq. (3.6)): (a) $\text{LHS} \Rightarrow (p_C \neq 0)$, (b) $\text{LHS} \wedge (p_C \neq 0) \Rightarrow (i_S = p_C \xrightarrow[\text{lnode}]{\mathfrak{m}'} \text{val})$, and (c) $\text{LHS} \wedge (p_C \neq 0) \Rightarrow \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C) \sim \text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(p_C \xrightarrow[\text{lnode}]{\mathfrak{m}'} \text{next})$.

The first two proof obligations fall in type II and are discharged through over- and under-approximation schemes as discussed in section 3.5.4:

1. The first proof obligation with postcondition $(p_C \neq 0)$ evaluates to *true* because the LHS ensures that p_C is the return value of an allocation function (i.e. `malloc`) which must be non-zero due to the $(\mathcal{C} \text{ fits})$ assumption.
2. The second proof obligation with postcondition $(i_S = p_C \xrightarrow[\text{lnode}]{\mathfrak{m}'} \text{val})$ also evaluates to *true* because i_C is written at address $p_C + \text{offsetof}(\text{lnode}, \text{val})$ in \mathfrak{m}' (eq. (3.5)) and the LHS ensures that $i_S = i_C$.

For ease of exposition, we simplify the postcondition of the third proof obligation by rewriting $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(p_C \xrightarrow[\text{lnode}]{\mathfrak{m}'} \text{next})$ to $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(l_C)$. This simplification is valid because l_C is written to address $p_C + \text{offsetof}(\text{lnode}, \text{next})$ in \mathfrak{m}' (eq. (3.5)). Also, we have already shown that $(p_C \neq 0)$ holds due to the $(\mathcal{C} \text{ fits})$ assumption. This simplification-based rewriting is only done for ease of exposition, and has no effect on the completeness of the algorithm. Thus, the third proof obligation can be rewritten as a recursive relation between two lifted expressions:

$$\text{LHS} \Rightarrow \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C) \sim \text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(l_C) \quad (3.7)$$

Hence, we are interested in proving equality between two `List` values lifted from \mathcal{C} values under a

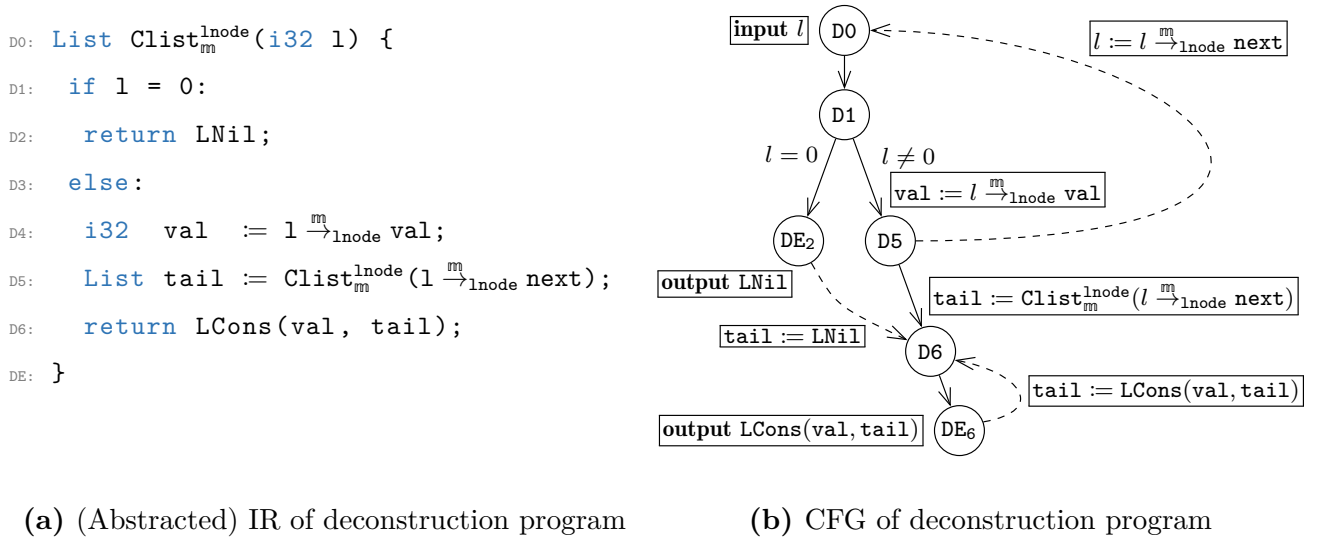
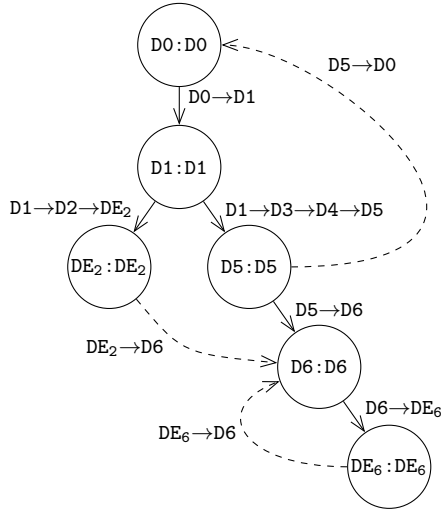


Figure 3.5: IR and CFG representation of deconstruction program based on the lifting constructor $\text{Clist}^{\text{lnode}}_m$ defined in eq. (2.2). The edge $D5 \rightarrow D6$ contains a recursive function call. In fig. 3.5b, the square boxes show the transfer functions for the deconstruction program. The dashed edges represent the recursive function call in the CFG representation as shown in fig. 3.5b.

precondition. Next, we show how the above can be reposed as the problem of showing equivalence between two procedures through bisimulation.

3.6.2 Deconstruction Programs for Lifted Values

Consider a program that recursively calls the definition (i.e. unrolling procedure) of $\text{Clist}^{\text{lnode}}_m$ (eq. (2.2)) to deconstruct $\text{Clist}^{\text{lnode}}_m(l)$. For example, $\text{Clist}^{\text{lnode}}_m(l)$ may yield a recursive call to $\text{Clist}^{\text{lnode}}_m(l \xrightarrow{m} \text{lnode next})$ and so on, until the argument becomes zero. This program essentially deconstructs $\text{Clist}^{\text{lnode}}_m(l)$ into its terminal (scalar) values and reconstructs a **List** value equal to the value represented by $\text{Clist}^{\text{lnode}}_m(l)$. We call this program a *deconstruction program* (denoted by \mathcal{D}) based on the lifting constructor $\text{Clist}^{\text{lnode}}_m$. Figure 3.5 show the IR and CFG representations for the deconstruction program based on $\text{Clist}^{\text{lnode}}_m$.



(a) Decons-PCFG

PC-Pair	Invariants
(D0:D0)	(P) $l^{fst} = l^{snd}$
(D1:D1)	(I1) $l^{fst} = l^{snd}$
(D5:D5)	(I2) $val^{fst} = val^{snd}$ (I3) $l^{fst} \xrightarrow{m}_{lnode} next = l^{snd} \xrightarrow{m'}_{lnode} next$
(D6:D6)	(I4) $val^{fst} = val^{snd}$ (I5) $tail^{fst} = tail^{snd}$
(DE2:DE2)	(E) $ret^{fst} = ret^{snd}$
(DE6:DE6)	

(b) Node invariants for decons-PCFG in fig. 3.6a

Figure 3.6: Decons-PCFG and its associated node invariants for the deconstruction programs based on $\text{Clist}_{\mathbb{m}}^{\text{lnode}}(l_C)$ and $\text{Clist}_{\mathbb{m}'}^{\text{lnode}}(l_C)$ respectively.

Theorem 1. Under an antecedent LHS, $\text{Clist}_{\mathbb{m}}^{\text{lnode}}(l_C) \sim \text{Clist}_{\mathbb{m}'}^{\text{lnode}}(l_C)$ holds if and only if the two deconstruction programs \mathcal{D}_1 and \mathcal{D}_2 , based on $\text{Clist}_{\mathbb{m}}^{\text{lnode}}(l_C)$ and $\text{Clist}_{\mathbb{m}'}^{\text{lnode}}(l_C)$, are equivalent. The equivalence must ensure that the observables generated by both programs (i.e. output **List** values) are equal, given that the input l_C is provided to both programs respectively and the antecedent LHS holds at the program entries.

Proof Sketch. The proof follows from noting that the only observables of \mathcal{D}_1 and \mathcal{D}_2 are their output **List** values. Also, the value represented by a lifted expression is equal to the output of its deconstruction program. Thus, a successful equivalence proof ensures equal values represented by the lifting constructors and vice versa. \square

Thus, to check if $\text{Clist}_{\mathbb{m}}^{\text{lnode}}(l_C) \sim \text{Clist}_{\mathbb{m}'}^{\text{lnode}}(l_C)$ holds; we instead check if a bisimulation relation exists between their respective deconstruction programs \mathcal{D}^{fst} and \mathcal{D}^{snd} (implying equivalence). Theorem 1 generalizes to arbitrary lifted expressions with potentially different arguments and memory states.

3.6.3 Checking Bisimulation between Deconstruction Programs

To check bisimulation, we attempt to show that both deconstructions proceed in lockstep, and the invariants at each step of this lockstep execution ensure equal observables. We use a product-CFG to encode this lockstep execution between \mathcal{D}^{fst} and \mathcal{D}^{snd} — to distinguish this product-CFG from the top-level product-CFG that relates \mathcal{S} and \mathcal{C} , we call this product-CFG that relates two deconstruction programs, a *deconstruction product-CFG* or *decons-PCFG* for short.

The decons-PCFG for the proof obligation in eq. (3.7) is shown in fig. 3.6a. We distinguish states between the first and second programs using superscripts: *fst* and *snd* respectively. However, these are omitted in case the states are equal in both programs (e.g. p_C). To check bisimulation between the programs that deconstruct $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(1_C)$ and $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(1_C)$ (i.e. \mathcal{D}^{fst} and \mathcal{D}^{snd} respectively), the decons-PCFG correlates one unrolling of the first program with one unrolling of the second program, as defined by the unrolling procedure in eq. (2.2). Thus, the PC-transition correlations of \mathcal{D}^{fst} and \mathcal{D}^{snd} are trivially obtained by unifying the static program structures as shown in fig. 3.6a. A node is created in the decons-PCFG that encodes the correlation of the entries of both programs; we call this node the *recursive-node* in the decons-PCFG (e.g. (D0:D0) in fig. 3.6a). A recursive call becomes a back-edge in the decons-PCFG that terminates at the recursive-node. Furthermore, a bisimulation check involves identification of invariants at correlated PC-pairs strong enough to ensure observable equivalence. At the start of both deconstruction programs, $\textcircled{\text{P}} \ 1^{fst} = 1^{snd} = 1_C$ — the same 1_C is passed to both \mathcal{D}^{fst} and \mathcal{D}^{snd} , only the memory states \mathfrak{m} and \mathfrak{m}' (defined in eq. (3.5)) are different. The observables include the returned `List` values at correlated program exits ($\text{DE}_2:\text{DE}_2$) and ($\text{DE}_6:\text{DE}_6$), which forms the postcondition (labeled $\textcircled{\text{E}}$ in fig. 3.6b). Next, the bisimulation check involves identification of inductive invariants (labeled $\textcircled{\text{I}}$ in fig. 3.6b) at correlated PC-pairs. The proof obligations arising due to this bisimulation check include:

1. The if condition ($1^{fst} = 0$) in \mathcal{D}^{fst} is equal to the corresponding if condition ($1^{snd} = 0$) in \mathcal{D}^{snd} : ($1^{fst} = 0$) = ($1^{snd} = 0$).
2. If the if condition evaluates to false in both \mathcal{D}^{fst} and \mathcal{D}^{snd} , then observable values val^{fst} and val^{snd} along the path (D1:D1) \rightarrow (D5:D5) (used in the construction of the output lists)

are equal. This forms the invariant $\textcircled{\text{I2}}$ in fig. 3.6b and lowers to the following proof obligation:

$$(1^{fst} \neq 0) \wedge (1^{snd} \neq 0) \Rightarrow 1^{fst} \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val} = 1^{snd} \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \text{val}.$$

3. If the **if** condition evaluates to false in both \mathcal{D}^{fst} and \mathcal{D}^{snd} , then the preconditions are satisfied at the beginning of the programs invoked through the recursive call. This involves checking that, along the path $(\text{D1:D1}) \rightarrow (\text{D5:D5})$, the actual arguments to the recursive call satisfies the precondition $\textcircled{\text{P}}$ at the beginning of the procedure i.e. the recursive-node (D0:D0) . This forms the invariant $\textcircled{\text{I3}}$ in fig. 3.6b and lowers the following proof obligation:
 $(1^{fst} \neq 0) \wedge (1^{snd} \neq 0) \Rightarrow 1^{fst} \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next} = 1^{snd} \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \text{next}.$

A successful discharge of the above invariant $(\textcircled{\text{I3}})$, by induction, ensures that postcondition $(\textcircled{\text{E}})$ is satisfied by the values returned by the recursive call at product-CFG node (D6:D6) . Hence, we can assume that invariant $\textcircled{\text{I5}}$ holds at (D6:D6) . This special case of correlating procedure call edges is further discussed in section 4.2.3 as part of our overall product-CFG construction algorithm.

The first check succeeds due to the precondition $\textcircled{\text{P}} \ 1^{fst} = 1^{snd}$ at the recursive-node. For the second and third checks, we additionally need to reason that the memory objects $1^{snd} \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \text{val}$ and $1^{snd} \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \text{next}$ cannot alias with the writes in \mathfrak{m}' (eq. (3.5)) to the newly allocated objects $\text{p}_C \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \text{val}$ and $\text{p}_C \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \text{next}$. We capture this aliasing information using a points-to analysis described next in section 3.6.4.

Notice that a bisimulation check between the deconstruction programs is significantly easier than the top-level bisimulation check between \mathcal{S} and \mathcal{C} : here, the correlation of PC traisitons is trivially identified by unifying the unrolling procedures of both lifted expressions, and the candidate invariants are obtained by equating each pair of terminal values that form the observables of both programs.

3.6.4 Points-to Analysis

To reason about aliasing (as required during bisimulation check in section 2.4), we conservatively compute *may-point-to* information for each program value using an interprocedural flow-sensitive version of Andersen’s algorithm [10]. The range of this computed may-point-to function is the set of *region labels*, where each region label identifies a set of memory objects. The sets of memory objects identified by two distinct region labels are necessarily disjoint. We write $p \rightsquigarrow \{R_1, R_{2+}\}$ to represent the condition that value p *may point to* an object belonging to one of the region labels R_1 or R_{2+} (but may not point to any object outside of R_1 and R_{2+}).

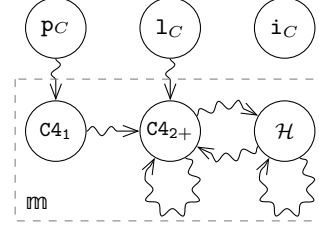
We populate the set of all region labels using *allocation sites* of C i.e., PCs where a call to `malloc` occurs. For example, **C4** in fig. 1.2b is an allocation site. For each allocation site A , we create two region labels: (a) the first region label, called A_1 , identifies the set of memory objects that were allocated by the most recent execution of A , and (b) the second region label, called A_{2+} , identifies the set of memory objects that were allocated by older (not the most recent) executions of A . We also include a special heap region, \mathcal{H} to represent the rest of the memory not covered by the allocation site regions associated with `malloc` calls.

For example, at the start of PC **C7** in fig. 1.2b, $i_C \rightsquigarrow \emptyset$, $p_C \rightsquigarrow \{\mathbf{C4}_1\}$, and $l_C \rightsquigarrow \{\mathbf{C4}_{2+}\}$. Since the may-point-to analysis determines the sets of objects pointed-to by p_C and l_C to be disjoint, ($\mathbf{C4}_1$ against $\mathbf{C4}_{2+}$), any memory accessed through p_C and l_C cannot alias at **C7** (for accesses within the bounds of the allocated objects).

The may-point-to information is computed not just for program values (e.g., p_C , l_C) but also for each region label. For region labels R_1 , R_2 and R_3 : $R_1 \rightsquigarrow \{R_2, R_3\}$ represents the condition that the values (pointers) stored in objects identified by R_1 may point to objects identified by either R_2 or R_3 (but not to any other object outside R_2 and R_3). In fig. 1.2b, at PC **C7**, we get $\mathbf{C4}_1 \rightsquigarrow \{\mathbf{C4}_{2+}\}$ and $\mathbf{C4}_{2+} \rightsquigarrow \{\mathbf{C4}_{2+}, \mathcal{H}\}$. The condition $\mathbf{C4}_1 \rightsquigarrow \{\mathbf{C4}_{2+}\}$ holds because the `next` pointer of the object pointed-to by p_C (which is a $\mathbf{C4}_1$ object at **C7**) may point to a $\mathbf{C4}_{2+}$ object (e.g., object pointed to by l_C). On the other hand, pointers within a $\mathbf{C4}_{2+}$ object may not point to a $\mathbf{C4}_1$ object.

Points-to Invariants	
(J1) $p_C \rightsquigarrow \{C4_1\}$	(J2) $C4_1 \rightsquigarrow \{C4_{2+}\}$
(J3) $l_C \rightsquigarrow \{C4_{2+}\}$	(J4) $C4_{2+} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$
(J5) $i_C \rightsquigarrow \emptyset$	(J6) $\mathcal{H} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$

(a) Points-to invariants at C5 of fig. 1.2b



(b) Points-to graph at C5 of fig. 1.2b

Figure 3.7: Points-to invariants at C5 of \mathcal{C} in fig. 1.2b. Figure 3.7b shows the graphical representation of the relations in fig. 3.7a. Each node represents a \mathcal{C} pseudo-register or a memory region in \mathfrak{m} . An edge $A \rightarrow R$ represents the condition that A (or objects in A in case of a memory region) may point to the memory region R .

3.6.5 Transferring Points-to Information to Decons-PCFG

Recall that in section 3.6.3, we reduce the condition $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(l_C) \sim \text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(l_C)$ to an equivalence check between their deconstruction programs: \mathcal{D}^{fst} and \mathcal{D}^{snd} . Also, recall that we discharge the equivalence check through construction of a decons-PCFG encoding the lockstep execution of the two deconstruction programs. During this bisimulation check, we need to prove that, $l^{fst} \xrightarrow{\mathfrak{m}}_{\text{lnode}} \{\text{val}, \text{next}\}$ and $l^{snd} \xrightarrow{\mathfrak{m}'}_{\text{lnode}} \{\text{val}, \text{next}\}$ are equal in decons-PCFG node (D1:D1) shown in fig. 3.6b. Recall that the invariant (I1) asserts $l^{fst} = l^{snd}$. Thus, to successfully discharge these proof obligations, it suffices to show that l^{snd} cannot alias with the memory writes that distinguish \mathfrak{m} and \mathfrak{m}' .

Figure 3.7a shows the may-point-to relations identified by our points-to analysis on \mathcal{C} in fig. 1.2b at the program point C5. The points-to analysis determines that at C5 (i.e. start of the product-CFG edge (S3:C5) \rightarrow (S3:C3) across which the proof obligation is generated), the pointer to the *head* of the list, i.e. (J3) $l_C \rightsquigarrow \{C4_{2+}\}$. It also determines that the distinguishing writes (in eq. (3.5)) modify memory regions belonging to $C4_1$ only ((J1)). Further, we get (J4) $C4_{2+} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$ at PC C5. Figure 3.7b shows the points-to graph for the \mathcal{C} variables and the memory regions (in \mathfrak{m}). This graphical representation clearly illustrates that the objects pointed to by p_C (i.e. $C4_1$) and l_C (i.e. $C4_{2+}$) are mutually isolated.

However, notice that these determinations only rule out aliasing of the list-head with the distinguishing writes. We also need to confirm non-aliasing of the internal nodes of the linked list with the distinguishing writes. For this, we need to identify a points-to invariant: $1^{snd} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$, at the recursive-node of the decons-PCFG (shown in fig. 3.6a). To identify such points-to invariants, we run our points-to analysis on the deconstruction programs (i.e. \mathcal{D}^{fst} and \mathcal{D}^{snd}) before comparing them for equivalence. To model procedure calls, A *supergraph* is created with edges representing control flow to (and from) the entry (and exits) of the program respectively (e.g., dashes edges in fig. 3.5b). To see why $1^{snd} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$ is an inductive invariant at (D0:D0):

(base case) The invariant holds at entry of the decons-PCFG because $1^{snd} = 1_C$ at entry and $\textcircled{J3} 1_C \rightsquigarrow \{C4_{2+}\}$, which is a stronger condition.

(inductive step) If $1^{snd} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$ holds at the entry node, it also holds at the start of a recursive call. This follows from $\textcircled{J4} C4_{2+} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$ and $\textcircled{J6} \mathcal{H} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$ (points-to information at PC C5), which ensures that $1^{snd} \xrightarrow{m'}_{\text{lnode}} \text{next}$ may point to only $C4_{2+}$ and \mathcal{H} objects.

The same analysis is run for both \mathcal{C} , and the deconstruction programs \mathcal{D}^{fst} and \mathcal{D}^{snd} . For a deconstruction program \mathcal{D} , the boundary condition (at entry) for the points-to analysis is based on the results of the points-to analysis on \mathcal{C} at the PC where the proof obligation is being discharged. For example, the points-to information of \mathcal{C} PC C5 (in fig. 1.2b) is used during the points-to analysis on \mathcal{D}^{fst} and \mathcal{D}^{snd} .

During proof query discharge, the points-to invariants are encoded as SMT constraints. This allows us to complete the bisimulation proof on the decons-PCFG in fig. 3.6a, and consequently, successfully discharge the proof obligation $\{\phi_{S3:C5}\}(S3 \rightarrow S5 \rightarrow S3, C5 \rightarrow C3)\{1_S \sim \text{clist}_{\text{lm}}^{\text{lnode}}(1_C)\}$ in fig. 3.2b. The points-to analysis is further discussed in section 4.1.

3.6.6 Summary of Type III Proof Discharge Algorithm

Before the start of an equivalence check, a points-to analysis is run on the \mathcal{C} program (IR) once. During equivalence check, to discharge a type III proof obligation $P : \text{LHS} \Rightarrow \text{RHS}$ (expressed first-order logic), we substitute ADT values (in \mathcal{S}) in the RHS with lifted C values (in \mathcal{C}), based on the recursive relations present in the LHS. This is followed by decomposition of RHS and RHS-breaking.

Upon RHS-breaking, we obtain several smaller proof obligations, say $P_i : \text{LHS}_i \Rightarrow \text{RHS}_i$ (for $i = 1 \dots n$). To prove P , we require *all* of these smaller proof obligations P_i to be provable. However, a counterexample to *any* one of these proof obligations would also be a counterexample to the original proof obligation P . Due to decomposition and RHS-breaking, each RHS_i must be a decomposition clause and hence, relate atomic expressions. If RHS_i relate two scalar values, then P_i is a type II proof obligation and discharged using the algorithm summarized in section 3.5.4.

If RHS_i relates two lifted expressions (i.e. a recursive relation), we check if the deconstruction programs of the two ADT values being compared can be proven to be equivalent (assuming LHS_i holds at the correlated entry nodes on the first invocation). Similar to the top-level equivalence check, we attempt to find a bisimulation relation. To improve the precision during bisimilarity check, we transfer points-to invariants of the \mathcal{C} program (at the PC where the proof obligation is being discharged) to the entry of the deconstruction programs. The same points-to analysis is run on the deconstruction programs before the equivalence check begins, (through construction of decons-PCFG) to identify points-to invariants in the deconstruction programs.

If the bisimilarity check succeeds, we return *true* for P ; otherwise, we imprecisely return *false* (without counterexamples).

3.7 Overview of Proof Discharge Algorithm

Figure 3.8 gives the pseudocode of our proof discharge algorithm. The top-level function responsible for discharging Hoare triple proof obligations is *prove()*. It accepts a proof obligation along with the categorization (k) and approximation (d_o and d_u) parameters, and either returns **True**

Algorithm 1: Algorithm for discharging proof obligations containing recursive relations

```

1 Function prove( $\{\phi_s\}(e)\{\phi_d\}, k, d_o, d_u$ )
2    $F \leftarrow \text{lowerWP}(\{\phi_s\}(e)\{\phi_d\})$ 
3   foreach  $\text{LHS} \Rightarrow \text{RHS}_i$  in  $\text{RHSBreak}(F)$  do
4     if  $\text{solve}(\text{LHS}, \text{RHS}_i, k, d_o, d_u) = \text{False}(\Gamma)$  then
5       return  $\text{False}(\Gamma)$ 
6     end
7   end
8   return  $\text{True}$ 
9 end
10 Function solve( $\text{LHS}, \text{RHS}, k, d_o, d_u$ )
11    $(\text{LHS}_k, \text{RHS}_k) \leftarrow \text{decomposeAndUnroll}(\text{LHS}, \text{RHS}, k)$ 
12   switch  $\text{categorize}(\text{LHS}_k, \text{RHS}_k)$  do
13     case Type I do
14       return  $\text{solveSMT}(\text{LHS}_k \Rightarrow \text{RHS}_k)$ 
15     case Type II do
16        $\text{LHS}_o \leftarrow \text{overapproximate}(\text{LHS}, d_o)$ 
17       if  $\text{solveSMT}(\text{LHS}_o \Rightarrow \text{RHS}_k) = \text{True}$  then
18         return  $\text{True}$ 
19       end
20        $\text{LHS}_u \leftarrow \text{underapproximate}(\text{LHS}, d_u)$ 
21       if  $\text{solveSMT}(\text{LHS}_u \Rightarrow \text{RHS}_k) = \text{False}(\Gamma)$  then
22         return  $\text{False}(\Gamma)$ 
23       end
24       return  $\text{False}(\emptyset)$ 
25     case Type III do
26        $\text{RHS}' \leftarrow \text{rewriterRHSUsingLHS}(\text{LHS}, \text{RHS})$ 
27       foreach  $P_i \Rightarrow \text{RHS}_i$  in  $\text{decompose}(\text{RHS}')$  do
28         if  $\text{RHS}_i = l_1 \sim l_2$  then
29            $(\mathcal{D}_1, \mathcal{D}_2) \leftarrow \text{getDeconstructionPrograms}(l_1, l_2)$ 
30           if  $\text{checkEquivalence}(\text{LHS} \wedge P_i, \mathcal{D}_1, \mathcal{D}_2) = \text{False}$  then
31             return  $\text{False}(\emptyset)$ 
32           end
33         else if  $\text{solve}(\text{LHS} \wedge P_i, \text{RHS}_i, k, d_o, d_u) = \text{False}(\Gamma)$  then
34           return  $\text{False}(\Gamma)$ 
35         end
36       end
37       return  $\text{True}$ 
38     end
39   end
40 end

```

Figure 3.8: Pseudocode of the algorithm responsible for discharging proof obligations containing recursive relations

representing a successful proof attempt, or **False**(Γ), where Γ is a set of counterexamples. Recall that our proof discharge algorithm is *sound* and may return **False**(\emptyset) to indicate a failed (proof and counterexample generation) attempt. As discussed in section 2.6, we lower the Hoare triple into a first-order logic formula using weakest-precondition predicate transformer (**lowerWP**()). This is followed by **RHS-breaking** (introduced in section 3.1), which results in multiple smaller proof obligations (**RHSBreak**()). Next, we attempt to prove each of these proof obligations individually through a call to *solve*(). If any one of these queries fail, we immediately stop and return **False** with the counterexamples in Γ — a counterexample to one of the smaller queries is also a counterexample to the original query.

solve() is responsible for discharging these smaller queries. The inputs include **LHS**, **RHS** (representing the proof obligation $P : \text{LHS} \Rightarrow \text{RHS}$); along with the parameters k , d_o and d_u . We start by decomposing and finding the k -unrolled form of P , say $P_E : \text{LHS}_k \Rightarrow \text{RHS}_k$ through **decomposeAndUnroll**(). Next, we categorize P_E into one of the three types (**categorize**()). As discussed in section 3.4, we simply discharge a type I query using SMT solvers (through **solveSMT**()). **solveSMT**() is responsible for (a) translating the input formula (absent of recursive relations) to SMT logic, (b) discharging SMT solver queries, and (c) reconstructing counterexamples from the models returned by the SMT solvers. The steps (a) and (c) are further explored in sections 4.4.7 and 4.4.8 respectively. As summarized in section 3.5.4, for a type II query, we overapproximate **LHS** into **LHS_o** (through **overapproximate**()) and attempt to prove $\text{LHS}_o \Rightarrow \text{RHS}_k$ through **solveSMT**(). In case the proof attempt fails, we underapproximate **LHS** into **LHS_u** (through **underapproximate**()) and attempt to disprove (i.e. generate counterexamples) $\text{LHS}_u \Rightarrow \text{RHS}_k$ through **solveSMT**(). If both attempts fail, we *soundly* return **False** (without counterexamples). Lastly, a type III query P_E is discharged as detailed in section 3.6.6. In brief, we rewrite **RHS** using recursive relations in **LHS** (**rewriteRHSUsingLHS**()) and decompose (**decompose**()) P_E . This results in smaller proof obligations; ones without a recursive relation in its **RHS** are type II queries and discharged through a recursive call to *solve*. For those containing a recursive relation $l_1 \sim l_2$ in their **RHS**, we attempt to show equivalence (through **checkEquivalence**()) between the deconstruction programs of l_1 and l_2 constructed through **getDeconstructionPrograms**(). If any one of these queries fail, we immediately return **False** with the counterexamples (if available). Otherwise, we have successfully proven a type III query

and return True.

Chapter 4

Spec-to-C Equivalence Checker

This chapter presents our automatic equivalence checker S2C. Given a Spec and a C program along with the input-output specification for each function-pair, S2C searches for a proof of equivalence between the CFGs of each Spec and C procedures: \mathcal{S} and \mathcal{C} . Recall that CFGs represent deterministic programs and evidently, the C procedure is determinized during conversion to \mathcal{C} . Hence, S2C checks equivalence between a Spec procedure and the determinized C procedure. A translation validator (such as the Counter tool [27]) can be used to check equivalence between the same determinized C procedure against its generated assembly. By restricting the deterministic choices made by the C compiler to the same choices made during construction of \mathcal{C} , we can effectively establish end-to-end equivalence between a Spec procedure against its assembly. We start with a dataflow formulation of the points-to analysis used as part of S2C on \mathcal{C} as well as deconstruction programs during discharge of type III proof obligations in section 4.1. As stated in section 1.2, S2C is based on three major algorithms: (A1) an algorithm to incrementally construct a product-CFG by correlating program executions across \mathcal{S} and \mathcal{C} , (A2) an algorithm to identify inductive invariants at intermediate PCs in the (partially constructed) product-CFG, and (A3) an algorithm for solving proof obligations generated by (A1) and (A2) algorithms. We describe our counterexample-guided best-first search algorithm for construction of a product-CFG ((A1)) in section 4.2. This is followed by a dataflow formulation of our counterexample-guided invariant inference algorithm ((A2)) in section 4.3. Recall that algorithms (A1) and (A2) are based on prior

Table 4.1: Dataflow Formulation of the Points-to Analysis

Domain	$\Delta^C : (\mathbb{S}^C \cup \mathbb{R}) \rightarrow 2^{\mathbb{R}}$ $\Delta^D : (\mathbb{S}^C \cup \mathbb{R} \cup \mathbb{S}^D) \rightarrow 2^{\mathbb{R}}$
Direction	Forward
Boundary Condition	Δ_n for start node : $\Delta_n^C(t) = \begin{cases} \emptyset & t \in \mathbb{S}^C \\ \mathbb{R} & t \in \mathbb{R} \end{cases} \quad \Delta_n^D(t) = \begin{cases} \Delta_{n_C}^C(t) & t \in (\mathbb{S}^C \cup \mathbb{R}) \\ \emptyset & t \in \mathbb{S}^D \end{cases}$
Initialization to \top	Δ_n for non-start nodes : $\Delta_n(t) = \emptyset \quad t \in \text{Domain}(\Delta_n)$
Transfer function across edge $e = (s \rightarrow d)$	$\Delta_d = f_e(\Delta_s)$ (described in section 4.1)
Meet operator \otimes	$\Delta_n \leftarrow \Delta_n^1 \otimes \Delta_n^2$ $\Delta_n(t) = \Delta_n^1(t) \cup \Delta_n^2(t) \quad t \in \text{Domain}(\Delta_n)$

work on equivalence checking between unoptimized IR (aka determinized C) and assembly [48]. We only summarize these procedures in addition to the primary modifications made in the context of Spec-C equivalence. The previous chapter walked through our proof discharge algorithm (A3) using examples, ending with the pseudocode of proof discharge algorithm (fig. 3.8). In this chapter, we present pseudocode for multiple subprocedures utilized by our proof discharge algorithm in section 4.4. Additionally, we describe the process of encoding queries in SMT logic as well as the recovery of counterexamples from models returned by SMT solvers. We finish with a new representation of expressions, which allows us to simplify a number of steps performed by the proof discharge algorithm.

4.1 Points-to Analysis

Recall that in section 3.6.3, we needed to reason about aliasing to successfully discharge a type III proof obligation. These aliasing relationships are introduced in section 3.6.4 and subsequently used in section 3.6.5 to successfully discharge a type III proof obligation. An interprocedural, flow-sensitive, untyped points-to analysis is used to identify these relationships in \mathcal{C} as well as each

deconstruction program \mathcal{D} . Table 4.1 presents the dataflow formulation of our points-to analysis. We start by identifying the set \mathbb{R} of all region labels representing mutually non-overlapping regions of the \mathcal{C} memory \mathfrak{m} . For each call to `malloc()` at PC A of \mathcal{C} 's IR, we add A_1 and A_{2+} to \mathbb{R} . Recall that A_1 represents the region of memory returned by the *most recent* execution of A . A_{2+} represents the region of memory returned by older (i.e. all but most recent) executions of A . $\mathbb{R} = \bigcup_A \{A_1, A_{2+}\} \cup \{\mathcal{H}\}$, where \mathcal{H} is the region of memory \mathfrak{m} not covered by the labels associated with allocation sites.

Let $\mathbb{S}^{\mathcal{C}}$ be the set of all scalar pseudo-registers in \mathcal{C} 's IR. We use a forward dataflow analysis to identify a may-point-to function $\Delta^{\mathcal{C}} : (\mathbb{S}^{\mathcal{C}} \cup \mathbb{R}) \mapsto 2^{\mathbb{R}}$ at each program point in \mathcal{C} 's IR. For a deconstruction program \mathcal{D} , we are also interested in finding the may-point-to function for all scalar pseudo-registers in \mathcal{D} 's IR, say $\mathbb{S}^{\mathcal{D}}$. Thus, the domain of the may-point-to function for \mathcal{D} ($\Delta^{\mathcal{D}}$) contains $\mathbb{S}^{\mathcal{D}}$ in addition to the domain of $\Delta^{\mathcal{C}}$ i.e. $\Delta^{\mathcal{D}} : (\mathbb{S}^{\mathcal{C}} \cup \mathbb{R} \cup \mathbb{S}^{\mathcal{D}}) \mapsto 2^{\mathbb{R}}$. The ' \rightsquigarrow ' operator introduced in section 3.6.4 is called the *element-wise* may-point-to function and is related to the may-point-to function Δ as follows: $p \rightsquigarrow S \Leftrightarrow \Delta(p) \subseteq S$.

The meet operator is element-wise set-union e.g., $p \rightsquigarrow S_1$ and $p \rightsquigarrow S_2$ combines into $p \rightsquigarrow S_1 \cup S_2$. Evidently, the \top value is the constant function that returns \emptyset . At entry of \mathcal{C} , we conservatively assume that all memory regions may point to each other. However, at entry of a deconstruction program \mathcal{D} , created during a proof obligation at product-CFG node $(n_S : n_C)$, we use \mathcal{C} 's precomputed may-point-to function at n_C ($\Delta_{n_C}^{\mathcal{C}}$) to initialize the points-to relationships for all state elements in \mathcal{C} 's IR (i.e. $\mathbb{S}^{\mathcal{C}} \cup \mathbb{R}$). This is a crucial step for proving equality of \mathcal{C} values under different memory states as seen in section 3.6.5.

Next, we discuss the transfer function f_e for our points-to analysis. For an IR instruction $x := c$, for constant c , the transfer function updates $\Delta(x) := \emptyset$. For instruction $x := y \text{ op } z$ (for some arithmetic or logical operator `op`), we update $\Delta(x) := \Delta(y) \cup \Delta(z)$. For a load instruction $x := \mathfrak{m}[y]_{\text{T}}$, we update $\Delta(x) := \bigcup_{t \in \Delta(y)} \Delta(t)$. For a store instruction $\mathfrak{m} := \mathfrak{m}[x \leftarrow y]_{\text{T}}$, for all $t \in \Delta(x)$, we update $\Delta(t) := \Delta(t) \cup \Delta(y)$. For a malloc instruction $x := \text{malloc}_A()$ (where A represents the allocation site), we perform the following steps (in order):

1. Convert all existing occurrences of A_1 to A_{2+} , i.e., for all $t \in (\mathbb{S}^{\mathcal{C}} \cup \mathbb{R})$, if $A_1 \in \Delta(t)$, then

update $\Delta(t) := (\Delta(t) \setminus \{A_1\}) \cup \{A_{2+}\}$.

2. Update $\Delta(x) := \{A_1\}$.
3. Update $\Delta(A_{2+}) := \Delta(A_{2+}) \cup \Delta(A_1)$.
4. Update $\Delta(A_1) := \emptyset$.

For function calls, a *supergraph* is created by adding control flow edges from the call-site to the procedure head (copying actual arguments to the formal arguments) and from the procedure exit to the program point just after the call-site (copying returned value to the variable assigned at the callsite), e.g., in fig. 3.5b, the dashed edges represent supergraph edges.

The allocation-site abstraction (with a bounded-depth call stack) is known to be effective at disambiguating memory regions belonging to different data structures [29, 16, 11]. In our work, we also need to reason about non-aliasing of the most-recently allocated object (through a `malloc` call) and the previously-allocated objects (as in the `List` construction example). The coarse-grained $\{1, 2+\}$ categorization of allocation recency is effective for such disambiguation.

4.2 Counterexample-guided Product-CFG Construction

S2C constructs a product-CFG incrementally to search for a bisimulation relation between the Spec and C CFGs : \mathcal{S} and \mathcal{C} . Multiple candidate product-CFGs are partially constructed during this search; the search completes when one of these candidates yield an equivalence proof. Recall that the incremental product-CFG construction algorithm is based on prior work [48] and we simply summarize the relevant components in the context of Spec-C equivalence.

Anchor nodes are identified in \mathcal{S} and \mathcal{C} , and represents the CFG nodes (i.e. IR PCs) being considered for correlation. The algorithm ensures that every cycle in both \mathcal{S} and \mathcal{C} contains at least one anchor node. The start and exit nodes are always anchor nodes. Also, for every function call, the nodes just before and after its callsite are considered anchor nodes. For example, in fig. 1.3b, C4 and C5 are anchor nodes around the call to `malloc`. A valid selection of anchor

Algorithm 2: Best-first search algorithm for incremental construction of a product-CFG between \mathcal{S} and \mathcal{C}

```

1 Function bestFirstSearch( $\mathcal{S}, \mathcal{C}, \mu$ )
2    $\pi_{init} \leftarrow \text{createInitProductCFG}(\mathcal{S}, \mathcal{C});$ 
3    $Q \leftarrow \{\pi_{init}\};$ 
4   while  $Q$  is not empty do
5      $\pi_{cur} \leftarrow \text{extractMostPromising}(Q);$ 
6      $\text{InferInvariantsAndCounterexamples}(\pi_{cur});$ 
7     if  $\text{getPathsetToCorrelateInC}(\mathcal{C}, \pi_{cur}) = \text{Found}(\xi_C)$  then
8       foreach  $\xi_S$  in  $\text{enumeratePathsetsInS}(\mathcal{S}, \xi_C, \mu)$  do
9          $\pi_{next} \leftarrow \text{extendProductCFG}(\pi_{cur}, \xi_S, \xi_C);$ 
10         $Q \leftarrow Q \cup \{\pi_{next}\};$ 
11      end
12    else if  $\text{productCFGRepresentsBisim}(\pi_{cur})$  then
13      return  $\text{Found}(\pi_{cur});$ 
14    end
15  end
16  return  $\text{NotFound};$ 
17 end

```

Figure 4.1: Pseudocode of the best-first search algorithm responsible for incremental construction of a product-CFG between \mathcal{S} and \mathcal{C}

nodes for the CFGs in figs. 1.3a and 1.3b are: $\{S0, S3, SE\}$ and $\{C0, C3, C4, C5, CE\}$ respectively. For each anchor node in \mathcal{C} , our search algorithm searches for a correlated anchor node in \mathcal{S} — if a (partially constructed) product-CFG π contains a product-CFG node $(n_S : n_C)$, then π correlates node n_C in \mathcal{C} with node n_S in \mathcal{S} . The search procedure begins with a single partially-constructed product-CFG π_{init} . π_{init} contains exactly one node $(S0 : C0)$ that encodes the correlation of the entry nodes (i.e. $S0$ and $C0$) of \mathcal{S} and \mathcal{C} .

4.2.1 Correlating Pathsets

At each step of the incremental construction process, a node $(n_S : n_C)$ is chosen in a product-CFG π and a 1-pathset ξ_C in \mathcal{C} starting at n_C (and ending at an anchor node in \mathcal{C}) is selected (`getPathsetToCorrelateInC()`). Then, we enumerate potentially correlated μ -pathsets in \mathcal{S} for the pathset ξ_C in \mathcal{C} (`enumeratePathsetsInS()`). A pathset ξ is essentially a set of paths with the following additional properties: (a) all paths $\rho \in \xi$ begin at the same node and terminate at the same node, (b) all paths $\rho \in \xi$ are mutually-exclusive i.e. *at most* one of pathcond_ρ can be true. A μ -pathset ξ is a pathset where each path $\rho \in \xi$ contains *at most* μ occurrences of any node as the destination (to-PC) of an edge in ρ . μ -pathsets are based on earlier work on the Counter tool [27]¹ and helps improve the completeness of the bisimulation search by attempting to correlate a set of \mathcal{C} paths with a set of \mathcal{S} paths, where individual paths of \mathcal{C} and \mathcal{S} may be uncorrelated. An example of such a scenario is depicted in section 5.1.1. Section 4.4.1 briefly discusses the techniques of handling Hoare triples (i.e. proof obligations) involving pathsets.

Revisiting our incremental construction process, recall that we choose a 1-pathset ξ_C in \mathcal{C} and enumerate potentially correlated μ -pathsets ξ_S in \mathcal{S} . Hence, μ represents the maximum number of iterations of a loop (in \mathcal{S}), that may be correlated with a pathset in \mathcal{C} consisting of acyclic paths. μ is a fixed parameter of the S2C algorithm and is called the *unroll factor*. For example, during construction of the product-CFG shown in fig. 1.4a, say we select the product-CFG node $(S3 : C3)$. We choose the \mathcal{C} path(set) $C3 \rightarrow C4$ and enumerate its potential correlations (i.e. μ -pathsets in \mathcal{S} starting at $S3$): ϵ , $S3 \rightarrow S5 \rightarrow S3$, ..., $S3 \rightarrow (S5 \rightarrow S3)^\mu$. Importantly, for pathsets ξ_S (in \mathcal{S}) and ξ_C (in

¹Counter tool [27] considers (μ, δ) -unrolled pathsets in its correlation search algorithm. Our μ -pathsets are a special case of (μ, δ) -unrolled pathsets where $\delta = \mu$.

\mathcal{C}) to be considered for correlation, they must originate and terminate at anchor nodes, i.e. the path $\mathbf{S3} \rightarrow \mathbf{S5}$ is skipped during enumeration. Moreover, the pathset ξ_C may only contain anchor nodes as its source and destination. Hence, the path $\mathbf{C3} \rightarrow \mathbf{C4} \rightarrow \mathbf{C5}$ is not considered for ξ_C , instead we attempt to correlate the subpaths $\mathbf{C3} \rightarrow \mathbf{C4}$ and $\mathbf{C4} \rightarrow \mathbf{C5}$ individually.

For each enumerated correlation possibility (ξ_S, ξ_C) , a separate product-CFG π' is created (by cloning π) and a new product-CFG edge $e = (\xi_S, \xi_C)$ is added to π' (`extendProductCFG()`). The head of the product-CFG edge e is the (potentially newly added) product-CFG node representing the correlation of the end-points of pathsets ξ_S and ξ_C . For example, the node $(\mathbf{S3}:\mathbf{C4})$ is added to the product-CFG if it correlates pathsets ϵ and $\mathbf{C3} \rightarrow \mathbf{C4}$ starting at $(\mathbf{S3}:\mathbf{C3})$. Recall that, we consider ϵ as a candidate for ξ_S , but not ξ_C . The algorithm ensures that no cycle in \mathcal{C} is correlated with ϵ in \mathcal{S} (to preserve divergence discussed in section 2.4). For each node n in a product-CFG π , we maintain a small number of concrete machine state pairs (of \mathcal{S} and \mathcal{C}). The concrete machine state pairs at s are obtained as counterexamples to unsuccessful proof obligations $\{\phi_s\}(s \rightarrow d)\{\phi_d\}$ (for some edge $s \rightarrow d$ and node d in π). Thus, by construction, these counterexamples represent concrete state pairs that may potentially occur at n during the lockstep execution encoded by π .

4.2.2 Best-First Ranking of Partial Product-CFGs

To evaluate the promise of a possible correlation (ξ_S, ξ_C) starting at node n in product-CFG π , we examine the execution behaviour of the counterexamples at n on the product-CFG edge $e = (s \rightarrow d) = (\xi_S, \xi_C)$. If the counterexamples ensure that the machine states remain related at d , then that candidate correlation is ranked higher. This ranking criterion is based on prior work [27]. A best-first search (BFS) procedure based on this ranking criterion is used to incrementally construct a product-CFG (starting from π_{init}). For each intermediate candidate product-CFG π generated during this search procedure, an automatic invariant inference procedure (discussed next in section 4.3) is used to identify invariants at all the nodes in π (`InferInvariantsAndCounterexamples()`). The counterexamples obtained from the proof obligations generated by this invariant inference procedure are added to the respective nodes in π ; these counterexamples help rank future correlations starting at those nodes.

If after invariant inference, we realize that an intermediate candidate product-CFG π_1 is not promising enough, we backtrack and choose another candidate product-CFG π_2 and explore the potential correlations that can be added to π_2 (`extractMostPromising()`). Thus, a product-CFG is constructed one edge at a time. If at any stage, a product-CFG π satisfies the well-formedness conditions (introduced in section 2.4.1) and invariants ensure equal observables (i.e. *Post* holds at correlated exit nodes), we have successfully shown equivalence (`productCFGRepresentsBisim()`). This counterexample-guided BFS procedure is similar to the one described in prior work on the Counter algorithm [27].

4.2.3 Correlation in the Presence of Function Calls

Recall that \mathcal{S} and \mathcal{C} may make function calls (including self calls), e.g., C memory allocation, recursive traversal of a tree data structure. Recall that the nodes just before and after a function call are always considered anchor nodes. Calls to memory allocation functions in \mathcal{C} (i.e. `malloc`) are handled by correlating the function call edge with the empty path (ϵ) in \mathcal{S} . For example, in the product-CFG shown in fig. 1.4a, the `malloc` edge $C4 \rightarrow C5$ in \mathcal{C} is correlated with ϵ in \mathcal{S} .

For all other calls, our correlation algorithm (in section 4.2) ensures that the anchor nodes around such a callsite are correlated one-to-one across both procedures. For example, let there be a call to procedure δ in \mathcal{S} at PC n_S , i.e. n_S is the call-site. Let us denote the program point just after this call-site as n'_S . Let \mathbf{args}_{n_S} represent the values of the actual arguments of this function call (at n_S). Let $\mathbf{ret}_{n'_S}$ represent the value returned by this function call (at n'_S). Similarly, for a procedure call δ in \mathcal{C} , let n_C , n'_C , \mathbf{args}_{n_C} and $\mathbf{ret}_{n'_C}$ represent the function call-site, program point just after the call-site, the values of the actual arguments and the value returned respectively. Our algorithm ensures that the only correlation possible in a product-CFG π for these program points are $(n_S : n_C)$ and $(n'_S : n'_C)$.

We utilize the user-supplied input-output specification for δ , say $(Pre_\delta, Post_\delta)$, to obtain the desired invariants at nodes $(n_S : n_C)$ and $(n'_S : n'_C)$ in the product-CFG. A successful proof must *ensure* that $Pre_\delta(\mathbf{args}_{n_S}, \mathbf{args}_{n_C})$ holds at $(n_S : n_C)$. Further, the proof can *assume* that $Post_\delta(\mathbf{ret}_{n'_S}, \mathbf{ret}_{n'_C})$ holds at $(n'_S : n'_C)$. Note that \mathbf{args}_{n_C} and $\mathbf{ret}_{n'_C}$ includes the \mathcal{C} memory

Table 4.2: Dataflow Formulation of the Invariant Inference Algorithm

Domain	$\left\{ (\phi_n, \Gamma_n) \mid \begin{array}{l} \phi_n \text{ is a conjunction of predicates drawn from} \\ \mathbb{G} \text{ (in fig. 4.2b), } \Gamma_n \text{ is a set of counterexamples} \end{array} \right\}$
Direction	Forward
Boundary Condition	(ϕ_n, Γ_n) for start node : $\phi_n \leftarrow Pre, \Gamma_n \leftarrow \emptyset$
Initialization to \top	(ϕ_n, Γ_n) for non-start nodes : $\phi_n \leftarrow \mathbf{false}, \Gamma_n \leftarrow \emptyset$
Transfer function across edge $e = (s \rightarrow d)$	$(\phi_d, \Gamma_d) = f_e(\phi_s, \Gamma_s)$ (shown in fig. 4.2a)
Meet operator \otimes	$(\phi_n, \Gamma_n) \leftarrow (\phi_n^1, \Gamma_n^1) \otimes (\phi_n^2, \Gamma_n^2)$ $\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2, \phi_n \leftarrow \mathbf{strongestInvCover}(\Gamma_n)$

states \mathfrak{m}_{n_C} (at n_C) and $\mathfrak{m}_{n'_C}$ (at n'_C) respectively. Thus, for function calls, we inductively prove the precondition (on the arguments) at $(n_S : n_C)$ and assume the postcondition (on the returned values) at $(n'_S : n'_C)$.

4.3 Invariant Inference and Counterexample Generation

We formulate our counterexample-guided invariant inference algorithm as a forward dataflow analysis as shown in table 4.2. The invariant inference procedure is responsible for inferring invariants ϕ_n at each intermediate node n of a (partially constructed) product-CFG, while also generating a set of counterexamples Γ_n that represents the potential concrete machine states at n . Recall that the invariant inference algorithm is based on prior work [48] and we simply summarize the relevant components in the context of Spec-C equivalence.

Given the invariants and counterexamples at node s : (ϕ_s, Γ_s) , the transfer function initializes the new candidate set of counterexamples at d (Γ_d^{can}) with the current set of counterexamples at d (Γ_d) *union*-ed with the counterexamples obtained by executing Γ_s on edge e (through \mathbf{exec}_e). The candidate invariant at d (ϕ_d^{can}) is computed as the strongest cover of Γ_d^{can} ($\mathbf{strongestInvCover}()$).

```

1 Function  $f_e(\phi_s, \Gamma_s)$ 
2    $\Gamma_d^{can} \leftarrow \Gamma_d \cup \text{exec}_e(\Gamma_s)$ 
3    $\phi_d^{can} \leftarrow \text{strongestInvCover}(\Gamma_d^{can})$ 
4   while  $\text{prove}(\{\phi_s\}(e)\{\phi_d^{can}\}) = \text{False}(\gamma_s)$  do
5      $\gamma_d \leftarrow \text{exec}_e(\gamma_s)$ 
6      $\Gamma_d^{can} \leftarrow \Gamma_d^{can} \cup \gamma_d$ 
7      $\phi_d^{can} \leftarrow \text{strongestInvCover}(\Gamma_d^{can})$ 
8   end
9   return  $(\phi_d^{can}, \Gamma_d^{can})$ 
10 end

```

(a) Transfer function f_e across edge $e = (s \rightarrow d)$.

$$\begin{aligned} \text{Inv} &\rightarrow \sum_i c^i v^i = c \mid v^1 \odot v^2 \\ &\mid \alpha_S \sim \text{Clift}_{\mathfrak{m}}^T(v_C \dots) \end{aligned}$$

(b) Predicate grammar \mathbb{G} for constructing invariants. v represents a bitvector variable in either \mathcal{S} or \mathcal{C} . c represents a bitvector constant. $\odot \in \{<, \leq\}$. α_S represents an ADT variable in \mathcal{S} . v_C represents a bitvector variable in \mathcal{C} . \mathfrak{m} represents the current \mathcal{C} memory state.

Figure 4.2: Transfer function f_e and predicate grammar \mathbb{G} for invariant inference dataflow analysis in table 4.2. Given invariants ϕ_s and counterexamples Γ_s at node s , f_e returns the updated invariants ϕ_d and counterexamples Γ_d at node d . **strongestInvCover**(Γ) computes the strongest invariant cover for counterexamples Γ . **exec** _{e} (Γ) (concretely) executes counterexamples Γ over edge e . *prove*(P) (in fig. 3.8) discharges a proof obligation P , and returns either **True** or **False**(Γ).

At each step, the transfer function attempts to prove $\{\phi_s\}(e)\{\phi_d^{can}\}$ (through a call to *prove()*). If the proof succeeds (*prove()* returns **True**), the candidate invariant ϕ_d^{can} is returned along with the counterexamples Γ_d^{can} learned so far. Otherwise, *prove()* returns **False**(γ_s). The candidate invariant ϕ_d^{can} is weakened using the counterexamples obtained (i.e. γ_s) and the proof attempt is repeated.

The candidate invariants are drawn from the predicate grammar \mathbb{G} shown in fig. 4.2b. In addition to affine and inequality relations between bitvectors in \mathcal{S} and \mathcal{C} , \mathbb{G} supports recursive relations between an ADT variable in \mathcal{S} and a lifted expression in \mathcal{C} . The candidate lifting constructors of the form $\text{Clift}_{\mathfrak{m}}^T$ (where \mathfrak{m} is the current memory state in \mathcal{C}) are derived from the lifting constructors present in the precondition *Pre* and the postcondition *Post*, as supplied by the user. More sophisticated strategies for inference of new lifting constructors is left as future work.

strongestInvCover() for affine relations involve identifying the basis vectors of the kernel of the matrix formed by the counterexamples in the bitvector domain [37, 18]. For inequality relations, **strongestInvCover**(Γ) returns *true* (i.e. the weakest invariant) iff any counterexample in Γ evaluates the relation to false — this effectively simulates the Houdini approach [25]. Similarly, in case of a recursive relation $l_1 \sim l_2$, **strongestInvCover**(Γ) returns *true* iff any counterexample in Γ evaluates its η -depth over-approximation $l_1 \sim_\eta l_2$ to false (effectively falsifying a weaker condition and thus $l_1 \sim l_2$ itself), where η is a fixed parameter of the algorithm.

4.4 More on Proof Discharge Algorithm

4.4.1 Handling Proof Obligations On Pathsets

Recall that our correlation algorithm attempts to correlate pathsets (instead of paths) between \mathcal{S} and \mathcal{C} . Evidently, each edge of a (possibly partial) product-CFG π is associated with a pair of pathsets (ξ_S, ξ_C) . Proof obligations originating across a product-CFG edge $e[s \rightarrow d] = (\xi_S, \xi_C)$ are of the form $\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\}$. A Hoare triple of the above form can be broken down into a conjunction of Hoare triples involving purely paths as follows:

$$\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\} \Leftrightarrow \bigwedge_{\substack{\rho_S \in \xi_S \\ \rho_C \in \xi_C}} \{\phi_s\}(\rho_S, \rho_C)\{\phi_d\} \quad (4.1)$$

Recall that our proof discharge algorithm requires that proof obligations satisfy the conjunctive recursive relation property (defined in section 3.1). If the original proof obligation $\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\}$ satisfies this property, so does each of the smaller proof obligation $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$. Hence, the proof discharge algorithm presented in chapter 3 is capable of handling these smaller proof obligations and by eq. (4.1), also the original proof obligations. If $|\xi|$ represents the number of paths in a pathset ξ , the proof obligation $\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\}$ results in $|\xi_S| \times |\xi_C|$ smaller proof obligations. In practice, the number of paths in \mathcal{S} that are correlated with a path in \mathcal{C} is quite low and consequently, most of these proof obligations usually end up with a **false** LHS when lowered to first-order logic (through eq. (2.4)), due to the presence of pathcond_{ρ_S} and pathcond_{ρ_C} . Our proof discharge procedure begins with an attempt to disprove LHS (overapproximated in case of recursive relations), which trivially resolves the smaller proof obligations involving uncorrelated paths to **true**.

4.4.2 Canonicalization Procedure

Algorithm 3 shows the pseudocode for the canonicalization procedure. $\text{Canonicalize}(e)$ is responsible for converting an expression e to its canonical form \hat{e} (introduced in section 3.2). Recall that a pseudo-variable is an expression of the form $v.a_1.a_2...a_n$, where v is a variable. Also recall that, an expression e is canonical iff each *accessor* and *sum-is* expression operate on a pseudo-variable. An ADT expression with a data constructor, a lifting constructor or the **if-then-else**-operator at its top-level, is called a *foldable* expression. $\text{Canonicalize}(e)$ iteratively folds each *accessor* and *sum-is* subexpressions of e that operate on a foldable argument. Thus, $\text{Canonicalize}(e)$ returns an expression where none of the *accessor* or *sum-is* subexpressions is foldable. This condition entails the requirements of the canonical form. For example, $a + \text{LCons}(b, l).\text{tail.val}$ and $\text{Clist}_m^{\text{inode}}(p)$ is **LNil** canonicalizes to $a + l.\text{val}$ and $(p = 0)$ respectively.

Algorithm 3: Pseudocode for Canonicalization Procedure

```

Function Canonicalize(e)
   $\hat{e} \leftarrow e$ ;
  while e contains  $e' = e_1.a^i$  where e1 is foldable do
    if  $e_1 = V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n)$  then
      |  $\hat{e} \leftarrow \{e' \mapsto e_1^i\}\hat{e}$ ;
    else if  $e_1 = \text{if } c_1 \text{ then } V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n) \text{ else } e_1^{e1}$  then
      | if  $V_1^{(n)}$  contains  $a^i$  then  $\hat{e} \leftarrow \{e' \mapsto e_1^i\}\hat{e}$ ;
      | else  $\hat{e} \leftarrow \{e' \mapsto e_1^{e1}.a^i\}\hat{e}$ ;
    else  $e_1 = \text{Clift}_m^T(e^1, e^2, \dots, e^n)$ 
      |  $\hat{e} \leftarrow \{e' \mapsto \text{rewrite}(e_1).a^i\}\hat{e}$ ;
    end
  end
  while e contains  $e' = e_1$  is  $V_2^{(m)}$  where e1 is foldable do
    if  $e_1 = V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n)$  then
      | if  $V_1^{(n)} = V_2^{(m)}$  then  $\hat{e} \leftarrow \{e' \mapsto \text{true}\}\hat{e}$ ;
      | else  $\hat{e} \leftarrow \{e' \mapsto \text{false}\}\hat{e}$ ;
    else if  $e_1 = \text{if } c_1 \text{ then } V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n) \text{ else } e_1^{e1}$  then
      | if  $V_1^{(n)} = V_2^{(m)}$  then  $\hat{e} \leftarrow \{e' \mapsto c_1\}\hat{e}$ ;
      | else  $\hat{e} \leftarrow \{e' \mapsto \neg c_1 \wedge (e_1^{e1} \text{ is } V_2^{(m)})\}\hat{e}$ ;
    else  $e_1 = \text{Clift}_m^T(e^1, e^2, \dots, e^n)$ 
      |  $\hat{e} \leftarrow \{e' \mapsto \text{rewrite}(e_1).a^i\}\hat{e}$ ;
    end
  end
  return  $\hat{e}$ ;
end

```

Algorithm 4: Pseudocode for Unification Procedure

```

Function  $\theta(p_1, e_1, p_2, e_2)$ 
  if  $e_1$  is atomic then
    | return  $\text{Succ}(\{\langle p_1, e_1, p_2, e_2 \rangle\})$ ;
  else if  $e_2$  is atomic then
    | return  $\text{Succ}(\{\langle p_2, e_2, p_1, e_1 \rangle\})$ ;
  else if  $e_1 = V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n)$  and  $e_2 = V_2^{(m)}(e_2^1, e_2^2, \dots, e_2^m)$  then
    | if  $V_1^{(n)} \neq V_2^{(m)}$  then
    |   | return Fail;
    | end
    | return  $\sqcup_{i \in [1, n]} \theta(p_1, e_1^i, p_2, e_2^i)$ ;
  else if  $e_1 = V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n)$  and  $e_2 = \text{if } c_2 \text{ then } e_2^{\text{th}} \text{ else } e_2^{\text{el}}$  then
    |  $R^{\text{th}} \leftarrow \theta(p_1, \text{true}, p_2, c_2) \sqcup \theta(p_1, e_1, p_2 \triangle c_2, e_2^{\text{th}})$ ;
    | if  $R^{\text{th}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    |  $R^{\text{el}} \leftarrow \theta(p_1, \text{true}, p_2, \neg c_2) \sqcup \theta(p_1, e_1, p_2 \triangle \neg c_2, e_2^{\text{el}})$ ;
    | if  $R^{\text{el}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    | return Fail;
  else if  $e_1 = \text{if } c_1 \text{ then } e_1^{\text{th}} \text{ else } e_1^{\text{el}}$  and  $e_2 = V_2^{(m)}(e_2^1, e_2^2, \dots, e_2^m)$  then
    |  $R^{\text{th}} \leftarrow \theta(p_1, c_1, p_2, \text{true}) \sqcup \theta(p_1 \triangle c_1, e_1^{\text{th}}, p_2, e_2)$ ;
    | if  $R^{\text{th}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    |  $R^{\text{el}} \leftarrow \theta(p_1, \neg c_1, p_2, \text{true}) \sqcup \theta(p_1 \triangle \neg c_1, e_1^{\text{el}}, p_2, e_2)$ ;
    | if  $R^{\text{el}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    | return Fail;
  else  $e_1 = \text{if } c_1 \text{ then } e_1^{\text{th}} \text{ else } e_1^{\text{el}}$  and  $e_2 = \text{if } c_2 \text{ then } e_2^{\text{th}} \text{ else } e_2^{\text{el}}$ 
    |  $R_1 \leftarrow \theta(p_1, c_1, p_2, c_2)$ ;
    |  $R_2 \leftarrow \theta(p_1 \triangle c_1, e_1^{\text{th}}, p_2 \triangle c_2, e_2^{\text{th}})$ ;
    |  $R_3 \leftarrow \theta(p_1 \triangle \neg c_1, e_1^{\text{el}}, p_2 \triangle \neg c_2, e_2^{\text{el}})$ ;
    | return  $R_1 \sqcup R_2 \sqcup R_3$ ;
  end
end

```

4.4.3 Unification Procedure

Algorithm 4 shows the pseudocode for the unification algorithm introduced in section 3.2. $\theta(p_1, e_1, p_2, e_2)$ is responsible for unifying expressions e_1 and e_2 under the expression path conditions p_1 and p_2 respectively. θ either fails to unify with the **Fail** output, or it successfully returns $\text{Succ}(S)$, where S is the set of correlation tuples that relate (a) either two atomic expressions, or (b) an atom with a non-atomic expression. $\theta(p_1, e_1, p_2, e_2)$ terminates when one of e_1 and e_2 is an atomic expression. In case both e_1 and e_2 contains a data constructor at their top-level, θ attempts to recursively unify the data constructors and their corresponding children. If exactly one of e_1

Algorithm 5: Pseudocode for Iterative Unification and Rewriting Procedure

```

Function  $\Theta(p_a, e_a, p_b, e_b)$ 
   $R \leftarrow \emptyset$ ;
   $S \leftarrow \theta(p_a, e_a, p_b, e_b)$ ;
  if  $S = \text{Fail}$  then return Fail;
  foreach  $\langle p_1, a_1, p_2, e_2 \rangle$  in  $S$  do
    if  $e_2$  is atomic then
       $R \leftarrow R \cup \{\langle p_1, a_1, p_2, e_2 \rangle\}$ ;
    else
       $e_1 \leftarrow \text{rewrite}(a_1)$ ;
       $R_1 \leftarrow \Theta(p_1, e_1, p_2, e_2)$ ;
      if  $R_1 = \text{Fail}$  then return Fail;
       $R \leftarrow R \cup R_1$ ;
    end
  end
  return  $\text{Succ}(R)$ ;
end

```

and e_2 is a if-then-else expression, θ attempts to unify both branches of if-then-else (along with the path conditions) with the other expression and return whichever succeeds. If both e_1 and e_2 are if-then-else expressions, θ attempts to recursively unify their children. θ uses the \sqcup -operator to combine the results of successive self-calls. $A \sqcup B$ is equal to $\text{Succ}(S_1 \cup S_2)$ if $A = \text{Succ}(S_1)$ and $B = \text{Succ}(S_2)$; otherwise (if one of A and B is **Fail**), $A \sqcup B = \text{Fail}$. Additionally, for a if-then-else expression with if condition c , c is well-formed under the expression path condition. Hence, when conjuncting c to the expression path condition, we use an ‘ordered and’ operator \triangle ; $e_1 \triangle e_2$ is equivalent to $e_1 \wedge (e_1 \rightarrow e_2)$.

4.4.4 Iterative Unification and Rewriting Procedure

Algorithm 5 shows the pseudocode for the iterative unification and rewriting procedure introduced in section 3.2. $\Theta(p_a, e_a, p_b, e_b)$ is responsible for unifying expressions e_a and e_b under the expression path conditions p_a and p_b respectively. Θ either fails to unify with the **Fail** output, or it successfully returns $\text{Succ}(S)$, where S is the set of correlation tuples that relate *only* atomic expressions. Θ attempts to iteratively (a) unify the expressions (through a call to the unification procedure θ in section 4.4), and (b) perform rewriting (of atom a_1 for those correlation tuples $\langle p_1, a_1, p_2, e_2 \rangle$ where e_2 is non-atomic), followed by a recursive call to Θ . For example, the unifica-

Algorithm 6: Pseudocode of Decomposition Procedure for Recursive Relations

```

Function Decompose( $l_1, l_2$ )
   $ret \leftarrow true$ ;
   $\hat{l}_1 \leftarrow Canonicalize(l_1)$ ;
   $\hat{l}_2 \leftarrow Canonicalize(l_2)$ ;
   $R \leftarrow \Theta(true, \hat{l}_1, true, \hat{l}_2)$ ;
  if  $R = \text{Fail}$  then return false;
  foreach  $\langle p_1, a_1, p_2, a_2 \rangle$  in  $R$  do
    if  $a_1$  is scalar then
       $ret \leftarrow ret \wedge ((p_1 \wedge p_2) \rightarrow (a_1 = a_2))$ ;
    else
       $ret \leftarrow ret \wedge ((p_1 \wedge p_2) \rightarrow (a_1 \sim a_2))$ ;
    end
  end
  return  $ret$ ;
end

```

tion of l_1 and $LCons(42, \text{Clist}_m^{\text{lnode}}(l_2))$ yields the correlation tuples: $\langle true, l_1 \text{ is } LCons, true, true \rangle$, $\langle l_1 \text{ is } LCons, l_1.val, true, 42 \rangle$ and $\langle l_1 \text{ is } LCons, l_1.tail, true, \text{Clist}_m^{\text{lnode}}(l_2) \rangle$.

4.4.5 Decomposition Procedure for Recursive Relations

Algorithm 6 shows the pseudocode for the decomposition algorithm defined in section 3.2. *Decompose*(l_1, l_2) is responsible for computing the decomposition of the recursive relation $l_1 \sim l_2$. Recall that, decomposition of a recursive relation $l_1 \sim l_2$ requires the unification of (canonicalized) l_1 and l_2 through the top-level invocation of $\Theta(true, l_2, true, l_2)$. If the n correlation tuples obtained after a successful unification are $\langle p_1^i, a_1^i, p_2^i, a_2^i \rangle$ (for $i = 1 \dots n$), then the decomposition of $l_1 \sim l_2$ is defined by eq. (3.2). If the unification fails (with a **Fail** output), the decomposition is defined to be **false**. For example, recall that the unification of l_1 and $LCons(42, \text{Clist}_m^{\text{lnode}}(l_2))$ yields the correlation tuples: $\langle true, l_1 \text{ is } LCons, true, true \rangle$, $\langle l_1 \text{ is } LCons, l_1.val, true, 42 \rangle$ and $\langle l_1 \text{ is } LCons, l_1.tail, true, \text{Clist}_m^{\text{lnode}}(l_2) \rangle$. Consequently, $l_1 \sim LCons(42, \text{Clist}_m^{\text{lnode}}(l_2))$ decomposes into the conjunctive predicate: $(l_1 \text{ is } LCons) \wedge (l_1 \text{ is } LCons \rightarrow l_1.val = 42) \wedge (l_1 \text{ is } LCons \rightarrow l_1.next \sim \text{Clist}_m^{\text{lnode}}(l_2))$.

4.4.6 Reduction Procedures for Approximate Recursive Relations

Recall that type II proof obligations (summarized in section 3.5.4) are discharged by over- and under-approximating the LHS (resulting in a weaker and a stronger proof obligation respectively), followed by discharging both proof obligations through SMT solvers. We overapproximate LHS by substituting each recursive relation $l_1 \sim l_2$ in the LHS with its d_o -depth overapproximation $l_1 \sim_{d_o} l_2$. Similarly, the LHS is underapproximated by substituting each recursive relation $l_1 \sim l_2$ with its d_u -depth underapproximation $l_1 \sim_{d_u} l_2$.

D-depth Iterative Unification and Rewriting Procedure

Recall that, section 3.5.3 briefly describes the process of reducing an overapproximate recursive relation into its SMT-encodable equivalent absent of recursive relations. We use modified versions of unification and ‘iterative unification and rewriting’ procedures (defined in sections 4.4.3 and 4.4.4 respectively) to reduce an overapproximate recursive relation into its SMT-equivalent. The *D*-depth unification and ‘iterative unification and rewriting’ procedures are represented by $\Theta_D(p_a, e_a, p_b, e_b, d)$ and $\theta_D(p_1, e_1, p_2, e_2, d)$ respectively, where *D* is a parameter of the algorithm. The pseudocode for these two procedures are shown in algorithms 7 and 8 respectively. The *D*-depth ‘iterative unification and rewriting’ returns depth-augmented correlation tuples of the form $\langle p_1, a_1, p_2, a_2 \rangle_d$ such that $d \geq D$ for all correlation tuples relating ADT values. Unlike Θ which terminates unification iff both expressions are atomic, Θ_D performs rewriting of both ADT atomic expressions and continues to unify deeper into their respective expression trees until all correlation tuples relate expressions at depth $\geq D$. For example, the 1-depth unification of l and $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p)$ yields the (depth augmented) correlation tuples: $\langle \text{true}, l \text{ is LNil}, \text{true}, p = 0 \rangle_0$, $\langle l \text{ is LCons}, l.\text{val}, p \neq 0, p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val} \rangle_1$ and $\langle l \text{ is LCons}, l.\text{tail}, p \neq 0, \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next}) \rangle_1$.

Reduction Procedure for Overapproximate Recursive Relations

$\text{Overapprox}_D(l_1, l_2)$ is responsible for reducing the overapproximation $l_1 \sim_D l_2$ into its SMT-encodable equivalent condition. Overapprox_D is similar to the decomposition procedure in

Algorithm 7: Pseudocode for D -Depth Unification Procedure

```

Function  $\theta_D(p_1, e_1, p_2, e_2, d)$ 
  if  $e_1$  is atomic then
    | return  $\text{Succ}(\{\langle p_1, e_1, p_2, e_2 \rangle_d\})$ ;
  else if  $e_2$  is atomic then
    | return  $\text{Succ}(\{\langle p_2, e_2, p_1, e_1 \rangle_d\})$ ;
  else if  $e_1 = V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n)$  and  $e_2 = V_2^{(m)}(e_2^1, e_2^2, \dots, e_2^m)$  then
    | if  $V_1^{(n)} \neq V_2^{(m)}$  then
    |   | return Fail;
    | end
    | return  $\bigsqcup_{i \in [1, n]} \theta_D(p_1, e_1^i, p_2, e_2^i, d + 1)$ ;
  else if  $e_1 = V_1^{(n)}(e_1^1, e_1^2, \dots, e_1^n)$  and  $e_2 = \text{if } c_2 \text{ then } e_2^{\text{th}} \text{ else } e_2^{\text{el}}$  then
    |  $R^{\text{th}} \leftarrow \theta_D(p_1, \text{true}, p_2, c_2, d) \sqcup \theta_D(p_1, e_1, p_2 \triangle c_2, e_2^{\text{th}}, d)$ ;
    | if  $R^{\text{th}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    |  $R^{\text{el}} \leftarrow \theta_D(p_1, \text{true}, p_2, \neg c_2, d) \sqcup \theta_D(p_1, e_1, p_2 \triangle \neg c_2, e_2^{\text{el}}, d)$ ;
    | if  $R^{\text{el}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    | return Fail;
  else if  $e_1 = \text{if } c_1 \text{ then } e_1^{\text{th}} \text{ else } e_1^{\text{el}}$  and  $e_2 = V_2^{(m)}(e_2^1, e_2^2, \dots, e_2^m)$  then
    |  $R^{\text{th}} \leftarrow \theta_D(p_1, c_1, p_2, \text{true}, d) \sqcup \theta_D(p_1 \triangle c_1, e_1^{\text{th}}, p_2, e_2, d)$ ;
    | if  $R^{\text{th}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    |  $R^{\text{el}} \leftarrow \theta_D(p_1, \neg c_1, p_2, \text{true}, d) \sqcup \theta_D(p_1 \triangle \neg c_1, e_1^{\text{el}}, p_2, e_2, d)$ ;
    | if  $R^{\text{el}} = \text{Succ}(S)$  then return  $\text{Succ}(S)$ ;
    | return Fail;
  else  $e_1 = \text{if } c_1 \text{ then } e_1^{\text{th}} \text{ else } e_1^{\text{el}}$  and  $e_2 = \text{if } c_2 \text{ then } e_2^{\text{th}} \text{ else } e_2^{\text{el}}$ 
    |  $R_1 \leftarrow \theta_D(p_1, c_1, p_2, c_2, d)$ ;
    |  $R_2 \leftarrow \theta_D(p_1 \triangle c_1, e_1^{\text{th}}, p_2 \triangle c_2, e_2^{\text{th}}, d)$ ;
    |  $R_3 \leftarrow \theta_D(p_1 \triangle \neg c_1, e_1^{\text{el}}, p_2 \triangle \neg c_2, e_2^{\text{el}}, d)$ ;
    | return  $R_1 \sqcup R_2 \sqcup R_3$ ;
  end
end

```

Algorithm 8: Pseudocode for D -Depth Iterative Unification and Rewriting Procedure

```

Function  $\Theta_D(p_a, e_a, p_b, e_b, d)$ 
   $R \leftarrow \emptyset$ ;
   $S \leftarrow \Theta_D(p_a, e_a, p_b, e_b, d)$ ;
  if  $S = \text{Fail}$  then return  $\text{Fail}$ ;
  foreach  $\langle p_1, a_1, p_2, e_2 \rangle_{d'}$  in  $S$  do
    if  $e_2$  is atomic then
      if  $d' \leq D$  and  $a_1$  is not scalar then
         $e_1 \leftarrow \text{rewrite}(a_1)$ ;
         $e_2^r \leftarrow \text{rewrite}(e_2)$ ;
         $R_1 \leftarrow \Theta_D(p_1, e_1, p_2, e_2^r, d')$  if  $R_1 = \text{Fail}$  then return  $\text{Fail}$ ;
         $R \leftarrow R \cup R_1$ ;
      else
         $R \leftarrow R \cup \{\langle p_1, a_1, p_2, e_2 \rangle_{d'}\}$ ;
      end
    else
       $e_1 \leftarrow \text{rewrite}(a_1)$ ;
       $R_1 \leftarrow \Theta_D(p_1, e_1, p_2, e_2, d')$ ;
      if  $R_1 = \text{Fail}$  then return  $\text{Fail}$ ;
       $R \leftarrow R \cup R_1$ ;
    end
  end
  return  $\text{Succ}(R)$ ;
end

```

Algorithm 9: Pseudocode of Reduction Procedure for Overapproximated Recursive Relations

```

Function  $\text{Overapprox}_D(l_1, l_2)$ 
   $ret \leftarrow \text{true}$ ;
   $\hat{l}_1 \leftarrow \text{Canonicalize}(l_1)$ ;
   $\hat{l}_2 \leftarrow \text{Canonicalize}(l_2)$ ;
   $R \leftarrow \Theta_D(\text{true}, \hat{l}_1, \text{true}, \hat{l}_2, 0)$ ;
  if  $R = \text{Fail}$  then return  $\text{false}$ ;
  foreach  $\langle p_1, a_1, p_2, a_2 \rangle_d$  in  $R$  do
    if  $a_1$  is scalar and  $d \leq D$  then
       $ret \leftarrow ret \wedge (p_1 \wedge p_2) \rightarrow (a_1 = a_2)$ ;
    end
  end
  return  $ret$ ;
end

```

Algorithm 10: Pseudocode for *isDepthBounded* Procedure

```

Function isDepthBoundedD(l, p, d)
  if d > D then return  $\neg p$ ;
  if l is atomic then
    if l is scalar then return true;
    else
      lr  $\leftarrow$  rewrite(l);
      return isDepthBoundedD(lr, p, d);
    end
  else if l =  $V^{(n)}(l^1, l^2, \dots, l^n)$  then
    return  $\bigwedge_{i \in [1, n]} \text{isDepthBounded}_D(l^i, p, d + 1)$ 
  else l = if c then lth else lel
    condth  $\leftarrow$  isDepthBoundedD(lth, p  $\wedge$  c, d);
    condel  $\leftarrow$  isDepthBoundedD(lel, p  $\wedge$   $\neg c$ , d);
    return condth  $\wedge$  condel
  end
end

```

section 4.4.5 except it only preserves scalar equalities till a maximum depth of D (inclusive). This essentially asserts that both l_1 and l_2 have identical structures (by equating expression path conditions) and equal scalar values (by equating scalar leaf expressions) till a depth of D . For example, recall that l and $\text{Clist}_{\text{m}}^{\text{lnode}}(p)$ 1-depth unifies into the correlation tuples: $\langle \text{true}, l \text{ is LNil}, \text{true}, p = 0 \rangle_0$, $\langle l \text{ is LCons}, l.\text{val}, p \neq 0, p \xrightarrow{\text{m}}_{\text{lnode}} \text{val} \rangle_1$ and $\langle l \text{ is LCons}, l.\text{tail}, p \neq 0, \text{Clist}_{\text{m}}^{\text{lnode}}(p \xrightarrow{\text{m}}_{\text{lnode}} \text{next}) \rangle_1$. Keeping only the scalar clauses till depth 1, $l \sim_1 \text{Clist}_{\text{m}}^{\text{lnode}}(p)$ reduces to: $((l \text{ is LNil}) = (p = 0)) \wedge ((l \text{ is LCons}) \wedge (p \neq 0) \rightarrow l.\text{val} = p \xrightarrow{\text{m}}_{\text{lnode}} \text{val})$.

Reduction Procedure for Underapproximate Recursive Relations

Recall that, an underapproximate recursive relation $l_1 \approx_{du} l_2$ is equivalent to $\Gamma_{du}(l_1) \wedge \Gamma_{du}(l_2) \wedge l_1 \sim_{du} l_2$, where $\Gamma_d(l)$ asserts that l has a depth of at most d (defined in section 3.5.2). The function responsible for computing $\Gamma_D(l)$ is *isDepthBounded*_D(*l*, *p*, *d*), where *p* and *d* represents the current expression path condition and depth of *l*, and D is a parameter of the algorithm. Algorithm 10 gives the pseudocode for *isDepthBounded*_D. The top-level invocation is given by *isDepthBounded*_D(*l*, *true*, 0). *isDepthBounded*_D recursively traverses the expression tree of *l* (while rewriting as necessary), until it reaches a node at depth $> D$, at which point it returns

Algorithm 11: Pseudocode of Reduction Procedure for Underapproximated Recursive Relations

```

Function  $Underapprox_D(l_1, l_2)$ 
   $\hat{l}_1 \leftarrow Canonicalize(l_1);$ 
   $\hat{l}_2 \leftarrow Canonicalize(l_2);$ 
   $overapprox \leftarrow Overapprox_D(\hat{l}_1, \hat{l}_2);$ 
   $depthbound_1 \leftarrow isDepthBounded_D(\hat{l}_1, true, 0);$ 
   $depthbound_2 \leftarrow isDepthBounded_D(\hat{l}_2, true, 0);$ 
  return  $overapprox \wedge depthbound_1 \wedge depthbound_2;$ 
end

```

the condition asserting the unreachability of such a node. For example, for a `List` variable l , $\Gamma_2(l)$ (i.e. $isDepthBounded_2(l, true, 0)$) returns the predicate: $\neg(l \text{ is } LCons \wedge l.tail \text{ is } (LCons))$. Expanding $e_1 \wedge e_2$ to $e_1 \wedge (e_1 \rightarrow e_2)$ and using the identity $\neg(e \text{ is } LNil) = e \text{ is } LCons$, the above reduces to the equivalent condition: $(l \text{ is } LNil) \vee ((l \text{ is } LCons) \wedge (l.tail \text{ is } LNil))$.

Finally, $Underapprox_D(l_1, l_2)$ is responsible for reducing the underapproximation $l_1 \approx_D l_2$ into its SMT-encodable equivalent condition. The pseudocode for $Underapprox_D$ is given in algorithm 11. For example, $l \approx_1 Clist_m^{lnode}(p) \Leftrightarrow \Gamma_1(l) \wedge \Gamma_1(Clist_m^{lnode}(p)) \wedge l \sim_1 Clist_m^{lnode}(p)$. $\Gamma_1(l)$ and $\Gamma_1(Clist_m^{lnode}(p))$ reduces to the conditions $l \text{ is } LNil$ and $(p = 0)$ respectively. Finally, $l \approx_1 Clist_m^{lnode}(p)$ reduces to the condition: $l \text{ is } LNil \wedge (p = 0) \wedge ((l \text{ is } LNil) = (p = 0)) \wedge ((l \text{ is } LCons) \wedge (p \neq 0) \rightarrow l.val = p \xrightarrow{m}_{lnode} val)$.

4.4.7 SMT Encoding of First Order Logic Formula

As summarized in fig. 3.8, our proof discharge algorithm solves a proof obligation $P : LHS \Rightarrow RHS$, through a sequence of queries $P_i : LHS_i \Rightarrow RHS_i$ to off-the-shelf SMT solvers. Recall that P may contain recursive relations. However, our algorithm ensures that each P_i is free of recursive relations and only contain scalar equalities. We encode each query P_i in SMT logic with bitvector and array theories. In this section, we describe the process of encoding a proof obligation P_i into SMT logic. We begin by converting $P_i : LHS_i \Rightarrow RHS_i$ into its canonical form \hat{P}_i (as described in section 4.4.2). Although \hat{P}_i does not contain recursive relations, it may still contain ADT variables alongside *accessor* and *sum-is* expressions. Due to canonicalization, all top-level *accessor*

and *sum-is* expressions must be of the form $v.a_1.a_2\dots a_n$ and $v.a_1.a_2\dots a_n$ is V respectively. We call such an expression e *flattenable* and the ADT variable v is called the *index* of e . \hat{P}_i is lowered into an intermediate expression P_i^f through a process called *flattening*. This involves ‘flattening’ all flattenable expressions to variables such that P_i^f only contains scalar values with scalar and memory operations (but importantly not ADT values). The flattening process is described below.

1. For each top-level *accessor* expression $e = v.a_1.a_2\dots a_n$, we replace it with a variable named $v \parallel a_1 \parallel a_2 \parallel \dots \parallel a_n$, where \parallel concatenates two strings with a ‘_’ character in between i.e. “a” \parallel “b” = “a_b”.
2. For each ADT T with data constructors V_1, V_2, \dots, V_k , we define an enumeration type $\mathcal{E}(T)$ in SMT logic with items $\mathcal{E}(V_1), \mathcal{E}(V_2), \dots, \mathcal{E}(V_k)$ respectively. In the canonical form, each *sum-is* expression e must operate on a pseudo-variable. The last step guarantees that e must be of the form: $e = v$ is V . We replace e with its SMT equivalent: $(v \parallel tag) = \mathcal{E}(V)$ ².

For example, the canonical expression $a + l.val$ flattens to $a + l_val$. Similarly, $(l.tail$ is $LCons)$ flattens to $l_tail_tag = \mathcal{E}(LCons)$. Due to flattening, each flattenable expression e in \hat{P}_i with index v gets lowered into a variable in P_i^f whose name begins with $v_$. For the ADT variable v , let $\mathcal{F}(v)$ be the set of all such lowered variables in P_i^f . For example, flattening of an expression with $l.val$ and l is $LCons$ results in $\mathcal{F}(l) = \{l_val, l_tag\}$. Importantly, P_i^f may only contain scalar and memory operations (but not ADT values).

Scalar types and their operations map one-to-one to their SMT equivalents. The memory element \mathfrak{m} is represented as a byte-addressable (i.e. `i8`) array. A memory load $\mathfrak{m}[a]_T$ is expanded into the concatenation of `sizeof(T)` *array-select* operations. A memory write $\mathfrak{m}[a \leftarrow v]_T$ is expanded into `sizeof(T)` nested *array-store* operations.

Algorithm 12: Pseudocode for Reconciliation Procedure

```

Function Reconcile( $v : T, \gamma$ )
  if  $T$  is scalar then
    if  $\gamma$  maps  $v$  then return  $\gamma[v]$ ;
    else return  $Rand(T)$ ;
  else
    if  $\gamma$  maps  $v \parallel tag$  then
       $E^V \leftarrow \gamma[v]$ ;
       $args \leftarrow []$ ;
      Let  $[a_1 : T_1, a_2 : T_2, \dots, a_n : T_n]$  be the fields of  $V$ .
      foreach  $(a' : T')$  in  $[a_1 : T_1, a_2 : T_2, \dots, a_n : T_n]$  do
         $arg \leftarrow Reconcile(v \parallel a' : T', \gamma)$ ;
         $args.append(arg)$ ;
      end
      return  $V(args)$ ;
    else
      return  $Rand(T)$ ;
    end
  end
end

```

4.4.8 Reconciliation of Counterexamples

As detailed in section 4.4.7, each ADT variable v gets lowered into a set of scalar variables $\mathcal{F}(v)$ during SMT encoding. Evidently, the models returned by SMT solvers map these variables (in $\mathcal{F}(v)$ instead of v) to constant values. We are interested in recovering a counterexample for the original query from a model returned by the SMT solver. Recall that, these counterexamples help guide the correlation search (in section 4.2) and invariant inference (in section 4.3) procedures. The process of constructing a constant for v from the constant values returned for $\mathcal{F}(v)$ by an SMT solver is called *reconciliation*. Obviously, the reconciled counterexample must be a valid counterexample to the original proof obligation. *Reconcile*($v : T, \gamma$) is responsible for performing reconciliation for variable v (of type T) from the model γ (returned by a SMT solver). *Rand*(T) returns an arbitrary constant of type T . For example, consider the rather contrived proof obligation $P : \text{true} \Rightarrow l$ is LNil. Clearly, any valuation of l where l is a non-empty list is a valid counterexample to P . However, a counterexample γ returned by an SMT solver would

²Spec does not allow naming a field of a data constructor **tag**. Furthermore, fields cannot contain the ‘_’ character. Combined, these two conditions prevent collision between variable names obtained due to flattening.

instead contain the mapping $\{l_tag \mapsto \mathcal{E}(\text{LCons})\}$. During reconciliation, we find that l_tag is mapped to the data constructor **LCons** and recurse for each of its fields **val** and **tail**. Since γ do not contain a mapping for either of these fields, we soundly generate random constants for these instead. Indeed, *Reconcile* correctly constructs a non-empty but otherwise arbitrary list for l , which is a counterexample to P .

4.4.9 Value Tree Representation

This section presents a graphical representation of expressions that helps us simplify the implementation of multiple subprocedures used by our proof discharge algorithm. This includes the process of canonicalization, reducing approximate recursive relations as well as construction of deconstruction programs as part of type III proof obligations. We call this the *Value Tree* representation and use $\mathcal{V}(e)$ to denote a value tree associated with e . We give an algorithm to convert an expression e into $\mathcal{V}(e)$ and list its applications.

Before diving into value trees, we start by introducing an analogous (but simpler) representation for types, called *Type Trees*. We use $\mathcal{T}(\tau)$ to denote a type tree associated with τ . Recall that ADTs are simply ‘sum of product’ types where each data construction represents a variant (of the sum-type) and each data construction contains values for each of its fields (of the product-type). On top of ADTs, IR has build-in scalar types: **unit**, **bool** and **i<N>**. Types in IR can be represented in *first order recursive types* [26] using the product (\times) and sum ($+$) type constructors; and the scalar types (i.e. nullary type constructors). The type system is characterized by the grammar \mathbb{T} as follows:

$$T \rightarrow \mu\alpha. T \mid T \times \cdots \times T \mid T + \cdots + T \mid \text{unit} \mid \text{bool} \mid \text{i}\langle N \rangle \mid \alpha$$

Every IR type can be encoded as a closed term (i.e. term without free variables) in \mathbb{T} . For example, the **List** type can be written as $\mu\alpha.\text{unit} + (\text{i}32 \times \alpha)$. Note the use of a type variable α which is bound using μ to represent recursion. Similarly, the **Matrix** type is represented by the term $\mu\alpha.\text{unit} + ((\mu\beta.\text{unit} + (\text{i}32 \times \beta)) \times \alpha)$, where the type variables α and β are used to bind

recursive types `Matrix` and `List` at their definitions respectively.

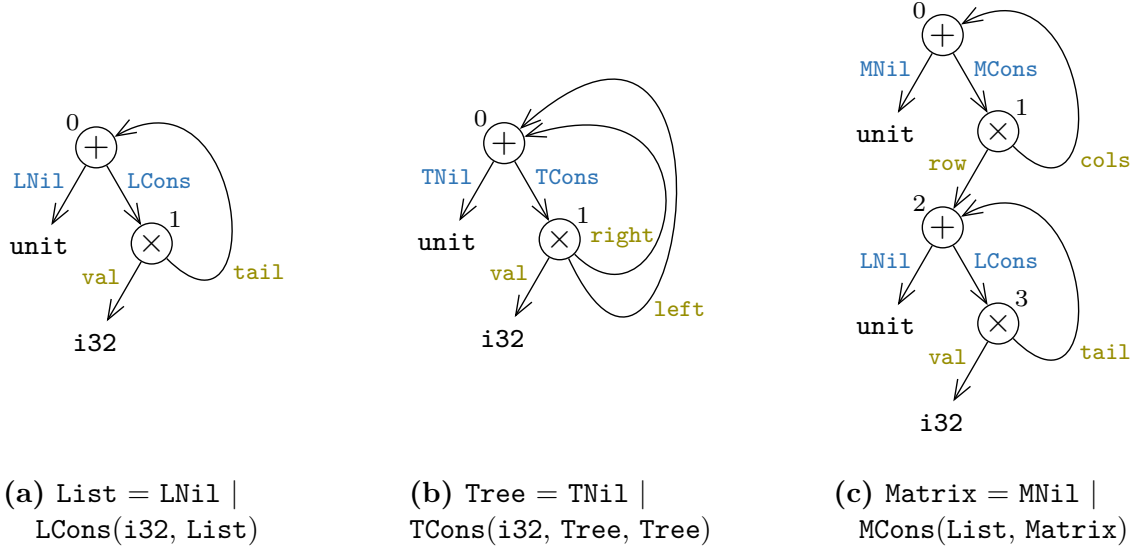
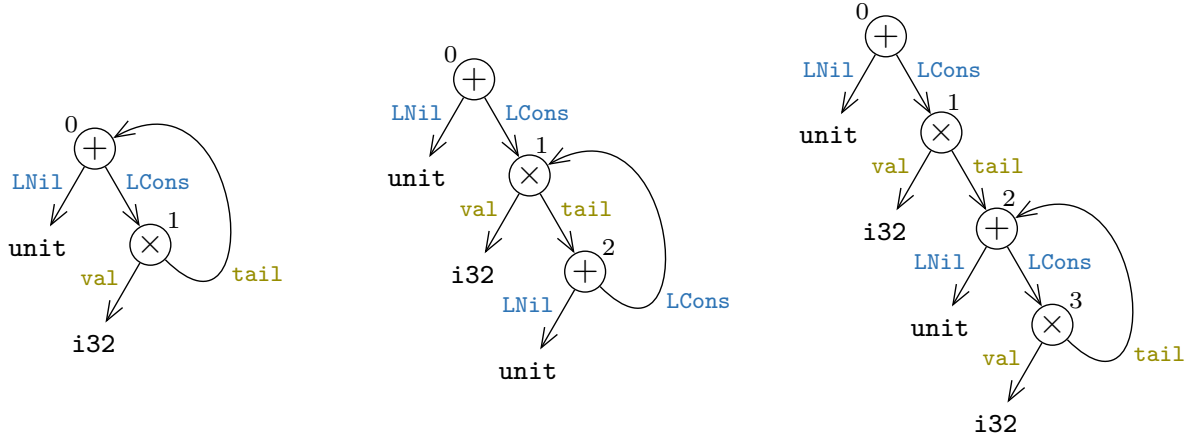


Figure 4.3: Type trees for the ADTs `List`, `Tree` and `Matrix` respectively.

Figure 4.3 shows the type trees for three ADTs `List`, `Tree`, and `Matrix` respectively. In a type tree, each internal node represents either a product (\otimes) or a sum (\oplus) type constructor. The leaf nodes are the scalar types. Each outgoing edge of a \oplus node is associated with a data constructor of the corresponding ADT (i.e. `LCons` for `List`). Similarly, each outgoing edge of a \otimes node is associated with a field of the corresponding data constructor (i.e. `val` for `LCons`). We assign integer indices to the internal nodes and use $[v \rightarrow \text{label}]^3$ to identify the edge outgoing at v associated with `label`, where `label` is either a data constructor or a field name. The root node is denoted by v_0 . The edges going outward from the root node are called *tree-edges* e.g., $[0 \rightarrow \text{LCons}]$ and $[1 \rightarrow \text{val}]$ in fig. 4.3a. Edges that are not tree-edges, are called *back-edges* e.g., $[1 \rightarrow \text{cols}]$ in fig. 4.3c. Every back-edge induces an unique simple cycle in the type tree representation.

³The $[v \rightarrow \text{label}]$ syntax is chosen over $v_1 \rightarrow v_2$ because type trees may have multi-edges, in which case $v_1 \rightarrow v_2$ no longer represents an unique edge as shown in fig. 4.3b.



(a) Canonical List ADT

(b) Peeled List ADT

(c) Unrolled List ADT

Figure 4.4: Three type trees for List ADT. Figure 4.4a shows the type tree for the canonical form of List. Figure 4.4b is obtained by peeling the back-edge $[1 \rightarrow \text{tail}]$ in fig. 4.4a.

Figure 4.4c is obtained by unrolling the back-edge $[1 \rightarrow \text{tail}]$ in fig. 4.4a or by peeling the back-edge $[2 \rightarrow \text{LCons}]$ in fig. 4.4b respectively.

Recall that types in Spec (and in IR) follow equirecursive typing rules i.e. types $\mu\alpha.T$ and $T[\mu\alpha.T/\alpha]$ in \mathbb{T} are *equal* types, where $T[\mu\alpha.T/\alpha]$ represents the new type obtained by substituting all free instances of α with $\mu\alpha.T$, and is defined as the *unfolding* of $\mu\alpha.T$. In general, under equirecursive typing, two types are equal iff their infinite expansions (through unfolding) are equal. In the type tree representation, two types are equal iff their infinite expansions are equivalent. Such type trees are called isomorphic and two types are isomorphic iff they represent the same type. An unfolding in the term representation corresponds to *unrolling* one iteration of a simple cycle in its type tree. Figure 4.4 shows three type trees for the List type. Figure 4.4a corresponds to the canonical (intuitively the ‘smallest’) type tree for the List type. The type trees figs. 4.4b and 4.4c are obtained by *peeling* and unrolling the back-edge $[1 \rightarrow \text{tail}]$ (in fig. 4.4a) respectively. Peeling is a form of partial unrolling which only extracts the starting node of the cycle. In practice, equality of two types (encoded in \mathbb{T}) can be reposed as syntactic equality of their *canonical forms* [19]. In general, type trees may contain cycles (due to back-edges) and hence are not quite ‘trees’. However, they represent the actual (possibly infinite) trees obtained through repeated unrolling

of cycles.

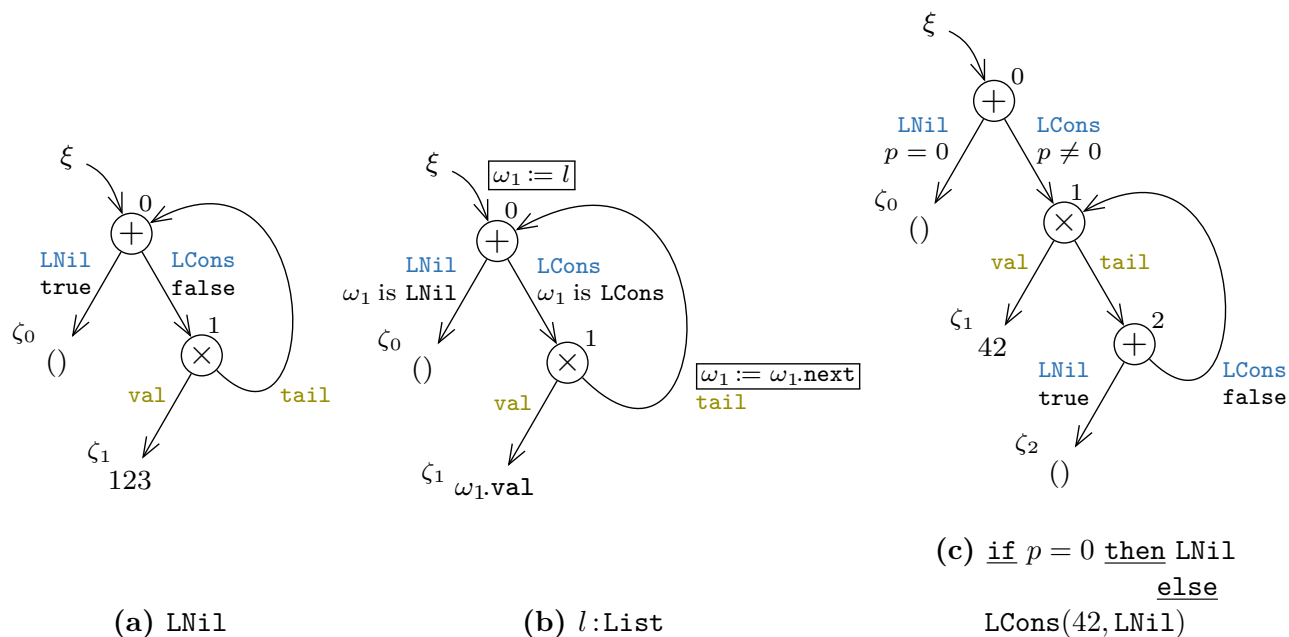


Figure 4.5: Value trees of three `List`-typed expressions

With type trees out of the way, we are ready to present their value analogue called ‘value trees’. Figure 4.5 shows the value trees for three **List** expressions. Note that, all three value trees are isomorphic to one of the **List** type trees shown in fig. 4.4, e.g., fig. 4.5c is isomorphic to fig. 4.4b. In general, for an expression e of type τ , its value tree $\mathcal{V}(e)$ resembles its type tree with the following distinctions:

1. Similar to a type tree, each internal node is either a \oplus or a \otimes node.
2. Instead of a scalar type τ_s , each leaf node in $\mathcal{V}(e)$ contains an expression of type τ_s .
3. Similar to the Control-Flow Graph representation (presented in section 2.2.3), each node v is associated with a symbolic state Ω_v .
4. In addition to a data constructor, each edge originating at a \oplus node v also contains an

edge condition expression (a boolean valued function over Ω_v). We identify such an edge with $[v \xrightarrow{c} V]$, where v is the sum node, V is a data constructor and c is the edge condition. The set of edge conditions for all outgoing edges at a \oplus node must be mutually exclusive and exhaustive.

5. In addition to a field name, each edge $v \rightarrow v'$ originating at a \otimes node v also contains a transfer function ($\Omega_{v'}$ as a function of Ω_v). We identify such an edge with $[v \xrightarrow{\text{tf}} \text{fi}]$, where v is the product node, fi is a field name and tf is the transfer function.
6. Additionally, a value tree also contains a special node (called the *entry node*), and a special edge (called the *entry edge*) from the entry node to v_0 (i.e. the root of the tree). We use ξ to denote the entry node. The entry edge is associated with a transfer function tf_ξ . We often omit the entry node in figures for brevity.
7. A value tree $\mathcal{V}(e)$ can be converted to a type tree \mathcal{T} as follows: (a) remove the entry node and edge pair, (b) remove edge conditions and transfer functions associated with all edges, and (c) replace each leaf node expression of (scalar) type τ_s with τ_s itself. The resulting type tree \mathcal{T} represents the type τ of the expression e .

Intuitively, a value tree simultaneously represents the value of the expression as well as the CFG of its *abstracted* deconstruction program. We will subsequently discuss these properties along with their applications in the context of our proof discharge algorithm. Next, we give an algorithm to convert an expression e to its value tree representation $\mathcal{V}(e)$.

4.4.10 Conversion of Expressions to their Value Trees

In this section, we present an algorithm to recursively construct a value tree for any arbitrary expression e . We take a visual approach to align with the graphical nature of value trees.

$$\mathcal{V}(e_i) = \begin{array}{c} \boxed{\mathbf{tf}_i} \\ \searrow \\ e'_i \end{array} \quad \mathcal{V}(e_1 \odot e_2) = \begin{array}{c} \searrow \\ \mathbf{tf}_1(e'_1) \odot \mathbf{tf}_2(e'_2) \end{array}$$

Figure 4.6: Construction of $\mathcal{V}(e_1 \odot e_2)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$.
 \odot represents an arbitrary scalar operator.

Scalar Operators

Given an expression $e = e_1 \odot e_2$, fig. 4.6 shows the construction of $\mathcal{V}(e)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$ respectively. Since e_1 and e_2 have scalar types, their value trees must have exactly one node (i.e. the leaf node) containing an expression (e'_1 and e'_2 respectively) of the same type. \odot represents an arbitrary scalar operator, i.e. an operator whose arguments are scalar-typed values (e.g., bitvector arithmetic and relational operators). Given an expression s and a transfer function \mathbf{tf} , $\mathbf{tf}(s)$ represents the expression obtained by applying \mathbf{tf} , interpreted as a substitution, to s . This is equivalent to the weakest-precondition of s along an edge associated with the \mathbf{tf} . The construction shown in fig. 4.6 can be generalized to n -ary scalar operators for $n > 2$.

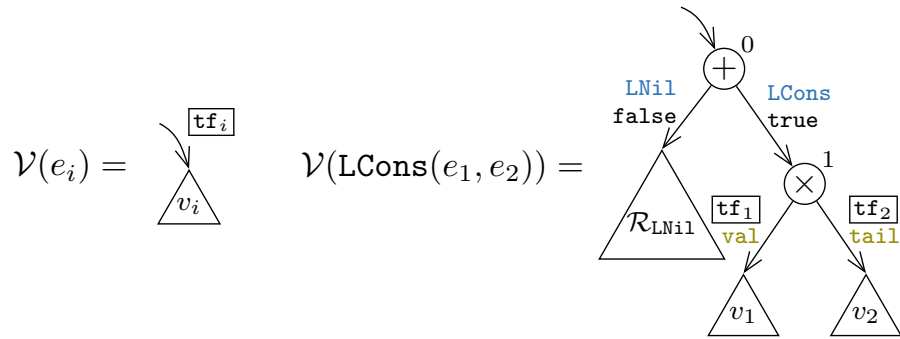


Figure 4.7: Construction of $\mathcal{V}(\text{LCons}(e_1, e_2))$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$.

$\mathcal{R}_{\text{LNil}}$ represents an arbitrary value tree corresponding to the product-type (in \mathbb{T}) associated with LNil .

ADT Data Constructors

Given an expression $e = \text{LCons}(e_1, e_2)$, fig. 4.7 depicts the construction of $\mathcal{V}(e)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$ respectively. In general, for an arbitrary data constructor V of ADT T , the process begins with a \oplus node (0 in fig. 4.7) such that the outgoing edge associated with the value constructor V (LCons in fig. 4.7) has an edge condition of **true** while all other edges are assigned the edge condition **false**. For each data constructor $V' \neq V$ of T , we append a random value tree corresponding to the product-type associated with V' in \mathbb{T} . For example, given **List** is associated with the sum-type $\mu\alpha.\text{Unit} + (\text{i32} \times \alpha)$, the product-types associated with **LNil** and **LCons** are: **Unit** and $\mu\alpha.\text{i32} \times (\text{Unit} + \alpha)$ respectively. We use \mathcal{R}_τ to denote an arbitrary (i.e. random) value tree of type τ . For the outgoing edge associated with the data constructor V ($[0 \xrightarrow{\text{true}} \text{LCons}]$ in fig. 4.7), we construct a product node (1 in fig. 4.7) and append the value trees corresponding to the arguments e_i as children of the product node.

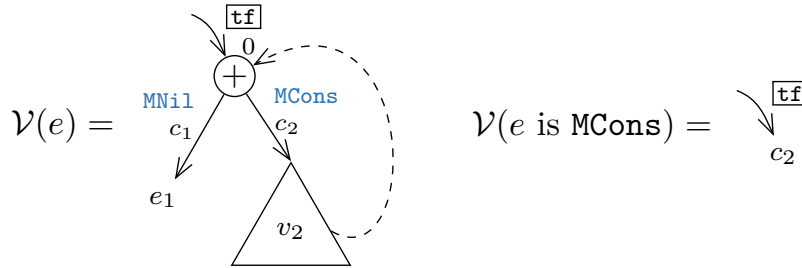


Figure 4.8: Construction of $\mathcal{V}(e \text{ is MCons})$ from $\mathcal{V}(e)$.

The dashed edge represents the (possibly empty) set of backedges originating in v_2 that terminates at 0.

Sum-Is Operator

Given a *sum-is* expression $e' = e \text{ is MCons}$, fig. 4.8 shows the construction of $\mathcal{V}(e')$ from $\mathcal{V}(e)$. The process is rather straightforward and for a general expression $e \text{ is } V_i$, entails extracting the edge condition c (c_2 in fig. 4.8) from the $\mathcal{V}(e \text{ is } V_i)$ edge $[v_0 \xrightarrow{c} V]$ ($[0 \xrightarrow{c_2} \text{MCons}]$ in fig. 4.8). Notice that the entry transfer function tf_ξ during this conversion.

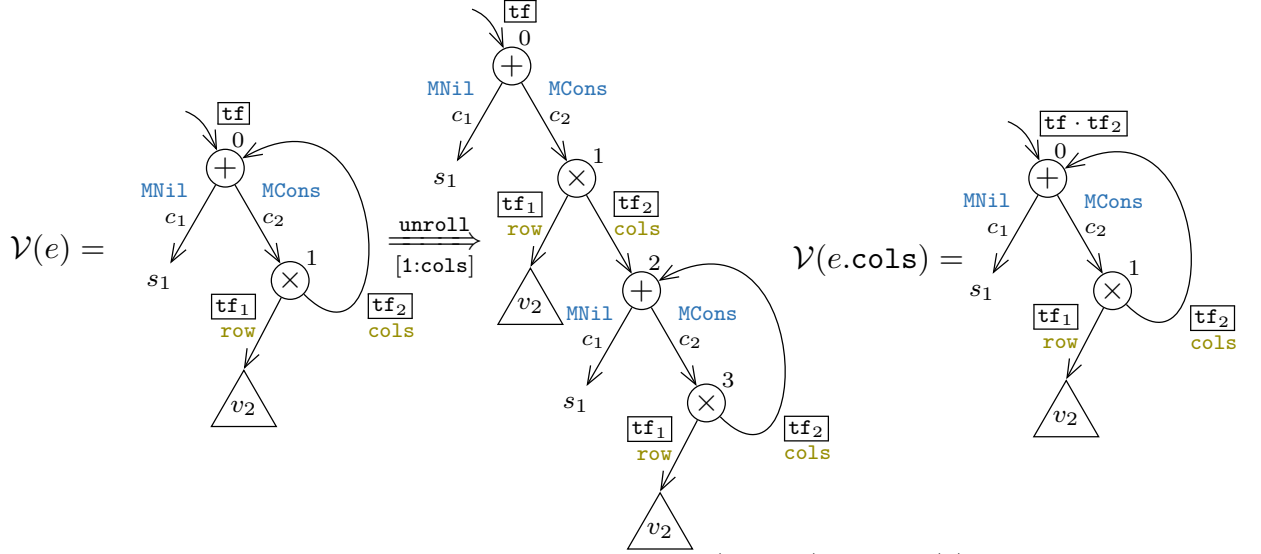


Figure 4.9: Construction of $\mathcal{V}(e.cols)$ from $\mathcal{V}(e)$.

Similar to type trees, `unroll [1 → cols]` represents the operation of hoisting one iteration of the cycle $0 \rightarrow 1 \rightarrow 0$.

Product-Access Operator

Given an expression $e' = e.cols$, fig. 4.9 depicts the construction of $\mathcal{V}(e')$ from $\mathcal{V}(e)$. Intuitively, $\mathcal{V}(e')$ represents the subtree of $\mathcal{V}(e)$ rooted at the $(+)$ node reached by taking the edges $[0 \xrightarrow{c_2} MCons]$ followed by $[1 \xrightarrow{\Omega_2} cols]$. However, this path may contain backedges or the subtree itself may contain backedges leaving the subtree. In such a case, we perform peeling until all such backedges strictly terminate within this subtree. For example, in fig. 4.9, the edge $[1 \xrightarrow{\Omega_2} cols]$ is a backedge and hence we peel it once. In the resulting (equivalent) value tree, the subtree (rooted at 2) contains a backedge leaving the subtree which requires one more peeling operation. The resulting value tree contains the subtree rooted at the $(+)$ node 2 which satisfies the two conditions above and hence $\mathcal{V}(e')$ is simply constructed by extracting the subtree rooted at $(+)$ node 2. Note that we preserve the transfer functions from the entry to the $(+)$ node 2 during extraction. $tf_1 \cdot tf_2$ represents the composition of the transfer functions tf_1 and tf_2 respectively.

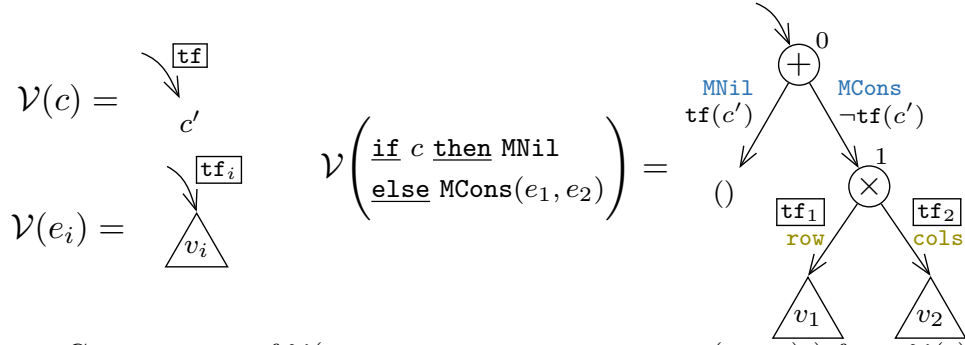


Figure 4.10: Construction of $\mathcal{V}(\text{if } c \text{ then MNil else MCons}(e_1, e_2))$ from $\mathcal{V}(c)$, $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$.

If-Then-Else Operator

Given an expression $e = \text{if } c \text{ then MNil else MCons}(e_1, e_2)$, fig. 4.10 describes the construction of $\mathcal{V}(e)$ using $\mathcal{V}(c)$, $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$. Let us consider a general if-then-else expression e (associated with the ADT T with data constructors V_1, V_2, \dots, V_n) such that the branch associated with V_i is given by $V_i(e_i^1, e_i^2, \dots)$. We begin with the construction of a (\oplus) root node (0 in fig. 4.10) such that the outgoing edge associated with V_i has the edge condition equal to the expression path condition of the branch $V_i(e_i^1, e_i^2, \dots)$ ($\text{tf}(c')$ and $\neg\text{tf}(c')$ for MNil and MCons respectively in fig. 4.10). For each outgoing edge associated with the data constructor V_i , we construct a (\otimes) node (1 for MCons in fig. 4.10) and append the value trees corresponding to the arguments e_i^j as its children.

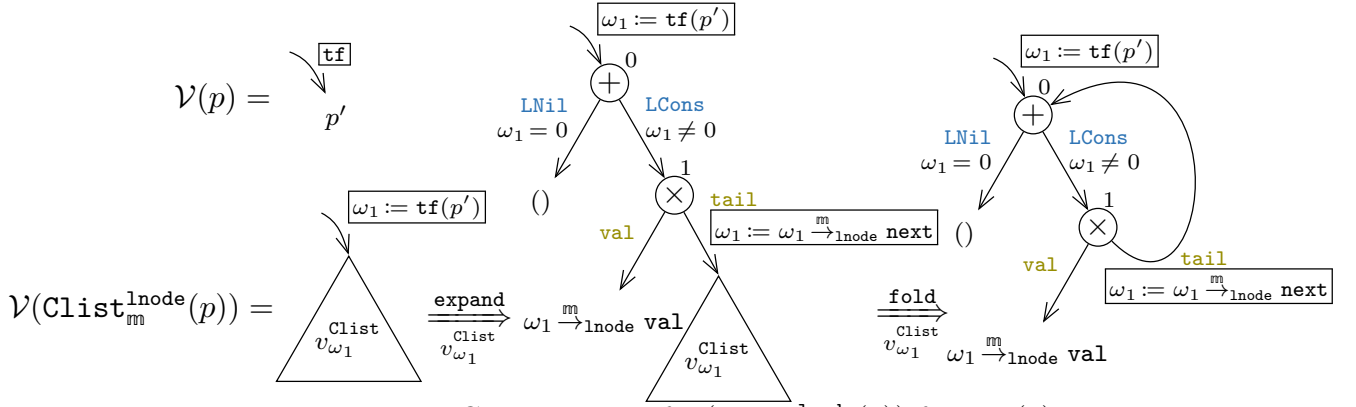


Figure 4.11: Construction of $\mathcal{V}(\text{Clist}_m^{\text{lnode}}(p))$ from $\mathcal{V}(p)$.

The process involves assuming $v_{\omega_1}^{\text{Clist}}$ to be the value tree of $\text{Clist}_m^{\text{lnode}}(\omega_1)$, followed by expansion of $v_{\omega_1}^{\text{Clist}}$ using the definition of $\text{Clist}_m^{\text{lnode}}$ (in eq. (2.2)) and, finally folding the tree-edge $[1 \rightarrow \text{tail}]$ incident on the self-referential subtree $v_{\omega_1}^{\text{Clist}}$ into a backedge.

Lifting Constructor

Given an expression $e = \text{Clist}_m^{\text{lnode}}(p)$, fig. 4.11 shows the construction of $\mathcal{V}(e)$ from $\mathcal{V}(p)$. Recall the recursive definition of the lifting constructor $\text{Clist}_m^{\text{lnode}}$ given in eq. (2.2). We start by assuming that $v_{\omega_1}^{\text{Clist}}$ is the value tree for the lifted expression $\text{Clist}_m^{\text{lnode}}(\omega_1)$. Hence, the value tree of $\text{Clist}_m^{\text{lnode}}(p)$ is identical to $v_{\omega_1}^{\text{Clist}}$ except we assign the actual argument (i.e. $\text{tf}(p')$) to the formal argument ω_1 along the entry edge. Next, we expand the subtree $v_{\omega_1}^{\text{Clist}}$ based on the unrolling procedure of $\text{Clist}_m^{\text{lnode}}$ (defined in eq. (2.2)) until the entire value tree becomes a self-referential structure. For example, after expanding through eq. (2.2) once in fig. 4.11, the value tree contains an tree-edge $[1 \rightarrow \text{tail}]$ incident on the self-referential subtree $v_{\omega_1}^{\text{Clist}}$. In the last step, we fold all self-referential tree-edges ($[1 \rightarrow \text{tail}]$ in fig. 4.11) by converting them into backedges terminating at the root of the subtree being referenced (0 for $v_{\omega_1}^{\text{Clist}}$ in fig. 4.11).

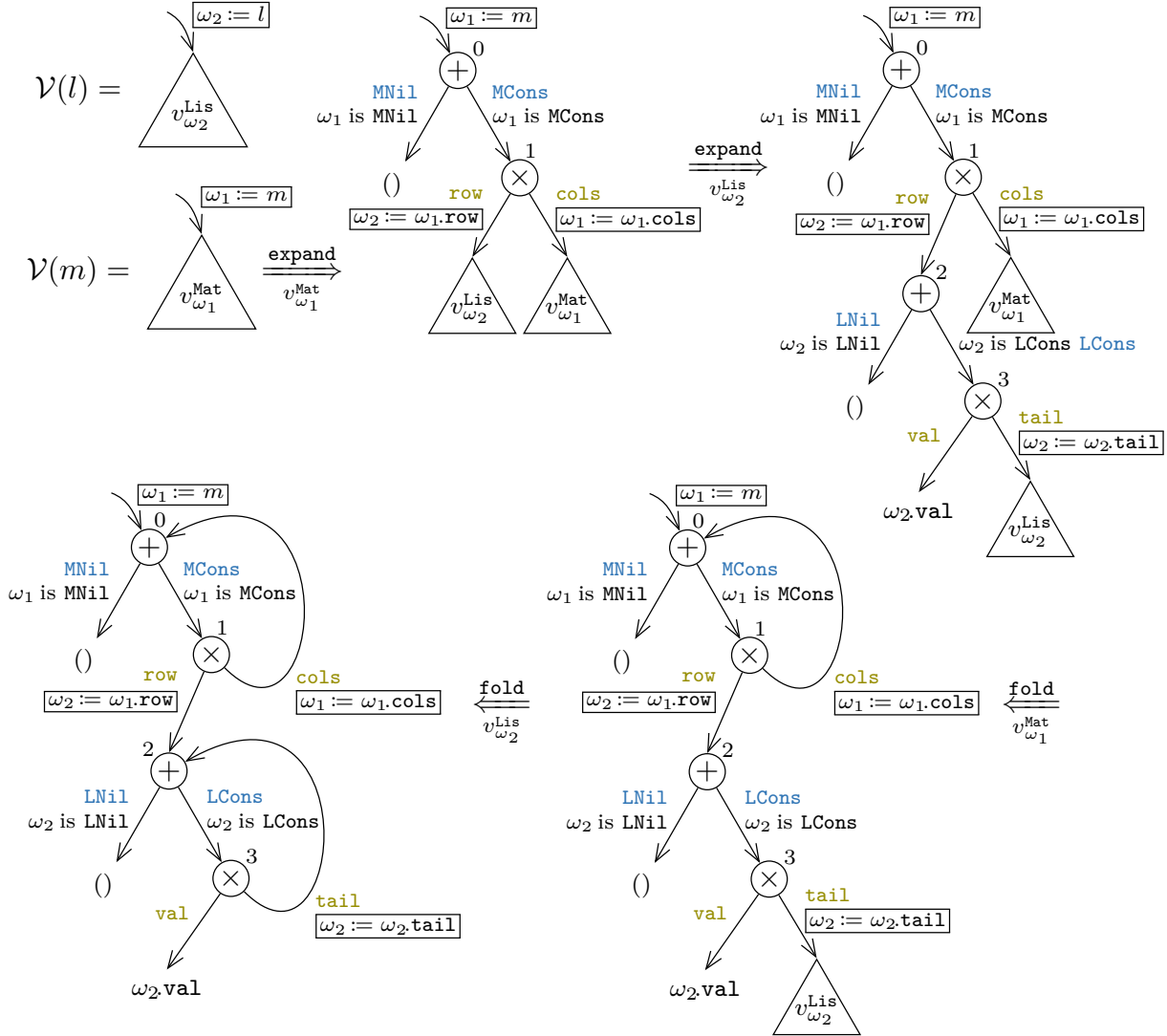


Figure 4.12: Construction of $\mathcal{V}(m)$ for a Matrix variable m . The process is identical to the construction of value trees for lifted expressions as shown in fig. 4.11.

Variables

Finally, we are interested in constructing the value tree for a variable. Recall that, every ADT (pseudo-)variable is associated with an unrolling procedure characterized by the ADT

itself. e.g. eq. (2.1) for the `List` variable l . The `Matrix` ADT is defined as: `Matrix = MNil | MCons(List, Matrix)`, and thus the unrolling procedure for a `Matrix` variable m is given by:

$$m = \text{if } m \text{ is MNil then MNil else MCons}(m.\text{row}, m.\text{cols}) \quad (4.2)$$

Figure 4.12 illustrates the construction $\mathcal{V}(m)$ for the `Matrix` variable m . The process consists of the same three steps used to construct the value tree for a lifted expression – assume, expand and fold. First, we assume that $v_{\omega_1}^{\text{Mat}}$ and $v_{\omega_2}^{\text{Lis}}$ are the value trees corresponding to the pseudo-variables ω_1 and ω_2 of `Matrix` and `List` types respectively. Thus, $\mathcal{V}(m)$ is equal to $v_{\omega_1}^{\text{Mat}}$ with the entry edge transfer function $\{\omega_1 \leftarrow m\}$. We expand the definitions of $v_{\omega_1}^{\text{Mat}}$ and $v_{\omega_2}^{\text{Lis}}$ once each before the value tree becomes self-referential. Finally, we fold the treeedges $[1 \rightarrow \text{cols}]$ and $[3 \rightarrow \text{tail}]$ into the back-edges terminating at the roots of the subtrees representing $v_{\omega_1}^{\text{Mat}}$ and $v_{\omega_2}^{\text{Lis}}$ respectively (nodes 0 and 3 in fig. 4.12).

4.4.11 Applications of Value Trees

With the conversion algorithm out of the way, we next discuss a handful of properties of value trees along with their applications, in the context of our proof discharge algorithm.

Program Interpretation of Value Trees

A value tree \mathcal{V} can be interpreted as a *non-deterministic* Control-Flow Graph corresponding to a program. Similar to the CFG representation, every node v is associated with a symbolic state Ω_v . In this interpretation, edges without an edge condition (e.g., outgoing at a \otimes node) are assigned the `true` edge condition. Similarly, edges without a transfer function (e.g., outgoing at a \oplus node) are assigned an identity transfer function. The entry node ξ is the start node and each leaf node ζ represents an exit node. An edge incident on a leaf node ζ (containing an expression e) is called an exit edge and is associated with an observable action that returns e .

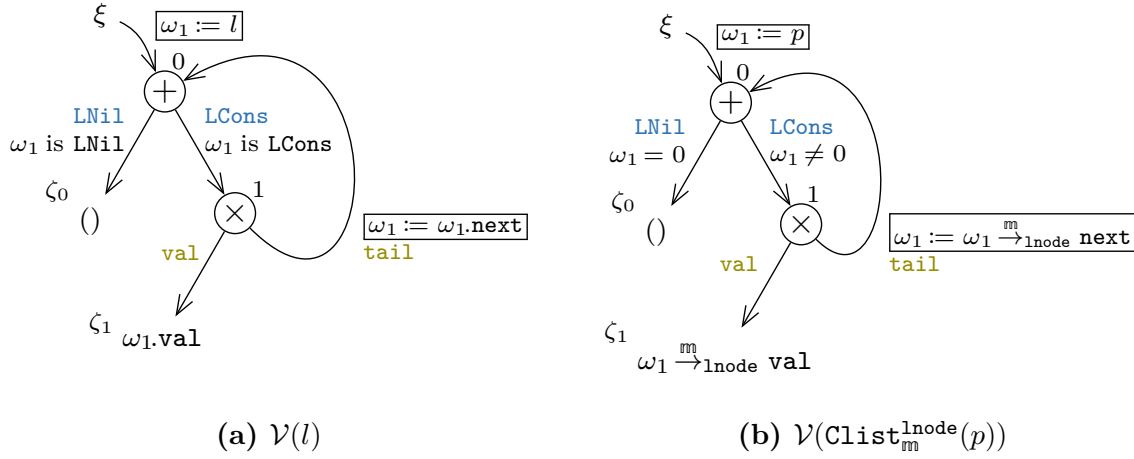


Figure 4.13: Value trees for a `List` variable l and a lifted expression $Clist_m^{lnode}(p)$ respectively.

Canonical Deconstruction Property

A path originating at the entry node ξ and terminating at an exit node ζ is called an *execution path* and is denoted by ep . Generalizing the edge syntax, $[v \rightarrow label_1, label_2, \dots, label_n]$ represents the path starting at v such that its n consecutive edges are associated with the labels $label_i \forall i \in [1, n]$. Hence, an execution path ep can be written as $\xi \rightarrow [v_0 \rightarrow label_1, label_2, \dots, label_n]$, or $ep[label_1, label_2, \dots, label_n]$ in short. Recall that, edges leaving a \times node are associated with field names. The *field trace of ep* , $Ftrace(ep)$ is defined as the ordered list of field names associated with edges in the path ep . For an execution path $ep[LCons, tail, LCons, val]$ in the value tree shown in fig. 4.13a, $Ftrace(ep)$ is given by $[tail, val]$. Given a value tree $\mathcal{V}(e)$ and an execution path ep with a field trace $[a_1, a_2, \dots, a_n]$, $e.a_1.a_2 \dots a_n$ is defined as the component of e with respect to ep , denoted by $Comp_{ep}(e)$. The *canonical deconstruction property* for a value tree ensures that:

1. The condition under which $Comp_{ep}(e)$ is accessible (i.e. is well-formed as discussed in section 2.2.1) is equal to the path condition of ep , denoted by $Pathcond(ep)$.

2. The value of $\text{Comp}_{\text{ep}}(e)$ is equal to the value returned by $\mathcal{V}(e)$ (interpreted as a program) along the execution path ep . This is equivalent to the weakest-precondition of the value returned along the path ep and is denoted by $\text{Retval}(\text{ep})$.
3. Both $\text{Pathcond}(\text{ep})$ and $\text{Retval}(\text{ep})$ are in the canonical form (section 4.4.2).

Consider the value tree corresponding to the lifted expression $e = \text{Clist}_{\text{m}}^{\text{lnode}}(p)$ as shown in fig. 4.13b. The execution path $\text{ep}[\text{LCons}, \text{val}]$ corresponds to the component $\text{Comp}_{\text{ep}}(e) = \text{Clist}_{\text{m}}^{\text{lnode}}(p).\text{val}$. The path condition and the value returned along ep are given by $\text{Pathcond}(\text{ep}) = (p \neq 0)$ and $\text{Retval}(\text{ep}) = p \xrightarrow{\text{m}}_{\text{lnode}} \text{val}$ respectively. Note that $\text{Comp}_{\text{ep}}(e)$ is well-formed iff $\text{Pathcond}(\text{ep})$ is true and $\text{Comp}_{\text{ep}}(e) = \text{Retval}(\text{ep})$. Furthermore, both are in canonical forms.

Reduction of Approximate Recursive Relations

Since an ADT represents a ‘sum of product’ type, each level of an ADT value (in its expression tree as shown in section 3.5.1) corresponds to two levels – \oplus and \otimes in the value tree representation. Combining the above observation with the value tree properties discussed above (section 4.4.11) allow us reformulate approximate recursive relations between l_1 and l_2 as *bounded* equivalence [14] between $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$. The d -depth over-approximation $l_1 \sim_d l_2$ asserts that $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ are equivalent for all execution paths up to a depth⁴ of $2 \cdot d$. Clearly, this is a weaker condition because $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ may behave differently for longer execution paths. The d -depth under-approximation $l_1 \approx_d l_2$ asserts that $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ are equivalent for all execution paths up to a depth of $2 \cdot d$ and all other execution paths are unreachable. This is a stronger condition than general equivalence because paths deeper than $2 \cdot d$ may indeed be reachable. Let $\text{EP}_\delta(l_1, l_2)$ be the set of all execution path pairs $\langle \text{ep}_1, \text{ep}_2 \rangle$ such that $\text{Ftrace}(\text{ep}_1) = \text{Ftrace}(\text{ep}_2)$ and depth of ep_1 and ep_2 is δ . For example, consider the value trees shown in fig. 4.13 corresponding to the **List** values $e_1 = l$ and $e_2 = \text{Clist}_{\text{m}}^{\text{clnode}}(p)$ respectively. Then, $\text{EP}_0(e_1, e_2) = \emptyset$, $\text{EP}_1(e_1, e_2) = \{ \langle \xi \rightarrow 0 \rightarrow \zeta_0, \xi \rightarrow 0 \rightarrow \zeta_0 \rangle \}$ and $\text{EP}_2(e_1, e_2) = \{ \langle \xi \rightarrow 0 \rightarrow 1 \rightarrow \zeta_1, \xi \rightarrow 0 \rightarrow 1 \rightarrow \zeta_1 \rangle \}$. Finally, the d -depth approximations of $l_1 \sim l_2$ are given by:

⁴The depth of an execution path ep is defined as the depth of the exit node in the unrolled (executable) value tree. If ep contains n edges (including the entry edge), then its depth is equal to $(n - 1)$.

$$\begin{aligned}
l_1 \sim_d l_2 &= \sum_{\delta=0}^{2 \cdot d} \left(\bigwedge_{\substack{\langle \text{ep}_1, \text{ep}_2 \rangle \\ \in \text{EP}_\delta(l_1, l_2)}} \begin{array}{l} \text{Pathcond}(\text{ep}_1) = \text{Pathcond}(\text{ep}) \\ \text{RetVal}(\text{ep}_1) = \text{RetVal}(\text{ep}_2) \end{array} \right) \\
l_1 \approx_d l_2 &= \sum_{\delta=0}^{2 \cdot d} \left(\bigwedge_{\substack{\langle \text{ep}_1, \text{ep}_2 \rangle \\ \in \text{EP}_\delta(l_1, l_2)}} \begin{array}{l} \text{Pathcond}(\text{ep}_1) = \text{Pathcond}(\text{ep}) \\ \text{RetVal}(\text{ep}_1) = \text{RetVal}(\text{ep}_2) \end{array} \right) \wedge \left(\bigwedge_{\substack{\langle \text{ep}_1, \text{ep}_2 \rangle \\ \in \text{EP}_{2 \cdot d + 1}(l_1, l_2)}} \begin{array}{l} \neg \text{Pathcond}(\text{ep}_1) \\ \neg \text{Pathcond}(\text{ep}_2) \end{array} \right)
\end{aligned} \tag{4.3}$$

Recall that the canonical deconstruction property ensures that the reductions in eq. (4.3) are in the canonical form. The value trees $\mathcal{V}(l_1 \sim_d l_2)$ and $\mathcal{V}(l_1 \approx_d l_2)$ contains a single (leaf) node with the boolean expressions given in eq. (4.3). The conversion algorithm presented in section 4.4.10 together with the handling of approximate recursive relations described above enables the reduction of a proof obligation P without recursive relations directly to its canonical form by – (a) constructing $\mathcal{V}(P)$ and (b) extracting the boolean expression at its root.

Bisimilarity of Value Trees

Unlike its approximations, a recursive relation $l_1 \sim l_2$ asserts complete equivalence between $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ respectively. Similar to our top-level equivalence check between \mathcal{S} and \mathcal{C} , we attempt to prove that $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ are bisimilar. To make the search for a bisimulation relation easier, we peel $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ to unify their static structures. Once unified, the bisimulation relation requires the inference of inductive invariants at correlated nodes such that under the precondition at $(\xi:\xi)$, inductive invariants hold and both $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ have equal observables at correlated exit nodes $(\zeta_i:\zeta_i)$.

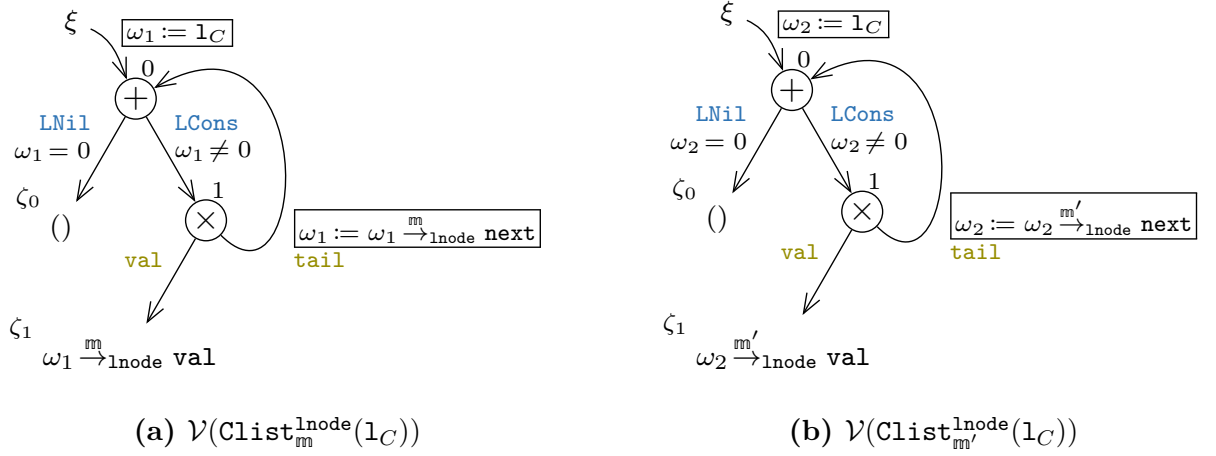


Figure 4.14: Value trees of $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}(1_C)$ and $\text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(1_C)$ along with the invariants table.

Recall the type III proof obligation illustrated in section 3.6:

$$\text{LHS} \Rightarrow \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(1_C) \sim \text{Clist}_{\mathfrak{m}'}^{\text{lnode}}(1_C) \quad (4.4)$$

Also recall the points-to invariants available at C5 (showing only the relevant ones): $p_C \rightsquigarrow \{C4_1\}$, $1_C \rightsquigarrow \{C4_{2+}\}$, $C4_1 \rightsquigarrow \{C4_{2+}\}$, $C4_{2+} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$, and $\mathcal{H} \rightsquigarrow \{C4_{2+}, \mathcal{H}\}$. Similar to a deconstruction program, we run our points-to analysis on the value trees to identify potentially beneficial

points-to invariants at all correlated nodes. Figure 4.14 shows the value trees of $\text{Clist}_{\mathbf{m}}^{\text{lnode}}(1_C)$ and $\text{Clist}_{\mathbf{m}'}^{\text{lnode}}(1_C)$ along with the table of invariants required for a successful bisimulation check.

Chapter 5

Evaluation

We have implemented S2C on top of the Counter tool [27]. We use *four* SMT solvers running in parallel for solving SMT proof obligations discharged by our proof discharge algorithm: **z3-4.8.7**, **z3-4.8.14** [22], **Yices2-45e38fc** [23], and **cvc4-1.7** [1]. An unroll factor of *four* is used to handle loop unrolling in the C implementation. We use a default value of *eight* for over- and under-approximation depths (d_o and d_u). The default values for unrolling parameter k (used for categorization of proof obligations) and η (used by **strongestInvCover()** during weakening of recursive relation invariants) are *five*.

S2C requires the user to provide a Spec program S (specification), a C implementation C , and a file that contains their input-output specifications. For each function pair, S2C attempts to find equivalence between their CFGs \mathcal{S} and \mathcal{C} under their respective Pre and $Post$ (given as part of input-output specification). An equivalence check requires the identification of lifting constructors to relate \mathcal{C} values to the ADT values in \mathcal{S} through recursive relations. Such relations may be required at the entry of both programs (i.e. in the precondition Pre), in the middle of both programs (i.e. as invariants at intermediate product-CFG nodes), and at the exit of both programs (i.e. in the postcondition $Post$). Pre and $Post$ are user-specified, whereas the inductive invariants are inferred automatically by our algorithm. During invariant inference, S2C derives the candidate lifting constructors from the user-specified Pre and $Post$. More sophisticated

approaches to finding lifting constructors are left as future work.

5.1 Experiments

We consider programs involving four distinct ADTs, namely, (T1) **Str**, (T2) **List**, (T3) **Tree** and (T4) **Matrix**. For each Spec program specification, we consider multiple C implementations that differ in their (a) memory layout of ADTs, and (b) algorithmic strategies. For example, a **Matrix** in C may be laid out in a two-dimensional array, a one-dimensional array using row or column major layouts etc. On the other hand, an optimized implementation may choose manual vectorization of an inner-most loop. Next, we consider each ADT in more detail. For each, we discuss (a) its corresponding programs, (b) C memory layouts and their lifting constructors, and (c) varying algorithmic strategies.

Table 5.1: String lifting constructors and their definitions.

Lifting Constructor	Definition
(T1) Str = SInvalid SNil SCons(ch:i8, tail:Str)	OptStr = NotFound Found(str:Str)
$\text{Cstr}_m^{\text{u8}[]} (p:i32)$	<pre> if p = 0_{i32} then SInvalid elif p[0_{i32}]_mⁱ⁸ = 0_{i8} then SNil else SCons(p[0_{i32}]_mⁱ⁸, Cstr_m^{u8}[(p + 1_{i32})]) </pre>
$\text{Coptstr}_m^{\text{u8}[]} (p:i32)$	<pre> if p = 0_{i32} then NotFound else Found(Cstr_m^{u8}[(p)]) </pre>
$\text{Cstr}_m^{\text{lnode}(\text{u8})} (p:i32)$	<pre> if p = 0_{i32} then SInvalid elif p $\xrightarrow{\text{m}}$ lnode val = 0_{i8} then SNil else SCons(p $\xrightarrow{\text{m}}$ lnode val, Cstr_m^{lnode(u8)}(p $\xrightarrow{\text{m}}$ lnode next)) </pre>
$\text{Coptstr}_m^{\text{lnode}(\text{u8})} (p:i32)$	<pre> if p = 0_{i32} then NotFound else Found(Cstr_m^{lnode(u8)}(p)) </pre>
$\text{Cstr}_m^{\text{clnode}(\text{u8})} (p:i32, i:i2)$	<pre> if p = 0_{i32} then SInvalid elif p $\xrightarrow{\text{m}}$ lnode chunk[i]_mⁱ⁸ = 0_{i8} then SNil else SCons(p $\xrightarrow{\text{m}}$ lnode chunk[i]_mⁱ⁸, Cstr_m^{clnode(u8)}(i = 3_{i2}?p $\xrightarrow{\text{m}}$ clnode next : p, i + 1_{i2})) </pre>
$\text{Coptstr}_m^{\text{clnode}(\text{u8})} (p:i32, i:i2)$	<pre> if p = 0_{i32} then NotFound else Found(Cstr_m^{clnode(u8)}(p, i)) </pre>

5.1.1 String

We wrote a single specification in Spec for each of the following common string library functions: `strlen`, `strchr`, `strcmp`, `strspn`, `strcspn`, and `strpbrk`. For each specification program, we took multiple C implementations of that program, drawn from popular libraries like `glibc` [3], `klibc` [4], `newlib` [7], `openbsd` [8], `uClibc` [9], `dietlibc` [2], `musl` [5], and `netbsd` [6]. These library implementations use a nul-terminated array to represent a string, and the corresponding lifting constructor is $\text{Cstr}_m^{\text{u8}\square}$. $\text{u}\langle N \rangle$ represents the N-bit unsigned integer type in C. For example, `u8` represents `unsigned char` type.

Further, we implemented custom C programs for these functions that uses linked list and *chunked linked list* data structures to represent a string. In a chunked linked list, a single list node (linked through a `next` pointer) contains a small array (chunk) of values. We use a default chunk size of four for our benchmarks. The corresponding lifting constructors are $\text{Cstr}_m^{\text{lnode}(\text{u8})}$ and $\text{Cstr}_m^{\text{cnode}(\text{u8})}$ respectively. These lifting constructors are defined in table 5.1. $\text{Cstr}_m^{\text{lnode}(\text{u8})}$ requires a single argument p representing the pointer to the list node. On the other hand, $\text{Cstr}_m^{\text{cnode}(\text{u8})}$ requires two arguments p and i , where p represents the pointer to the chunked linked list node and i represents the position of the initial character in the chunk.

An Example : `strchr`

Additionally, we define an optional string type `OptStr` to specify behaviour of functions that conditionally return a string (e.g. `strchr`, `strpbrk`). The `OptStr` ADT along with three (pairs of) lifting constructors for the three layouts of the `Str` ADT are shown in table 5.1. `strchr` accepts a string t and a character c^1 and returns the longest suffix of t that begins with c , otherwise it returns the null pointer to indicate failure to find c in the string t . In case c is the null character (i.e. `nul`), `strchr` is defined to return the empty string (instead of the null pointer). Figures 5.1a and 5.1b show the IRs of the `strchr` specification and a generic C implementation respectively. We demonstrate two important aspects of S2C using this example – (a) use of (\mathcal{S} def) and *Pre*

¹Due to historical reasons, the type of c is declared as `int` to maintain backward compatibility with pre C-98 code. However, the function is specified to cast it to a character and use it instead.

```

S0: OptStr strchr (Str s, i8 c) {
S1:   while true:
S2:     assume ¬(s is SInvalid);
S3:     if s is SNil:
S4:       if c = 0i8: return Found(s);
S5:       return NotFound();
S6:     i8 ch := s.ch; // (s is SCons)
S7:     if c = ch: return Found(s);
S8:     s := s.tail;
SE: }

```

(a) Strchr specification

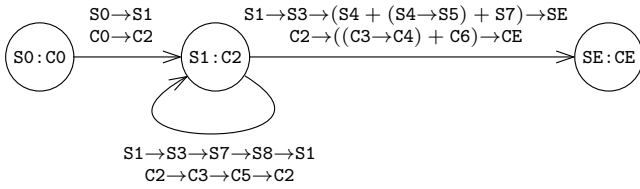
```

char* strchr(char* t, int c);

C0: i32 strchr (i32 t, i32 c) {
C1:   i8 ch := c[7:0];
C2:   while t[0i32]mi8 ≠ ch:
C3:     if t[0i32]mi8 = 0i8:
C4:       return 0i32;
C5:     t := t + 1i32
C6:   return t;
CE: }

```

(b) Strchr implementation



PC-Pair	Invariants
(S0:C0)	$\textcircled{P1} \ s_S \sim \text{Cstr}_m^{\text{u8}}(t_C)$ $\textcircled{P2} \ c_S = c_C[7:0]$
(S1:C2)	$\textcircled{I1} \ s_S \sim \text{Cstr}_m^{\text{u8}}(t_C)$ $\textcircled{I2} \ c_S = \text{ch}_C$
(SE:CE)	$\textcircled{E} \ \text{ret}_S \sim \text{Coptstr}_m^{\text{u8}}(\text{ret}_C)$

(c) Product-CFG for programs figs. 5.1a and 5.1b (d) Node invariants for product-CFG in fig. 5.1c

Figure 5.1: Figures 5.1a and 5.1b show the (abstracted) IRs for the Spec specification and a generic nul-terminated array based C implementation of `strchr`. Figure 5.1c shows the product-CFG representing a bisimulation relation between figs. 5.1a and 5.1b. The node invariants for the product-CFG in fig. 5.1c are given in fig. 5.1d.

to constrain the C implementation to only well-formed inputs (in section 2.3.1), and (b) need for correlating pathsets (instead of paths) (in section 4.2.1).

Recall that a nul-terminated C string is only well-formed if the string itself does not belong to a region of memory containing the null pointer. This well-formedness condition is necessary to prove that the pointer to the string returned at C6 (in fig. 5.1b) is non-null (used uniquely to indicate a failure to find the character c in the string t). As previously discussed in section 2.3, we expose this well-formedness condition in the specification using the explicit **Str** data constructor **SInvalid**. Finally, we assert that s_S in fig. 5.1a is well-formed using the **assuming-do** statement (**S3** in fig. 5.1a) and relate the non-null well-formedness condition of the C input string t_C with the condition of s_S being **SInvalid** using *Pre* (labeled $\textcircled{P1}$ in fig. 5.1d). Note the use of Coptstr_m^{u8} in the postcondition (labeled \textcircled{E} in fig. 5.1d).

Figure 5.1c shows the product-CFG showing the path correlations between \mathcal{S} and \mathcal{C} . Consider the product-CFG edge $(S1:C2) \rightarrow (SE:CE)$ correlating the pathsets: $S1 \rightarrow S3 \rightarrow (S4 + (S4 \rightarrow S5) + S7) \rightarrow SE$ (in \mathcal{S}) and $C2 \rightarrow ((C3 \rightarrow C4) + C6) \rightarrow CE$ (in \mathcal{C}). The \rightarrow and $+$ operators are used to represent ‘series’ and ‘parallel’ path combinations. The above two pathsets represent the following two sets $\{S1 \rightarrow S3 \rightarrow S4 \rightarrow SE, S1 \rightarrow S3 \rightarrow S4 \rightarrow S5 \rightarrow SE, S1 \rightarrow S3 \rightarrow S7 \rightarrow SE\}$ and $\{C2 \rightarrow C3 \rightarrow C4 \rightarrow CE, C2 \rightarrow C6 \rightarrow CE\}$ respectively. In \mathcal{S} , the case of c_S being nul is handled explicitly at **S4**, whereas **S7** handles the case of s_S containing the (non-nul) character c_S . Interestingly in \mathcal{C} , both these cases are taken care of by the singular exit edge outgoing at **C6**. For a successful bisimulation proof, we are required to correlate the \mathcal{C} path $C2 \rightarrow C6 \rightarrow CE$ with the \mathcal{S} pathset $\{S1 \rightarrow S3 \rightarrow S4 \rightarrow SE, S1 \rightarrow S3 \rightarrow S7 \rightarrow SE\}$. Such situations are rather frequent because the strongly-typed specification is forced to handle each case explicitly while a C implementation may take advantage of the underlying representation to generalize multiple explicit cases into one.

Another Example : strlen

Figure 5.2 shows the **strlen** specification and two vastly different C implementations. Figure 5.2b is a generic implementation using a C-style nul-terminated array to represent a string. The second implementation in fig. 5.2c differs from fig. 5.2b in the following: (a) it uses a chunked linked list

```

S0: i32 strlen (Str s) {
S1:   i32 len := 0i32;
S2:   while ¬(s is SNil):
S3:     assume ¬(s is SInvalid);
S4:     // (s is SCons)
S5:     s := s.tail;
S6:     len := len + 1i32;
S7:   return len;
SE: }

```

(a) Strlen specification

```

size_t strlen(char* s);

C0: i32 strlen (i32 s) {
C1:   i32 i := 0i32;
C2:   while s[0i32]mi8 ≠ 0i8:
C3:     s := s + 1i32;
C4:     i := i + 1i32;
C5:   return i;
CE: }

```

(b) Generic strlen implementation using array

```

typedef struct cnode {
  char chunk[4]; struct cnode* next; } cnode;
size_t strlen(cnode* cl);

C0 : i32 strlen (i32 cl) {
C1 :   i32 hi := 0x80808080i32; i32 lo := 0x01010101i32;
C2 :   i32 i := 0i32;
C3 :   while true:
C4 :     i32 dword_ptr := addrof(cl  $\xrightarrow{m}$  cnode chunk);
C5 :     i32 dword := dword_ptr[0i32]mi32;
C6 :     if ((dword - lo) & (~dword) & hi) ≠ 0i32:
C7 :       if dword_ptr[0i32]mi8 = 0i8: return i;
C8 :       if dword_ptr[1i32]mi8 = 0i8: return i + 1i32;
C9 :       if dword_ptr[2i32]mi8 = 0i8: return i + 2i32;
C10:      if dword_ptr[3i32]mi8 = 0i8: return i + 3i32;
C11:      cl := cl  $\xrightarrow{m}$  cnode next; i := i + 4i32;
CE : }

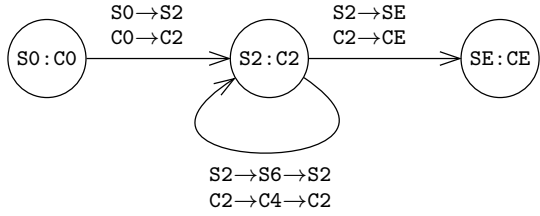
```

(c) Optimized strlen implementation using chunked linked list

Figure 5.2: Figure 5.2a shows the (abstracted) IR for the Spec specification of **strlen**.

Figures 5.2b and 5.2c show the (abstracted) IRs for two C implementations of **strlen**.

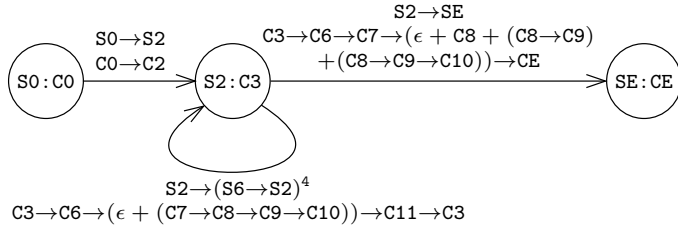
Figure 5.2b is a generic implementation using a nul-terminated array to represent a string, whereas fig. 5.2c is an optimized implementation with a chunked linked list memory layout for a string.



(a) Product-CFG for programs figs. 5.2a and 5.2b

PC-Pair	Invariants
(S0:C0)	$\textcircled{P} \ s_S \sim \text{Cstr}_m^{\text{char}}(s_C)$
(S2:C2)	$\textcircled{I1} \ s_S \sim \text{Cstr}_m^{\text{char}}(s_C)$ $\textcircled{I2} \ \text{len}_S = i_C$
(SE:CE)	$\textcircled{E} \ \text{ret}_S = \text{ret}_C$

(b) Invariants for product-CFG in fig. 5.3a



(c) Product-CFG for programs figs. 5.2a and 5.2c

PC-Pair	Invariants
(S0:C0)	$\textcircled{P} \ s_S \sim \text{Cstr}_m^{\text{cnode}}(cl_C, 0)$
(S2:C3)	$\textcircled{I1} \ s_S \sim \text{Cstr}_m^{\text{cnode}}(cl_C, 0)$ $\textcircled{I2} \ \text{len}_S = i_C$
(SE:CE)	$\textcircled{E} \ \text{ret}_S = \text{ret}_C$

(d) Invariants for product-CFG in fig. 5.3c

Figure 5.3: Product-CFGs and their node invariants representing bisimulation relations between the specification fig. 5.2a and its two implementations in figs. 5.2b and 5.2c respectively.

data layout for the input string and (b) it uses specialized bit manipulations to identify whether a (4 byte) chunk contains the nul character simultaneously. S2C is able to automatically find a bisimulation relation for both implementations against the unaltered specification. Figure 5.3 shows the product-CFG and its associated node invariants for each implementation.

Lifting constructors are named based on the C data layout being lifted and the type of the lifted value. For example, $\text{Cstr}^{\text{u8}}[]$ represents a Str lifting constructor for an array layout. In general, we use the following naming convention for different C data layouts: $\text{T}[]$ represents an array of type T (e.g., $\text{u8}[]$). $\text{lnode}(\text{T})$ represents a linked list node type containing a value of type T . Similarly, $\text{clnode}(\text{T})$ and $\text{tnode}(\text{T})$ represent a chunked linked list and a tree node with values of type T respectively.

Table 5.2: List lifting constructors and their definitions.

Lifting Constructor	Definition
	$(\text{T2}) \text{ List} = \text{LNil} \mid \text{LCons}(\text{val}:\text{i32}, \text{tail}:\text{List})$
$\text{Clist}_m^{\text{u32}}(p \ i \ n:\text{i32})$	$\text{if } i \geq_u n \ \text{then } \text{LNil}$ $\text{else } \text{LCons}(p[i]_m^{\text{i32}}, \text{Clist}_m^{\text{u32}}(p, i + 1_{\text{i32}}, n))$
$\text{Clist}_m^{\text{lnode}(\text{u32})}(p:\text{i32})$	$\text{if } p = 0_{\text{i32}} \ \text{then } \text{LNil}$ $\text{else } \text{LCons}(p \xrightarrow{m}_{\text{lnode}} \text{val}, \text{Clist}_m^{\text{lnode}}(p \xrightarrow{m}_{\text{lnode}} \text{next}))$
$\text{Clist}_m^{\text{clnode}(\text{u32})}(p:\text{i32}, i:\text{i2})$	$\text{if } p = 0_{\text{i32}} \ \text{then } \text{LNil}$ $\text{else } \text{LCons}(p \xrightarrow{m}_{\text{clnode}} \text{chunk}[i]_m^{\text{i32}}, \text{Clist}_m^{\text{clnode}}(i = 3_{\text{i2}}?p \xrightarrow{m}_{\text{clnode}} \text{next} : p, i + 1_{\text{i2}}))$

5.1.2 List

We wrote a Spec program specification that creates a list, a program that traverses a list to compute the sum of its elements and a program that computes the dot product of two lists. We use three different data layouts for a list in C: array ($\text{Clist}_m^{\text{u32}}[]$), linked list ($\text{Clist}_m^{\text{lnode}(\text{u32})}$), and a chunked linked list ($\text{Clist}_m^{\text{clnode}(\text{u32})}$). The lifting constructors are shown in table 5.2. Although similar to the string lifting constructors, these lifting constructors differ widely in their data encoding. For example, $\text{Clist}_m^{\text{u32}}(p, i, n)$ represents a List value constructed from a C array of

size n , pointed to by p and starting at the i^{th} index. The list becomes empty when we are at the end of the array. $\text{Clist}_{\text{m}}^{\text{lnode}(\text{u32})}$ and $\text{Clist}_{\text{m}}^{\text{clnode}(\text{u32})}$, on the other hand, encodes the empty list (i.e. LNil) using the null pointer. These layouts are in contrast to the Str layouts, all of which uses the nul character to indicate the empty string.

Table 5.3: Tree lifting constructors and their definitions.

Lifting Constructor	Definition
$\textcircled{\text{T3}}$ Tree = TNil TCons(val:i32, left:Tree, right:Tree)	
$\text{Ctree}_{\text{m}}^{\text{u32}[]} (p \ i \ n : i32)$	$\text{if } i \geq_u n \ \text{then TNil}$ $\text{else TCons}(p[i]_{\text{m}}^{\text{i32}}, \text{Ctree}_{\text{m}}^{\text{u32}[]} (p, 2_{\text{i32}} \times i + 1_{\text{i32}}, n), \text{Ctree}_{\text{m}}^{\text{u32}[]} (p, 2_{\text{i32}} \times i + 2_{\text{i32}}, n))$
$\text{Ctree}_{\text{m}}^{\text{tnode}(\text{u32})} (p : i32)$	$\text{if } p = 0_{\text{i32}} \ \text{then TNil}$ $\text{else TCons}(p \xrightarrow{\text{m}}_{\text{tnode}} \text{val}, \text{Ctree}_{\text{m}}^{\text{tnode}(\text{u32})} (p \xrightarrow{\text{m}}_{\text{tnode}} \text{left}), \text{Ctree}_{\text{m}}^{\text{tnode}(\text{u32})} (p \xrightarrow{\text{m}}_{\text{tnode}} \text{right}))$

5.1.3 Tree

We wrote a Spec program that computes the sum of the elements of a binary tree through an inorder traversal using recursion. We use two different data layouts for a tree: (a) a flat array where a *complete* binary tree is laid out in breadth-first search order commonly used for heaps ($\text{Ctree}_{\text{m}}^{\text{u32}[]}$), and (b) a linked tree node with two pointers for the left and right children ($\text{Ctree}_{\text{m}}^{\text{tnode}(\text{u32})}$) (shown in table 5.3). Both Spec and C programs contain non-tail recursive procedure calls for left and right children. S2C is able to correlate these recursive calls using user-provided *Pre* and *Post* as discussed in section 4.2.3. At the entry of the recursive calls, S2C is required to prove that *Pre* holds for the arguments and at the exit of the recursive calls, S2C assumes *Post* on the values returned.

Table 5.4: Matrix and auxiliary List lifting constructors and their definitions.

Lifting Constructor	Definition
(T4) Matrix = MNil MCons(row:List, cols:Matrix)	
$\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\square]}(p \ i \ u \ v : \text{i32})$	<u>if</u> $i \geq_u u$ <u>then</u> MNil <u>else</u> MCons($\text{Clist}_{\mathfrak{m}}^{\text{u32}[\square]}(p[i]_{\mathfrak{m}}^{\text{i32}}, 0_{\text{i32}}, v)$, $\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\square]}(p, i + 1_{\text{i32}}, u, v)$)
$\text{Clist}_{\mathfrak{m}}^{\text{u32}[\text{r}]}(p \ i \ j \ u \ v : \text{i32})$	<u>if</u> $j \geq_u v$ <u>then</u> LNil <u>else</u> LCons($p[i \times v + j]_{\mathfrak{m}}^{\text{i32}}$, $\text{Clist}_{\mathfrak{m}}^{\text{u32}[\text{r}]}(p, i, j + 1_{\text{i32}}, u, v)$)
$\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\text{r}]}(p \ i \ u \ v : \text{i32})$	<u>if</u> $i \geq_u u$ <u>then</u> MNil <u>else</u> MCons($\text{Clist}_{\mathfrak{m}}^{\text{u32}[\text{r}]}(p, i, 0_{\text{i32}}, u, v)$, $\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\text{r}]}(p, i + 1_{\text{i32}}, u, v)$)
$\text{Clist}_{\mathfrak{m}}^{\text{u32}[\text{c}]}(p \ i \ j \ u \ v : \text{i32})$	<u>if</u> $j \geq_u v$ <u>then</u> LNil <u>else</u> LCons($p[i + j \times u]_{\mathfrak{m}}^{\text{i32}}$, $\text{Clist}_{\mathfrak{m}}^{\text{u32}[\text{c}]}(p, i, j + 1_{\text{i32}}, u, v)$)
$\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\text{c}]}(p \ i \ u \ v : \text{i32})$	<u>if</u> $i \geq_u u$ <u>then</u> MNil <u>else</u> MCons($\text{Clist}_{\mathfrak{m}}^{\text{u32}[\text{c}]}(p, i, 0_{\text{i32}}, u, v)$, $\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\text{c}]}(p, i + 1_{\text{i32}}, u, v)$)
$\text{Cmat}_{\mathfrak{m}}^{\text{lnode}(\text{u32}[\square])}(p \ v : \text{i32})$	<u>if</u> $p = 0_{\text{i32}}$ <u>then</u> MNil <u>else</u> MCons($\text{Clist}_{\mathfrak{m}}^{\text{u32}[\square]}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}, 0_{\text{i32}}, v)$, $\text{Cmat}_{\mathfrak{m}}^{\text{lnode}(\text{u32}[\square])}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next}, v)$)
$\text{Cmat}_{\mathfrak{m}}^{\text{lnode}(\text{u32})[\square]}(p \ i \ u : \text{i32})$	<u>if</u> $i \geq_u u$ <u>then</u> MNil <u>else</u> MCons($\text{Clist}_{\mathfrak{m}}^{\text{lnode}(\text{u32})}(p[i]_{\mathfrak{m}}^{\text{i32}})$, $\text{Cmat}_{\mathfrak{m}}^{\text{lnode}(\text{u32})[\square]}(p, i + 1_{\text{i32}}, u)$)
$\text{Cmat}_{\mathfrak{m}}^{\text{clnode}(\text{u32})}(p \ i \ u : \text{i32})$	<u>if</u> $i \geq_u u$ <u>then</u> MNil <u>else</u> MCons($\text{Clist}_{\mathfrak{m}}^{\text{clnode}(\text{u32})}(p[i]_{\mathfrak{m}}^{\text{i32}}, 0_{\text{i2}})$, $\text{Cmat}_{\mathfrak{m}}^{\text{clnode}(\text{u32})[\square]}(p, i + 1_{\text{i32}}, u)$)

5.1.4 Matrix

We wrote a Spec program to count the frequency of a value appearing in a 2D matrix. A matrix is represented as an ADT that resembles a List of Lists ((T4) in table 5.4). The C implementations for a Matrix object include (a) a two-dimensional array ($\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\square]}$), (b) a flattened row-major array ($\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\text{r}]}$), (c) a flattened column-major array ($\text{Cmat}_{\mathfrak{m}}^{\text{u32}[\text{c}]}$), (d) a linked list of 1D arrays ($\text{Cmat}_{\mathfrak{m}}^{\text{lnode}(\text{u32}[\square])}$), (e) a 1D array of linked lists ($\text{Cmat}_{\mathfrak{m}}^{\text{lnode}(\text{u32})[\square]}$) and (f) a 1D array of chunked linked list ($\text{Cmat}_{\mathfrak{m}}^{\text{clnode}(\text{u32})}$) memory layouts. Note that both T[r] and T[c] represent a 1D array of type T; *r* and *c* emphasizes that these arrays are used to represent matrices in row-major and column-major encodings respectively. We also introduce two auxiliary lifting constructors

$\text{Clist}_m^{u32[r]}$ and $\text{Clist}_m^{u32[c]}$ for lifting each row of matrices lifted using the corresponding $\text{Cmat}_m^{u32[r]}$ and $\text{Cmat}_m^{u32[c]}$ *Matrix* lifting constructors. These lifting constructors are listed in table 5.4.

5.2 Results

Table 5.5 lists the various C implementations and the time S2C took to compute equivalence with their specifications. For functions that take two or more data structures as arguments, we show results for different combinations of data layouts for each argument. We also show the minimum under-approximation (d_u) and over-approximation (d_o) depths at which the equivalence proof completed (keeping all other parameters to their default values).

5.3 Limitations

S2C is not without limitations. Since S2C is only interested in finding a bisimulation relation, a whole class of non-bisimilar but equivalent program pairs is beyond our scope. In addition to recursive relations based on the lifting constructors provided as part of *Pre* and *Post*, S2C currently only supports bitvector affine and inequality relations. Consequently, non-linear bitvector invariants (such as polynomial invariants) are not supported. More importantly, S2C does not attempt to infer lifting constructors and merely uses the lifting constructors (with different arguments) provided as part of the input-output characteristics. While our correlation and invariant inference algorithms are based on the Counter tool [27], which is designed primarily for equivalence checking between (C-like) unoptimized IR and assembly, we found them to be quite effective for equivalence checking between Spec and deterministic C as well. However, S2C inherits many of the limitations of the Counter tool.

For example, S2C fails to find a proof of equivalence if the unrolling in the C implementation is higher than the unrolling parameter μ used during path enumeration as part of the product-CFG construction algorithm. Larger values of μ would significantly increase the correlation search space and likely have negative implications on the runtime of the algorithm. Additionally,

Table 5.5: Time taken by S2C for successful equivalence checks between specifications and their C implementations. Additionally, we show the minimum under- and over-approximation depths at which these equivalence checks succeed.

Data Layout	Variant	Time(s)	(d_u, d_o)	Data Layout	Variant	Time(s)	(d_u, d_o)
u32[]	list			u32[]	tree		
	sum naive	16	(1,2)		sum	264	(1,2)
	sum opt	49	(4,5)		sum	204	(1,2)
	dot naive	65	(1,2)		matfreq		
lnode(u32)	dot opt	176	(4,5)	u8[]	naive	974	(1,3)
	sum naive	8	(1,2)		opt	1.8k	(4,8)
	sum opt	54	(4,5)		naive	958	(1,3)
	dot naive	37	(1,2)		opt	1.9k	(4,8)
cnode(u32)	dot opt	120	(4,5)	u8[r]	naive	984	(1,3)
	construct	426	(1,1)		opt	1.9k	(4,6)
	sum opt	39	(4,5)		naive	753	(1,3)
	dot opt	118	(4,5)		opt	1.7k	(4,6)
u8[]	strlen			lnode(u8[])	naive	1.5k	(1,2)
	dietlibc _s	9	(1,2)		opt	2.3k	(4,6)
	dietlibc _f	44	(3,2)		opt	1.8k	(4,6)
	glibc	52	(3,2)		strpbrc		
lnode(u8)	klibc	9	(1,2)	u8[], u8[]	dietlibc	398	(1,2)
	musl	49	(3,2)		opt	494	(4,2)
	netbsd	9	(1,2)		naive	392	(1,2)
	newlib	50	(3,2)		opt	540	(4,2)
cnode(u8)	openbsd	8	(1,2)	u8[], cnode(u8)	opt	523	(4,2)
	uClibc	8	(1,2)		naive	497	(1,2)
	naive	13	(1,2)		opt	602	(4,2)
	opt	49	(3,5)		naive	345	(1,2)
u8[]	opt	45	(3,5)	lnode(u8), lnode(u8)	opt	503	(4,2)
	strchr				opt	572	(4,2)
	dietlibc _s	16	(1,1)		strcsn		
	dietlibc _f	89	(4,1)		dietlibc	462	(1,2)
lnode(u8)	glibc	127	(4,1)	u8[], u8[]	opt	538	(4,2)
	klibc	23	(1,1)		naive	395	(1,2)
	newlib _s	15	(1,1)		opt	521	(4,2)
	openbsd	24	(1,1)		opt	527	(4,2)
u8[], u8[]	uClibc	22	(1,1)	u8[], cnode(u8)	naive	601	(1,2)
	naive	19	(1,1)		opt	660	(4,2)
	opt	146	(4,1)		naive	349	(1,2)
	strcmp				opt	502	(4,2)
lnode(u8), lnode(u8)	dietlibc _s	39	(1,1)	lnode(u8), cnode(u8)	opt	595	(4,2)
	freebsd	39	(1,1)		strspn		
	glibc	41	(1,1)		dietlibc	277	(1,2)
	klibc	41	(1,1)		opt	388	(4,2)
cnode(u8), cnode(u8)	musl	41	(1,1)	u8[], lnode(u8)	naive	405	(1,2)
	netbsd	39	(1,1)		opt	682	(4,2)
	newlib _s	42	(1,1)		opt	535	(4,2)
	newlib _f	405	(4,1)		naive	409	(1,2)
lnode(u8), lnode(u8)	openbsd	40	(1,1)	u8[], cnode(u8)	opt	553	(4,2)
	uClibc	38	(1,1)		naive	357	(1,2)
	naive	47	(1,1)		opt	514	(4,2)
	opt	293	(4,1)		opt	616	(4,2)
cnode(u8), cnode(u8)	opt	254	(4,1)	lnode(u8), cnode(u8)	naive	357	(1,2)
					opt	514	(4,2)
					opt	616	(4,2)
					opt	616	(4,2)

S2C suffers from a similar trade-off with regards to the approximation parameters d_o and d_u – a smaller than necessary value would cause a failed equivalence check while a larger value may have major impact on the size of queries and counterexamples for reasons discussed next. Consider a recursive relation relating values of a non-linear ADT such as **Tree**. It’s d -depth approximation reduces into $\mathcal{O}(2^d)$ scalar equalities resulting in large SMT queries, which to our experience is a major source of SMT solver timeouts during evaluation. We partially subvert this issue by using an iterative deepening strategy for type II queries – we try prove and disprove attempts with approximation parameters smaller than the maximum d_o and d_u , in hopes that either attempt would succeed saving valuable time waiting on SMT solvers. We would this strategy to be rather helpful during our evaluation.

Also, the completeness of type III proof obligations is highly contingent on the precision of our points-to analysis on \mathcal{C} as well as the deconstruction programs being checked for equivalence as part of the nested bisimulation check. We found our coarse-grained $\{1,2+\}$ categorization of allocation recency combined with allocation-site based points-to analysis to be quite good at identifying required points-to invariants. However, such an abstraction is far from complete.

Chapter 6

Conclusion

As introduced in chapter 1, most of the current solutions to the problem of equivalence checking between a functional specification and a C program relies heavily on manually provided correlation, inductive invariants as well as proof assistants for discharging said obligations. While the size of programs considered in our work is quite small, we hope the ideas in S2C will help automate the proofs for such systems to some degree.

Prior work on push-button verification of specific systems [17, 42, 40, 41] involves a combination of careful system design and automatic verification tools like SMT solvers. Constrained Horn Clause (CHC) Solvers [21] encode verification conditions of programs containing loops and recursion, and raise the level of abstraction for automatic proofs. Comparatively, S2C further raises the level of abstraction for automatic verification from SMT queries and CHC queries to lifting constructors and automatic discharge of proof obligations involving recursive relations.

A key idea in S2C is the conversion of proof obligations involving recursive relations to bisimulation checks. Thus, S2C performs *nested* bisimulation checks as part of a ‘higher-level’ bisimulation search. This approach of identifying recursive relations as invariants and using bisimulation to discharge the associated proof obligations may have applications beyond equivalence checking.

Bibliography

- [1] Cvc4 theorem prover webpage. <https://cvc4.github.io/>, 2023.
- [2] diet libc webpage. <https://www.fefe.de/dietlibc/>, 2023.
- [3] Gnu libc sources. <https://sourceware.org/git/glibc.git>, 2023.
- [4] klibc libc sources. <https://git.kernel.org/pub/scm/libs/klibc/klibc.git>, 2023.
- [5] musl libc sources. <https://git.musl-libc.org/cgit/musl>, 2023.
- [6] Netbsd libc sources. <http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/>, 2023.
- [7] Newlib libc sources. <https://www.sourceware.org/git/?p=newlib-cygwin.git>, 2023.
- [8] Openbsd libc sources. <https://github.com/openbsd/src/tree/master/lib/libc>, 2023.
- [9] uclibc libc sources. <https://git.uclibc.org/uClibc/>, 2023.
- [10] Lars Ole Andersen. Program analysis and specialization for the C programming language. Technical report, 1994.
- [11] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis*, SAS’06, page 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.
- [12] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. Tvoc: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K.

- Rajamani, editors, *Computer Aided Verification*, pages 291–295, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [13] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 14–25, New York, NY, USA, 2004. Association for Computing Machinery.
- [14] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. volume 58, pages 117 – 148, 12 2003.
- [15] Benjamin C. *Types and Programming Languages*. MIT Press, 2002.
- [16] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 296–310, New York, NY, USA, 1990. Association for Computing Machinery.
- [17] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.
- [18] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1027–1040, New York, NY, USA, 2019. ACM.
- [19] Jeffrey Considine. Efficient hash-consing of recursive types. Technical report, USA, 2000.
- [20] Program Equivalence in Coq. <https://softwarefoundations.cis.upenn.edu/plf-current/Equiv.html>, apr 2023.
- [21] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through horn clause transformation. In Xavier Rival, editor, *Static Analysis*, pages 147–169, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

- [22] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [23] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.
- [24] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pages 349–360, New York, NY, USA, 2014. ACM.
- [25] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity, FME '01*, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.
- [26] Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP '04*, page 150–161, New York, NY, USA, 2004. Association for Computing Machinery.
- [27] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.
- [28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.
- [29] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82*, page 66–74, New York, NY, USA, 1982. Association for Computing Machinery.

- [30] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Validation of gcc optimizers through trace generation. *Softw. Pract. Exper.*, 39(6):611–639, April 2009.
- [31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.
- [32] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 327–337, New York, NY, USA, 2009. ACM.
- [33] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [34] A. Leung, D. Bounov, and S. Lerner. C-to-verilog translation validation. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 42–47, Sept 2015.
- [35] Nuno P. Lopes and José Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *Int. J. Softw. Tools Technol. Transf.*, 18(4):359–374, August 2016.
- [36] Simon Marlow et al. Haskell 2010 language report. Available online [http://www.haskell.org/\(May 2011\)](http://www.haskell.org/(May 2011)), 2010.
- [37] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 46–60, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

- [38] KedarS. Namjoshi and LenoreD. Zuck. Witnessing program transformations. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 304–323. Springer Berlin Heidelberg, 2013.
- [39] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.
- [40] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019.
- [41] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 41–61. USENIX Association, 2020.
- [42] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, pages 252–269, New York, NY, USA, 2017. ACM.
- [43] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, pages 151–166, London, UK, UK, 1998. Springer-Verlag.
- [44] Arnd Poetzsch-Heffter and Marek Gawkowski. Towards proof generating compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):37–51, May 2005.
- [45] Robert Harper Robin Milner, Mads Tofte and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

- [46] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 471–482, New York, NY, USA, 2013. Association for Computing Machinery.
- [47] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.
- [48] Shubhani. *Counterexample-guided Equivalence Checking*. PhD thesis, Indian Institute of Technology, Delhi, 2023.
- [49] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag.
- [50] Ofer Strichman and Benny Godlin. Regression verification - a practical way to verify programs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 496–501. Springer Berlin Heidelberg, 2008.
- [51] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM.
- [52] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.
- [53] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 35–51, Berlin, Heidelberg, 2008. Springer-Verlag.

- [54] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A methodology for the translation validation of optimizing compilers. 9(3):223–247, mar 2003.
- [55] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, November 2005.

Biography

Indrajit Banerjee is a MS(R) student of Computer Science & Engineering Department at IIT Delhi. He couldn't think of anything interesting to add here.

