

*Thesis on*

**Counterexample-Guided Verification of  
Imperative Programs Against Implementation  
Agnostic Functional Specification**

*by*

**Indrajit Banerjee**  
(2020CSY7569)

*Under the guidance of*

**Prof. Sorav Bansal**  
(Computer Science and Engineering)

*Submitted in the partial fulfillment  
of the requirements for the degree of*

**Master of Science (Research)**

*to the*



**Department of Computer Science and Engineering  
Indian Institute of Technology Delhi**

**June 2023**

# Certificate

This is to certify that the thesis titled “**Counterexample-Guided Verification of Imperative Programs Against Implementation Agnostic Functional Specification**”, being submitted by **Mr.Indrajit Banerjee**, to the Indian Institute of Technology, Delhi, for award of the degree **Master of Science (Research)**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Prof. Sorav Bansal**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology Delhi**  
**New Delhi - 110016**

# Acknowledgments

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 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Summary . . . . .	2
1.2	Our Contributions . . . . .	5
<b>2</b>	<b>Exposition of Languages and Equivalence Definition</b>	<b>7</b>
2.1	The Spec Language . . . . .	7
2.2	Intermediate Representations . . . . .	8
2.3	Bisimulation . . . . .	11
2.4	Proof Obligations . . . . .	11
2.5	Equivalence Definition . . . . .	11
2.6	Equivalence Definition . . . . .	12
<b>3</b>	<b>Algorithm through Linked List Examples</b>	<b>14</b>
3.1	Product CFG . . . . .	15
3.2	Recursive relations . . . . .	16
3.3	Proof Obligations . . . . .	18
3.4	Proof Discharge Algorithm and Its Soundness . . . . .	18
3.5	Iterative Unification and Unrolling . . . . .	20
3.6	$k$ -unrolling with respect to an unrolling procedure . . . . .	22
3.7	Handling Type I Proof Obligations . . . . .	23
3.8	Handling Type II Proof Obligations . . . . .	25
3.9	Handling Type III Proof Obligations . . . . .	30
<b>4</b>	<b>Formalism</b>	<b>38</b>
4.1	The Spec Language . . . . .	38
4.2	Counterexample-guided Product-CFG Construction . . . . .	39
4.3	Invariant Inference and Counterexample Generation . . . . .	42
4.4	Modeling Procedure Calls . . . . .	43
4.5	Points-to Analysis . . . . .	44

---

<b>5</b>	<b>Evaluation</b>	<b>45</b>
5.1	Experiments . . . . .	46
5.2	Results . . . . .	53
<b>6</b>	<b>Limitations</b>	<b>53</b>
<b>7</b>	<b>Conclusion</b>	<b>54</b>
<b>8</b>	<b>Outline of the Thesis</b>	<b>55</b>

---

# 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 and has several important applications such as (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 and more.

The verification of a C implementation against its manually written functional specification through manually-coded refinement proofs has been performed extensively in the seL4 microkernel [25]. Frameworks for program equivalence proofs have been developed in interactive theorem provers like Coq [16] where correlations and invariants are manually identified during proof codification. On the other hand, programming languages like Dafny [27] 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 [33, 43, 40, 42, 26, 45, 46, 37, 44, 28, 24, 30, 11, 39, 15, 22, 38, 32] across low level programming languages such as C and assembly. 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 has equivalent observable behaviour. On the other hand, a sound equivalence checker may be incomplete and fail to prove the programs to be equivalent, even if they were equivalent.

We present S2C, a *sound* algorithm to automatically (push-button) search for a proof of equivalence between a functional specification (written in Spec) and its optimized C implementation. 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 of a *list*) and (b) algorithmic strategies (e.g. alternate algorithms, optimizations) against a *single* functional specification. This opens the possibility of regression verification [41, 20], where S2C can be used to automate verification across software updates that change

---

memory layouts for data structures.

## 1.1 Summary

We restrict 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, where algebraic data types (ADTs) [13] 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 and bitvector ( $i<N>$ ) types.

Next, we give a brief overview of our approach through an example. This allows us to introduce the major subgoals and we state our primary contributions in the next section.

```

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

```

(b) C Program with malloc()

**Figure 1:** Spec and C Programs constructing a Linked List.



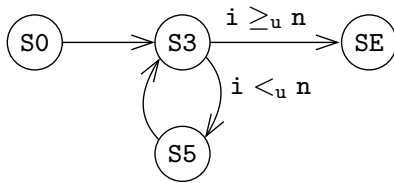
<pre> S0: List mk_list (i32 n) { S1:   List l := LNil; S2:   i32 i := 0<sub>i32</sub>; S3:   while ¬(i ≥<sub>u</sub> n): S4:     l := LCons(i, l); S5:     i := i + 1<sub>i32</sub>; S6:   return l; SE: }</pre>	<pre> C0: i32 mk_list (i32 n) { C1:   i32 l := 0<sub>i32</sub>; C2:   i32 i := 0<sub>i32</sub>;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

(a) (Abstracted) Spec IR

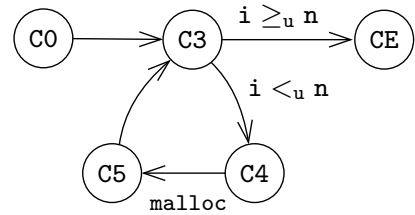
(b) (Abstracted) C IR

**Figure 2:** IRs for the Spec and C Programs in figs. 1a and 1b respectively.

Figures 1a and 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. 1a. An empty `List` is represented by the 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 are its well-typed arguments, which may include recursive data structure values. The inputs to a C procedure are its explicit arguments and the implicit state of program memory at procedure entry. We lower both Spec and C programs to a common intermediate representation (IR) as shown in figs. 2a and 2b. For the Spec program in fig. 1a, the tail-recursive function `mk_list_impl` is converted to a loop and inlined in the top-level function `mk_list`. For the C program in fig. 1b, the sizes and memory layouts of both scalar (e.g., `unsigned`) and compound (e.g., `struct lnode`) types are concretized in the IR.



(a) CFG of Spec Program



(b) CFG of C Program

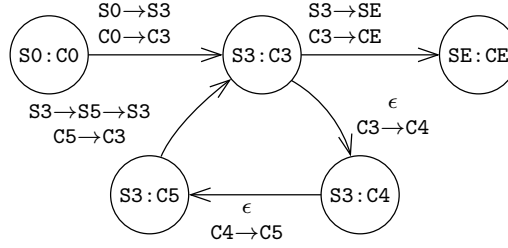
**Figure 3:** CFG representation for Spec and C IRs shown in figs. 2a and 2b

Figures 10a and 10b show the Control-Flow Graph (CFG) representation of the Spec and C IR programs in figs. 2a and 2b respectively. Each node represents a PC

**Table 1:** Node Invariants for Product-CFG in fig. 11

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) (S3:C5)	(I5) $n_S = n_C$ (I6) $i_S = i_C$ (I7) $i_S <_u n_S$ (I8) $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(SE:CE)	(E) $\text{ret}_S \sim \text{Clist}_m^{\text{lnode}}(\text{ret}_C)$

location of its corresponding program, and each edge represent conditional transition between PCs through instruction execution. For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 10b, the edge  $C5 \rightarrow C3$  represents the path  $C5 \rightarrow C6 \rightarrow C7 \rightarrow C8 \rightarrow C3$ .

**Figure 4:** Product-CFG between the CFGs in figs. 10a and 10b

We construct a *bisimulation relation* to identify equivalence between the two programs. A bisimulation relation correlates the transitions of Spec and C programs in lockstep, such that the lockstep execution ensures identical observable behavior. A bisimulation relation between two programs can be represented using a *product program* [44] and the CFG representation of a product program is called a *product-CFG*. Figure 11 shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 10a and 10b.

At each node of the product-CFG, invariants relate the states of the Spec and C program respectively. Table 2 lists invariants for the product-CFG in fig. 11. At the start node  $S0:C0$  of the product-CFG, the precondition  $Pre$  (labeled (P)) ensures equality of input arguments  $n_S$  and  $n_C$  at the programs' entry. Inductive invariants (labeled (I)) are inferred at each intermediate product-CFG node (e.g.,  $S3:C3$ ) relating both programs' states. For example, at node  $S3:C5$ , (I6)  $i_S = i_C$  is an inductive invariant.

In table 2, the invariant (I4)  $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$  is an example of a recursive

relation and represents equality between the Spec `List` variable `lS` and the `List` represented by chasing the `lnode` pointers starting at `lC`.  $\text{Clist}_{\mathfrak{m}}^{\text{lnode}}$  is an example of a *lifting constructor* that ‘lifts’ a C pointer value (pointing to an object of type `struct lnode`) and the C memory state  $\mathfrak{m}$  to a Spec `List` value, and is defined as follows:

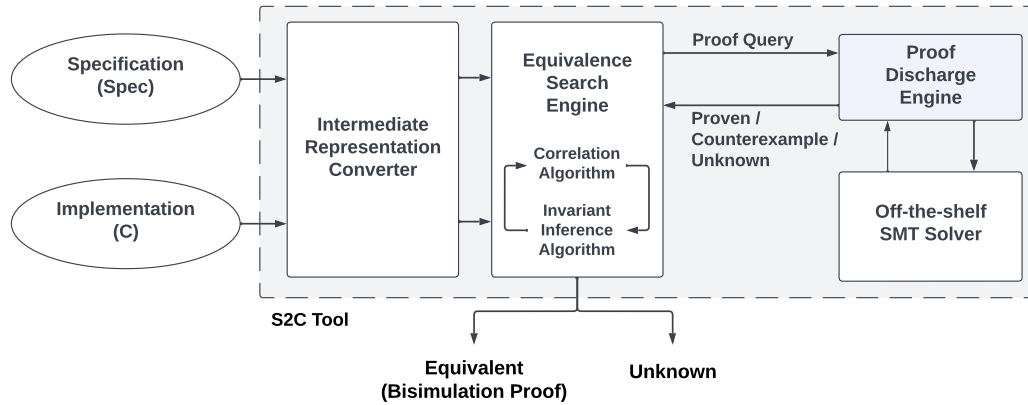
$$U_C : \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p:\text{i32}) = \text{if } p = 0 \text{ then } \text{LNil} \\ \text{else } \text{LCons}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{val}, \text{Clist}_{\mathfrak{m}}^{\text{lnode}}(p \xrightarrow{\mathfrak{m}}_{\text{lnode}} \text{next})) \quad (1)$$

Product-CFG invariants involving recursive relations (e.g.,  $\textcircled{\text{I4}}$ ) allow us to express equality between native Spec values with the C program state. Assuming that the precondition  $\text{Pre } (\textcircled{\text{P}})$  holds at the entry node `S0:C0`, a bisimulation check involves checking that the inductive invariants hold too, and consequently the postcondition  $\text{Post } (\textcircled{\text{E}})$  holds at the exit node `SE:CE`. Checking whether an invariant holds results in proof queries. These proof obligations are expressed as relational Hoare triples [12, 23] and discharged through a proof discharge algorithm i.e. a solver. We give a more formal exposition of the concepts introduced in this summary in the coming chapters.

## 1.2 Our Contributions

As previously summarized in section 1.1, showing equivalence of a Spec and a C program through a bisimulation proof requires three major procedures:  $\textcircled{1}$  An algorithm for construction of a product-CFG by correlating program executions across the Spec and C programs respectively.  $\textcircled{2}$  An algorithm for identification of inductive invariants at intermediate correlated PCs.  $\textcircled{3}$  An algorithm for solving proof obligations containing recursive relations. Our major contributions are as follows:

- **Proof Discharge Algorithm:** Solving proof obligations ( $\textcircled{3}$ ) involving recursive relations is rather interesting and forms our primary contribution. We describe a *sound* proof discharge algorithm capable of tackling proof obligations involving 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



**Figure 5:** A brief overview of our equivalence checker algorithm S2C. The inputs to the algorithm are the Spec and C programs. S2C either successfully finds a bisimulation proof implying equivalence or soundly fails with an unknown verdict.

queries. These counterexamples are the backbone of counterexample-guided algorithms for ① and ② steps. As part of our proof discharge procedure, we reformulate equality of values (i.e. recursive relations) as equivalence of their corresponding programs and discharge these proof queries using a nested (albeit much simpler) bisimulation check.

- **Spec-to-C Automatic Equivalence Checker Tool:** Our second contribution is S2C, an equivalence checker tool capable of proving equivalence between a Spec and a C program automatically. S2C is based on the Counter tool[22] and uses modified versions of (a) counterexample-guided correlation algorithm for incremental construction of a product-CFG and (b) counterexample-guided invariant inference algorithm for inference of inductive invariants at correlated PCs in the (partially constructed) product-CFG. S2C discharges required verification conditions (i.e. proof obligations) using our Proof Discharge Algorithm. Figure 5 gives an overview of the complete algorithm.

## 2 Exposition of Languages and Equivalence Definition

### 2.1 The Spec Language

We start with a discussion on the Spec language. Spec supports recursive algebraic data types (ADT) similar to the ones available in most functional languages. Additionally, Spec is equipped with the following scalar types: `Unit`, Boolean (`Bool`) and Bitvector of length  $N$  ( $i < N$ ). ADTs can be thought of as ‘sum of product’ types where each constructor represents a variant and the arguments to each constructor represents its fields. Evidently, types in Spec can be represented in *first order recursive types* with `Product` and `Sum` type constructors and `Unit`, `Bool`,  $i < N$  types (i.e., nullary type constructors) as follows:

$$T \rightarrow \mu\alpha. T \mid \text{Product}(T, \dots, T) \mid \text{Sum}(T, \dots, T) \mid \text{Unit} \mid \text{Bool} \mid i \langle N \rangle \mid \alpha$$

For example, the `List` type can be written as  $\mu\alpha. \text{Sum}(\text{Unit}, \text{Product}(i32, \alpha))$ .

The language also borrows its expression grammar heavily from functional languages. This includes the usual constructs like `let-in`, `if-then-else`, function application and the `match` statement for pattern-matching (i.e. deconstructing) sum and product values. Unlike functional languages, Spec only supports first order functions. Also, Spec does not support partial function application. Hence, we constrain our attention to C programs containing only first order functions. Spec is equipped with a special `assuming-do` construct for explicitly providing assertions. Spec also provides the typical boolean and bitvector operators for expressing computation in C succinctly yet explicitly. This includes logical operators (e.g., `and`), bitvector arithmetic operators (e.g., `bvadd(+)`) and relational operators for comparing bitvectors interpreted as signed or unsigned integers (e.g.,  $\leq_{u,s}$ ).

---

$\langle \text{expr} \rangle$	$\rightarrow$	$\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$ $ \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}_{i < N} \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$
$\langle \text{expr-list} \rangle$	$\rightarrow$	$\epsilon   \langle \text{expr} \rangle , \langle \text{expr-list} \rangle$
$\langle \text{id-list} \rangle$	$\rightarrow$	$\epsilon   \langle \text{id} \rangle , \langle \text{id-list} \rangle$
$\langle \text{literal}_{\text{Unit}} \rangle$	$\rightarrow$	$()$
$\langle \text{literal}_{\text{Bool}} \rangle$	$\rightarrow$	$\text{false}   \text{true}$
$\langle \text{literal}_{i < N} \rangle$	$\rightarrow$	$[0 \dots 2^N - 1]$

**Figure 6:** Simplified expression grammar of Spec language

## 2.2 Intermediate Representations

As summarized in section 1.1, we lower both Spec and C programs to a common intermediate representation (IR) for comparison. IR is a Three-Address-Code (3AC) style intermediate representation. We often omit intermediate registers in the IR for brevity and ease of exposition, and refer to this as the *abstracted* IR.

Figures 12a and 12b show Spec and C programs that traverse a linked list and return the sum of all the values in the linked list. The corresponding IR programs are shown in figs. 8a and 8b.

During conversion of a Spec source (figs. 1a and 12a resp.) to IR (figs. 2a and 8a resp.), (a) **match** statements are lowered to explicit **if-then-else** conditionals where each branch represents a distinct constructor, (b) all tail-recursive calls are converted to loops while non-tail calls are preserved and (c) all helper functions are inlined at their call-site.

Similarly, the following is performed during conversion of a C source (figs. 1b and 12b resp.) to IR (figs. 2b and 8b resp.): (a) the sizes and memory layouts of both scalar (e.g., **unsigned**) and compound (e.g., **struct lnode**) types are

```

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) => sum_list_impl(rest, sum + x).
A6:
A7: fn sum_list (l:List) : i32 = sum_list_impl(l, 0i32).

```

(a) Spec Program

```

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: }

```

(b) C Program

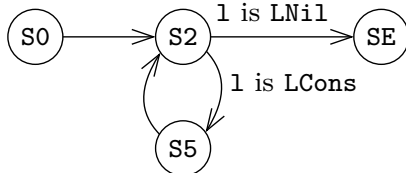
**Figure 7:** Spec and C Programs traversing a Linked List.

```

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: }

```

(a) (Abstracted) Spec IR



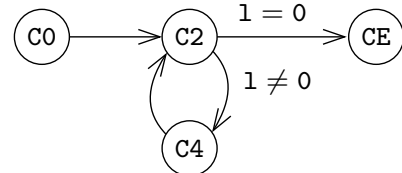
(c) CFG of Spec Program

```

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: }

```

(b) (Abstracted) C IR



(d) CFG of C Program

**Figure 8:** IRs and CFGs of the Spec and C Programs in figs. 12a and 12b respectively.

concretized, (b) the program memory along with reads and writes to it are made explicit and (c) we annotate `malloc` calls with the call-site i.e. IR PC (e.g., `mallocC4` in fig. 2b).

The IR supports both scalar and ADT types available in Spec. Each ADT value is modeled as a key-value dictionary that maps each of its field names to the constituent values. These key-value pairs are accessed using the *accessor*-operator, e.g., `l.val` and `l.next` represents the first and second fields of the `LCons` constructor in fig. 8a. The IR also allows querying the top-level value constructor of an ADT value using the *is*-operator, e.g., `l is LNil` in fig. 8a. Importantly, `l.val` is only well-formed if `l` is `LCons`. The construction of the Spec IR ensures the well-formedness of all expressions. Using the *accessor*- and *is*-operators, a `List` value `l` can be expanded as:

$$U_S : l = \text{if } l \text{ is LNil then LNil else LCons}(l.\text{val}, l.\text{next}) \quad (2)$$

In this expanded representation of `l`, the *sum-deconstruction* operator ‘if-then-else’<sup>1</sup> conditionally deconstructs the sum type into its variants `LNil` and `LCons`. Equation (3) is called the *unrolling procedure* for the `List` variable `l`. We can similarly define the unrolling procedure for any ADT variable.

The C memory is modeled as a byte-addressable array  $\mathfrak{m}$  in the IR. Memory reads are represented using the following two C-like syntaxes: (a) “ $p \xrightarrow{\mathfrak{m}}_{\mathsf{T}} \mathsf{f}$ ” is equivalent to “ $*(\text{typeof}(\mathsf{T}.\mathsf{f}))(\&\mathfrak{m}[p + \text{offsetof}(\mathsf{T}, \mathsf{f})])$ ” i.e., it returns the bytes in the memory array  $\mathfrak{m}$  starting at address ‘ $p + \text{offsetof}(\mathsf{T}, \mathsf{f})$ ’ and interpreted as an object of type ‘ $\text{typeof}(\mathsf{T}.\mathsf{f})$ ’ and (b) “ $p[i]_{\mathfrak{m}}^{\mathsf{T}}$ ” is equivalent to “ $*(\mathsf{T}*)(\&\mathfrak{m}[p + i \times \text{sizeof}(\mathsf{T})])$ ” i.e., it returns the bytes in the memory array  $\mathfrak{m}$  starting at address ‘ $p + i \times \text{sizeof}(\mathsf{T})$ ’ and interpreted as an object of type ‘ $\mathsf{T}$ ’. “ $\mathfrak{m}[a \leftarrow v]_{\mathsf{T}}$ ” represents an array that is equal to  $\mathfrak{m}$  everywhere except at addresses  $[a, a + \text{sizeof}(\mathsf{T}))$  which contains the value  $v$  of type ‘ $\mathsf{T}$ ’.

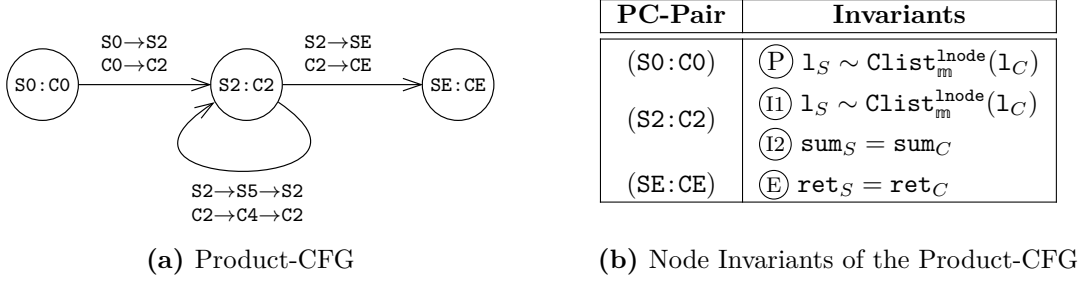
Figures 8c and 8d show the Control-Flow Graph (CFG) representation of the Spec and C IRs in figs. 8a and 8b respectively. Each CFG node represents a

---

<sup>1</sup>The sum-deconstruction operator ‘if-then-else’ for an ADT type  $T$  must contain exactly one branch for each top-level value constructor of  $T$ . For example, ‘if-then-else’ for the `List` type must have exactly two branches of the form `LNil` and `LCons( $e_1, e_2$ )` for some expressions  $e_1$  and  $e_2$ .

---





**Figure 9:** Product-CFG between the CFGs in figs. 8c and 8d. The inductive invariants of the Product-CFG are given in fig. 9b.

IR PC location of the program and edges represent transitions through execution of instructions. Each edge is associated with (a) a *edge condition* (the condition under which that edge is taken), (b) a *transfer function* (how the program state is mutated if that edge is taken) and (c) a *UB assumption* (what condition should be true for the program execution to be well-defined across this edge). In Spec, assertions expressed using the **assuming-do** statement form the UB assumptions. For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 10b, the edge  $C5 \rightarrow C3$  represents the path  $C5 \rightarrow C6 \rightarrow C7 \rightarrow C8 \rightarrow C3$ . In such a case, the transfer function of the edge is the composition of the sequence of instructions.

## 2.3 Bisimulation

## 2.4 Proof Obligations

## 2.5 Equivalence Definition

We restrict our attention to programs that construct, read, and write to recursive data structures in Spec and C. If a Spec or C program contains multiple procedures, we first convert all tail-recursive calls to loops, and then inline all non-recursive procedure calls to obtain a single top-level procedure which is compared for equivalence. A top-level procedure may make recursive calls to itself (which are not tail recursive). The C program may also contain calls to memory allocation library functions like `malloc` whose abstract semantics are available.

The inputs to a Spec procedure are its explicit program arguments, which may

include recursive data-structure values. The inputs to a C procedure include the explicit arguments passed to the C procedure (e.g., pointers) and the implicit state of program memory at procedure entry. Notice the difference in the nature of inputs to the two programs: while Spec inputs are explicit well-typed values, C procedure's inputs may be derived from the state of the input memory (e.g., linked list formed by chasing the `next` pointer). For checking equivalence, we require the user to specify a precondition (at the entry of both programs) that relates these two different types of program inputs.

Figure 2a shows the Three-Address-Code (3AC) style intermediate representation (IR) of the linked-list construction Spec program in fig. 1a. We often omit intermediate registers in the intermediate representation for brevity and ease of exposition, and refer to this as *abstracted* IR. The primary differences between the Spec source and IR are: (a) tail-recursive calls are converted to loops in IR, and (b) `match` statements are converted to `if-then-else` in IR, where each branch of an `if-then-else` expression represents a distinct constructor.

Similarly, the C implementation is also lowered to a 3AC IR that resembles LLVM IR [29]. The primary differences between a C source and its IR are: (a) the sizes and memory-layouts of both scalar (e.g., `int`) and compound (e.g., `struct`) types are concretized in the IR, and (b) we annotate any `malloc` calls with the IR PC at which that call appears (e.g., `mallocC4` in fig. 2b).

S2C computes equivalence between the IR of the Spec and C source programs. Henceforth, we will omit the source representation and only show the IR of both Spec and C programs. We will continue to refer to these IRs as Spec and C respectively.

## 2.6 Equivalence Definition

Given (1) a Spec program specification  $S$ , (2) a C implementation  $C$ , (3) a precondition  $Pre$  that relates the initial inputs  $\text{Input}_S$  and  $\text{Input}_C$  to  $S$  and  $C$  respectively, and (4) a postcondition  $Post$  that relates the final outputs  $\text{Output}_S$  and  $\text{Output}_C$  of  $S$  and  $C$  respectively<sup>2</sup>:

$S$  and  $C$  are equivalent under precondition  $Pre$  if for all possible inputs  $\text{Input}_S$

---

<sup>2</sup> $\text{Input}_C$  and  $\text{Output}_C$  include the initial and final memory state of  $C$  respectively.

and  $\text{Input}_C$ , such that  $Pre(\text{Input}_S, \text{Input}_C)$  holds,  $S$ 's execution is well-defined on  $\text{Input}_S$ , and  $C$ 's memory allocation requests during its execution on  $\text{Input}_C$  are successful, then both programs  $S$  and  $C$  produce outputs that satisfy  $Post$ .

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

The  $(S \text{ def})$  antecedent states that we are only interested in proving equivalence for well-defined executions of  $S$ , i.e., executions that are free of undefined behaviour (UB). For example, division-by-zero is UB in  $S$ . Sometimes the user may be interested in constraining the nature of inputs to the  $C$  program, e.g., the `strlen(char* sC)` function is well-defined only if  $s_C$  is not null. Thus, for `strlen`, we are only interested in computing equivalence for non-null input pointers. Spec has no notion of pointers and so this condition cannot be encoded in  $S$  alone. In these cases, we use a combination of  $Pre$  and  $(S \text{ def})$  to constrain the executions of  $C$  for which we are interested in proving equivalence. In the `strlen` example,  $(S \text{ def})$  is encoded as an abstract condition that the input string  $s_S$  is “not invalid”, written  $\neg(s_S \text{ is SInvalid})$ , where `SInvalid` is a constructor for the Spec `String` type. The precondition  $Pre$  then contains the relation  $(s_S \text{ is SInvalid}) \Leftrightarrow (s_C = 0)$ . This ensures that we compute equivalence only for those executions of  $C$  where the input pointer  $s_C$  is non-null. The use of an explicit constructor for expressing ill-formedness of Spec input values along with a cleverly chosen  $Pre$  allows us to constrain the executions to  $Spec$  and  $C$  to well-formed inputs only during equivalence check. Please refer to Chapter XXX of thesis for a more detailed explanation of this strategy for the function `strchr`.

The  $(C \text{ fits})$  antecedent states that we prove equivalence only if the  $C$  program's memory requirements fit within the available system memory, i.e., only for those executions of  $C$  in which all memory allocation requests (through `malloc` calls) are successful.

The returned values of  $S$  and  $C$  procedures form their observable outputs. For  $S$ , the returned values are explicit and may include well-typed recursive data-structure objects. For  $C$ , observable returned values also include portions of the implicit memory state at program exit. The postcondition relates these outputs of the two programs.

### 3 Algorithm through Linked List Examples

A `List` ADT in the Spec program is defined at line A0 in fig. 1a. An empty list is represented by the constant `LNil()`<sup>3</sup>; a non-empty list uses the `LCons` constructor to combine its first value (`val:i32`) and the remaining list (`tail:List`). Spec supports `i<N>` (bitvectors of length `N`), `bool`, and `unit` types, also called *scalar types*. Spec’s type system prevents the creation of cycles in ADT values. If `l` is an object of type `List`, then to access its constituent values, we may expand (or unroll) `l` to

$$U_S:l = \text{if } l \text{ is LNil then LNil else LCons}(l.\text{val}, l.\text{tail}) \quad (3)$$

In this expanded representation of `l`, the *sum-deconstruction* operator<sup>4</sup> ‘if-then-else’ deconstructs a sum type where the if condition ‘`l is Constructor`’ checks whether the top level constructor of `l` is ‘`Constructor`’. If `l` is a non-empty list constructed through `LCons`, then `l.val` and `l.tail` are used to access `l`’s first value and `l`’s tail respectively. The right-hand side of eq. (3) can also be viewed as an executable program that unrolls the input `List` object `l` once and outputs a `List` object constructed from `l`’s constituents — we call eq. (3) the *unrolling procedure*  $U_S$  of the `List` ADT. We can similarly define the unrolling procedure for any ADT variable.



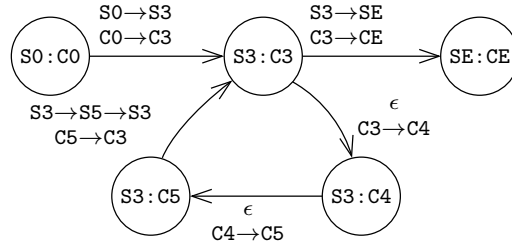
**Figure 10:** CFG representation for Spec and C IRs shown in figs. 2a and 2b

Figures 10a and 10b show the Control-Flow Graph (CFG) representation of the

<sup>3</sup>`LNil()` represents the application of the nullary constructor `LNil` on the unit value `()`. For brevity, we will simply write `LNil` for `LNil()` henceforth.

<sup>4</sup>The sum-deconstruction operator ‘if-then-else’ for a sum type  $T$  must contain exactly one branch for each top-level value constructor of  $T$ . For example, ‘if-then-else’ for the `List` type must have exactly two branches of the form `LNil` and `LCons( $e_1, e_2$ )` for some expressions  $e_1$  and  $e_2$ .

Spec and C programs in figs. 2a and 2b respectively. The CFG nodes represent PC locations of the program, and edges represent transitions through instruction execution. For brevity, we sometimes represent multiple program instructions with a single edge, e.g., in fig. 10b, the edge  $C5 \rightarrow C3$  represents the path  $C5 \rightarrow C6 \rightarrow C7 \rightarrow C8 \rightarrow C3$ . A control-flow edge is associated with an *edge condition* (the condition under which that edge is taken), a *transfer function* (how the program state is mutated if that edge is taken), and a *UB assumption* (what condition should be true for the program execution to be well-defined across this edge). For example, the UB assumption associated with a division instruction in  $S$  will encode that the divisor must be non-zero.



**Figure 11:** Product-CFG between the CFGs in figs. 10a and 10b

### 3.1 Product CFG

We construct a *bisimulation relation* to identify equivalence between two programs. A bisimulation relation correlates the transitions of  $S$  and  $C$  in lockstep, such that this lockstep execution ensures identical observable behavior. An equivalence proof through bisimulation construction can be represented using a *product program* [44] and the CFG of a product program is called a *product-CFG*. Figure 11 shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 10a and 10b.

A node in the product-CFG is formed by pairing nodes of  $S$  and  $C$  CFGs, e.g.,  $(S3:C5)$  is formed by pairing  $S3$  and  $C5$ . If the lockstep execution is at node  $(S3:C5)$  in the product-CFG, then  $S$ 's execution is at  $S3$  and  $C$ 's execution is at  $C5$ . The start node  $(S0:C0)$  of the product-CFG correlates the start nodes of the CFGs of both programs. Similarly, the exit node  $(SE:CE)$  correlates the exit nodes of the CFGs of both programs.

**Table 2:** Node Invariants for Product-CFG in fig. 11

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) (S3:C5)	(I5) $n_S = n_C$ (I6) $i_S = i_C$ (I7) $i_S <_u n_S$ (I8) $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$
(SE:CE)	(E) $\text{ret}_S \sim \text{Clist}_m^{\text{lnode}}(\text{ret}_C)$

An edge in the product-CFG is formed by pairing a path (a sequence of edges) in  $S$  with a path in  $C$ . A product-CFG edge encodes the lockstep execution of correlated transitions (or paths). For example, the product-CFG edge  $(S3:C5) \rightarrow (S3:C3)$  is formed by pairing the  $(S3 \rightarrow S5 \rightarrow S3)$  and  $C5 \rightarrow C3$  in figs. 10a and 10b, and represents that when  $S$  makes a transition  $(S3 \rightarrow S5 \rightarrow S3)$ , then  $C$  makes the transition  $C5 \rightarrow C3$  in lockstep. The edge  $(S3:C3) \rightarrow (S3:C4)$  correlates the  $\epsilon$  path (no transition) in  $S$  with  $C3 \rightarrow C4$  in  $C$ . In general, a product-CFG edge  $e$  may correlate a finite path  $\rho_S$  in  $S$  with a finite path  $\rho_C$  in  $C$ , written  $e = (\rho_S, \rho_C)$ .

At the start node (S0:C0) of the product-CFG, the precondition  $Pre$  (labeled (P)) ensures the equality of input arguments  $n_S$  and  $n_C$  at programs' entry. Inductive invariants are inferred at each product-CFG node that relate the variables of  $S$  with variables and memory locations of  $C$ . The inductive invariants are identified by running an invariant inference algorithm on the product-CFG, which is further discussed in section 4.3. The inductive invariants for our example are shown in table 2. For example, at node (S3:C5) in fig. 11,  $i_S = i_C$  is an inductive invariant. If the inferred invariants ensure that the postcondition  $Post$  holds at the exit node (SE:CE) (labeled (E)), we have shown equivalence of both programs.

## 3.2 Recursive relations

TODO:try to update the intro to recursive relations(first line)

In table 2, the relation between programs' variables at product-CFG nodes S3:C3, S3:C4 and S3:C5 is encoded as a recursive relation: " $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$ " where  $l_S$  and  $l_C$  represent the `l` variables in the Spec and C programs respectively, `lnode` represents the C `struct` type that contains the `val` and `next` fields, and  $m$  represents a 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  $l_2$ , i.e.,  $l_1$  and  $l_2$  are isomorphic and have equal values<sup>5</sup>. The *lifting constructor*  $\text{Clist}_m^{\text{lnode}}(\mathbf{p})$  is a constructor that *lifts* the C pointer value  $\mathbf{p}$  (pointing to an object of `struct lnode`) and the C memory state  $m$  to a Spec List value.  $\text{Clist}_m^{\text{lnode}}(\mathbf{p})$  is defined through its unrolling procedure as:

$$U_C : \text{Clist}_m^{\text{lnode}}(\mathbf{p} : \text{i32}) = \underline{\text{if}} (\mathbf{p} == 0) \underline{\text{then}} \text{LNil} \\ \underline{\text{else}} \text{LCons}(p \xrightarrow{m}_{\text{lnode}} \text{val}, \text{Clist}_m^{\text{lnode}}(p \xrightarrow{m}_{\text{lnode}} \text{next})) \quad (4)$$

By construction, this unrolling procedure  $U_C$  is isomorphic to List’s unrolling procedure  $U_S$  in eq. (3). “ $p \xrightarrow{m}_s \mathbf{f}$ ” represents the field ‘ $\mathbf{f}$ ’ of the ‘struct  $\mathbf{s}$ ’ object pointed-to by pointer ‘ $\mathbf{p}$ ’ in memory state ‘ $m$ ’. When represented in C-like syntax, ‘ $p \xrightarrow{m}_s \mathbf{f}$ ’ is equivalent to “\*((typeof  $\mathbf{s.f}$ \*)&[ $\mathbf{p} + \text{offsetof}(\mathbf{s}, \mathbf{f})$ ])”, i.e., the expression ‘ $p \xrightarrow{m}_s \mathbf{f}$ ’ returns the bytes in the memory array ‘ $m$ ’ starting at address ‘ $\mathbf{p} + \text{offsetof}(\mathbf{s}, \mathbf{f})$ ’ and interpreted as an object of type ‘`typeof  $\mathbf{s.f}$` ’.

Note the recursive nature of the lifting value constructor `Clist`: if the pointer  $\mathbf{p}$  of type `i32`<sup>6</sup> is zero (i.e.  $\mathbf{p}$  is a null pointer), then this represents the empty list (`LNil`); otherwise it represents the list formed by `LCons`-ing the value stored at  $\mathbf{p} \rightarrow \text{val}$  in memory  $m$  and the list formed by recursively lifting  $\mathbf{p} \rightarrow \text{next}$  using `Clist` in memory  $m$ . The recursive lifting constructor `Clist` allows us to compare C values and Spec values for equality. In general, an equality relation between two (possibly recursive) ADT values is called a *recursive relation*. However in the context of bisimulation, we will only consider recursive relations between Spec values (such as variables) and lifted C values (lifted using a lifting constructor such as `Clist`).

We later discuss in section 4.2 how a product-CFG can be constructed automatically through a counterexample-guided search. Before that, we discuss the proof obligations that arise from a given product-CFG. Consider the product-CFG in fig. 11. Assuming that the precondition  $\textcircled{\text{P}}$  holds at the entry node `S0:C0` of this product-CFG, a bisimulation check involves checking that the invariants at the other product-CFG nodes hold too, and consequently the postcondition  $\textcircled{\text{E}}$  holds

---

<sup>5</sup> $l_1 \sim l_2$  and  $l_1 = l_2$  are equivalent — the former emphasizes the recursive nature of the values being compared.

<sup>6</sup>The IR lowers integers and pointers in C to bitvectors of type `i<N>`. e.g., `i32` is a 32-bit bitvector type.

---

at the exit node **SE:CE**. Recall that the precondition  $\textcircled{\text{P}}$  and the postcondition  $\textcircled{\text{E}}$  are provided by the user, but all the other invariants are inferred automatically.

### 3.3 Proof Obligations

We use relational Hoare triples to express these proof obligations [12, 23]. If  $\phi$  denotes a predicate relating the machine states of programs  $S$  and  $C$ , then for a product-CFG edge  $e = (\rho_S, \rho_C)$ ,  $\{\phi_s\}(e)\{\phi_d\}$  denotes the condition: if the machine states  $\sigma_S$  and  $\sigma_C$  of programs  $S$  and  $C$  are related through precondition  $\phi_s(\sigma_S, \sigma_C)$  and paths  $\rho_S$  and  $\rho_C$  are executed in  $S$  and  $C$  respectively (implying the path conditions hold), then execution terminates normally in states  $\sigma'_S$  (for  $S$ ) and  $\sigma'_C$  (for  $C$ ) where postcondition  $\phi_d(\sigma'_S, \sigma'_C)$  hold.  $\{\phi_s\}(e)\{\phi_d\}$  can also be written as  $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$ .

For every product-CFG edge  $e = (s \rightarrow d) = (\rho_S, \rho_C)$  in fig. 11, we thus need to prove  $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$ , where  $\phi_s$  and  $\phi_d$  are the node invariants (shown in table 2) at nodes  $s$  and  $d$  of the product-CFG 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:

$$(\phi_s \wedge \text{pathcond}_{\rho_S} \wedge \text{pathcond}_{\rho_C} \wedge \text{ubfree}_{\rho_S}) \Rightarrow \text{WP}_{\rho_S, \rho_C}(\phi_d) \quad (5)$$

Here,  $\text{pathcond}_{\rho_X}$  represents the condition that path  $\rho_X$  is taken in program  $X$ .  $\text{ubfree}_{\rho_S}$  represents the condition that the execution of program  $S$  along path  $\rho_S$  is free of undefined behavior.  $\text{WP}_{\rho_S, \rho_C}(\phi_d)$  represents the weakest-precondition of the predicate  $\phi_d$  across the product-CFG edge  $e = (\rho_S, \rho_C)$ . We will use “LHS” and “RHS” to refer to the left and right hand sides of the implication operator “ $\Rightarrow$ ” in eq. (5).

### 3.4 Proof Discharge Algorithm and Its Soundness

We call an algorithm that evaluates the truth value of a proof obligation, 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 falsifies 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 a counterexample) when the proof obligation was actually true.

For the proof obligations generated by our equivalence procedure, it is always safe for a proof discharge algorithm to return false (without a counterexample). If a proof discharge algorithm conservatively evaluates a proof obligation to false (when it was actually true), it may prevent the overall equivalence proof from completing successfully; however, importantly, the overall equivalence procedure remains sound.

Resolving the truth value of a proof obligation that contains a recursive relation such as  $\mathbf{l}_s \sim \mathbf{Clist}_m^{\mathbf{lnode}}(\mathbf{l}_c)$  is unclear. Fortunately, the shapes of the proof obligations generated by our equivalence checking algorithm are restricted, which makes it possible to soundly resolve these proof obligations.

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  $\phi_s$ , it must be one of  $\phi_s^1$ ,  $\phi_s^2$ , ..., or  $\phi_s^k$ . We call this the *conjunctive recursive relation* property of an invariant  $\phi_s$ .

A proof obligation  $\{\phi_s\}(e)\{\phi_d\}$ , where  $e = (\rho_s, \rho_c)$ , gets lowered using  $\mathbf{WP}_e(\phi_d)$  (as shown in eq. (5)) to a first-order logic formula of the following form:

$$(\eta_1^1 \wedge \eta_2^1 \wedge \dots \wedge \eta_m^1) \Rightarrow (\eta_1^r \wedge \eta_2^r \wedge \dots \wedge \eta_n^r) \quad (6)$$

In this formula, the LHS and RHS are written as conjunctions of  $\eta_i^l$  and  $\eta_j^r$  respectively (for  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ ). Each  $\eta_j^r$  relation is obtained from  $\mathbf{WP}_e(\phi_d^j)$ , where  $\phi_d = (\phi_d^1 \wedge \phi_d^2 \wedge \dots \wedge \phi_d^n)$ . Thus, due to the conjunctive recursive relation property of  $\phi_s$  and  $\phi_d$ , any recursive relation in eq. (6) must appear as one of  $\eta_i^l$  or  $\eta_j^r$ .

To simplify proof obligation discharge, we break a first-order logic proof obligation  $P$  of the form in eq. (6) into multiple smaller proof obligations of the form

$P_j : (\text{LHS} \Rightarrow \eta_j^r)$ , 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*.

### 3.5 Iterative Unification and Unrolling

We begin with some definitions. An expression  $e$  whose top-level constructor is a lifting constructor, e.g.,  $e = \text{Clist}_m^{\text{lnode}}(l_c)$ , is called a *lifted expression*. An expression  $e$  of the form  $v.a_1.a_2 \dots a_n$  i.e. a variable with *zero* or more *product deconstruction* operators applied on it, is called a *pseudo-variable*. By definition, variables are pseudo-variables. An expression  $e$  in which (1) all product deconstructors (e.g. ‘`tail`’) appear as part of a *pseudo-variable* and (2) each *sum-is* operator (e.g. ‘`is LCons`’) operate on a *pseudo-variable*, is called a *canonical expression*.

Consider the expression tree of a canonical expression  $e$  of ADT  $T$ , formed using the ADT value constructors and the if-then-else sum-deconstruction operator. The leaves of  $e$  (also called atoms of  $e$ ) are the pseudo-variables (of scalar or ADT type), the scalar expressions (of `unit`, `bool`, or `i<N>` types), and lifted expressions.

The *expression path* to a node  $v$  in  $e$ ’s tree is the path from the root of  $e$  to that node  $v$ . The *expression path condition* represents the conjunction of all the if conditions (if the then branch is taken on the expression path), or their negation (if the else branch is taken on the expression path) seen on the expression path. For example, in expression if (c) then a else b, the expression path condition of c is `true`, of a is c, and of b is  $\neg c$ .

When we attempt to unify two expressions, we unify the structures created by the value constructors and the ‘if-then-else’ operator of their canonical forms. The unification procedure either fails to unify, or it returns tuples  $(p_1, p_2, a_1, e_2)$  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.

For two non-atomic expressions  $e_1$  and  $e_2$  to unify successfully, it must be true that either the top-level node in both  $e_1$  and  $e_2$  have the same value constructor (in which case a unification is attempted for each of the children of the top-level constructor), *or* the top-level node in one of  $e_1$  or  $e_2$  is if-then-else. If the top-level node of  $e_1$  is “if (c) then  $e_1^{\text{then}}$  else  $e_1^{\text{else}}$ ”, we attempt to unify both  $e_1^{\text{then}}$

and  $e_1^{\text{else}}$  with  $e_2$  and return success iff any of these attempts succeed (similarly for  $e_2$ ). Whenever we descend down an if-then-else operator, we conjunct the corresponding if condition (for  $e_1^{\text{then}}$ ) or its negation (for  $e_1^{\text{else}}$ ) to the respective expression path condition. If one of  $e_1$  and  $e_2$  (say  $e_2$ ) is atomic, unification always succeeds and returns  $(p_2, p_1, e_2, e_1)$ .

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. Every (pseudo-)variable is associated with its unrolling procedure as governed by its ADT. For example, the unrolling procedure for a Spec variable  $l$  of **List** type is  $U_S$  (eq. (3)). For lifted expressions, the unrolling procedure is given by the definition of the lifting constructor such as  $U_C$  (eq. (4)) for the lifting constructor **Clist**.

Given two expressions  $e_a$  and  $e_b$  of an ADT  $T$  at expression path conditions  $p_a$  and  $p_b$  respectively, an *iterative unrolling and unification procedure*  $\Theta(e_a, e_b, p_a, p_b)$  is used to identify a set of correlation tuples between the atoms in the two expressions. This iterative procedure proceeds by attempting to unify  $e_a$  and  $e_b$ . If this unification fails, we return a unification failure for the original expressions  $e_a$  and  $e_b$ . Else, we obtain correlation 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  $(p_1, a_1, p_2, a_2)$  to the final output. 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 unroll  $a_1$  once 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  $\Theta(e_1, e_2, p_a \wedge p_1, p_b \wedge p_2)$ . The maximum number of unrollings performed by  $\Theta(e_a, e_b, p_a, p_b)$  (before converging) is upper bounded by the sum of number of ADT value constructors in  $e_a$  and  $e_b$ .

If a proof obligation involves a recursive relation  $e_a \sim e_b$ , we unify  $e_a$  and  $e_b$  through a call to  $\Theta(e_a, e_b, \text{true}, \text{true})$ . For example, the unification of “if ( $c_1$ ) then **LNil** else **LCons**(0,  $l_S$ )” and “if ( $c_2$ ) then **LNil** else **LCons**(0, **Clist** <sub>$m$</sub> <sup>**lnode**</sup>( $l_C$ ))” yields the correlation tuples:  $(c_1, (), c_2, ())$ ,  $(\neg c_1, 0, \neg c_2, 0)$  and  $(\neg c_1, l_S, \neg c_2, \text{Clist}_m^{\text{lnode}}(l_C))$ .

If the set of  $n$  tuples obtained after a successful unification of  $e_a \sim e_b$  are

$(p_1^i, a_1^i, p_2^i, a_2^i)$  (for  $i = 1 \dots n$ ), then  $e_a \sim e_b \Leftrightarrow \bigwedge_{i=1}^n ((p_1^i = p_2^i) \wedge ((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i)))$ <sup>7</sup>. We call  $\bigwedge_{i=1}^n ((p_1^i = p_2^i) \wedge ((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i)))$  the *decomposition* of  $e_a \sim e_b$ . Each conjunctive clause of one of the forms  $(p_1^i = p_2^i)$  and  $((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i))$  in this decomposition is called a *decomposition clause*. A decomposition clause may relate only atomic values, i.e., in the decomposed form, all recursive relations relate only ADT variables and/or lifted expressions. The decomposition for a failed unification is defined to be **false**. We *decompose* a recursive relation by replacing it with its decomposition. We *decompose* a proof obligation  $P$  by decomposing all recursive relations in  $P$ .

### 3.6 $k$ -unrolling with respect to an unrolling procedure

We can *unroll an expression  $e$  with respect to an unrolling procedure  $U$*  by substituting all occurrences of LHS in  $U$  by its unrolled version (RHS in  $U$ ) and decomposing it. An expression  $e$  is unrolled (without specifying an unrolling procedure  $U$ ) by unrolling it with respect to the unrolling procedures associated with each of its ADT atoms. A  $k$ -unrolling of an expression  $e$  is obtained by unrolling a  $(k - 1)$  unrolling of  $e$ .

For a first-order logic proof obligation  $P : \text{LHS} \Rightarrow \text{RHS}$ , we identify a  $k$ -unrolling of  $P$  (for a fixed unrolling parameter  $k$ ). After unrolling, we eliminate those decomposition clauses  $(p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i)$  whose path condition  $(p_1^i \wedge p_2^i)$  evaluates to **false** under the LHS ignoring all recursive relations. For example, the one-unrolling of  $P : \text{LHS} \Rightarrow \text{ls} \sim \text{Clist}_{\mathbf{m}}^{\text{lnode}}(0)$  after simplification yields  $P' : \text{LHS} \Rightarrow \text{ls is LNil}$ . Note that unlike  $P$ ,  $P'$  does not contain a recursive relation in its RHS. We categorize a proof obligation  $P$  based on this  $k$ -unrolled form of  $P$  as follows:<sup>8</sup>:

- Type I:  $k$ -unrolling of  $P$  does not contain recursive relations

---

<sup>7</sup>If  $a_1^i$  and  $a_2^i$  are ADT values, then we replace  $a_1^i = a_2^i$  with  $a_1^i \sim a_2^i$ .

<sup>8</sup>TODO: make this a para, not a footnote. In general, checking whether a path condition is **false** requires a proof discharge for each clause. Instead, we use a syntactic simplifier to identify these decomposition clauses. A more sophisticated approach would likely improve the completeness of the overall algorithm at the cost of performance. Nevertheless, this imprecise categorization is sound. Similarly, a higher value for the parameter  $k$  would increase completeness of categorization and the overall proof discharge algorithm.

---

- Type II:  $k$ -unrolling of  $P$  contains recursive relations only in the LHS
- Type III:  $k$ -unrolling  $P$  contains recursive relations in the RHS

### 3.7 Type I: $k$ -unrolling of $P$ does not contain recursive relations

In fig. 11, consider a proof obligation generated across the product-CFG edge  $(S0 : C0) \rightarrow (S3 : C3)$  while checking if the  $\textcircled{\text{I4}}$  invariant,  $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$ , 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  $\phi_{S0:C0} = (n_S = n_C)$  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,  $\text{LNil}$  is obtained for  $l_S$  and  $0$  (null) is obtained for  $l_C$ . The one-unrolled form of this proof obligation yields  $(n_S = n_C) \Rightarrow \text{LNil} \sim \text{LNil}$  which trivially resolves to true.

Consider another 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  $S$  to  $S0 \rightarrow S3 \rightarrow S5 \rightarrow S3$  here. In this case, the corresponding first-order logic condition evaluates to: “ $(n_S = n_C) \Rightarrow (\text{LCons}(0, \text{LNil}) \sim \text{Clist}_m^{\text{lnode}}(0))$ ”. One-unrolling of this proof obligation converts  $\text{Clist}_m^{\text{lnode}}(0)$  to  $\text{LNil}$ , and decomposes RHS into **false**. 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 invariant inference and correlation search procedures (as we will discuss later in sections 4.2 and 4.3).

Thus, we unify the structure and values of the Spec objects on both sides of the  $\sim$  operator (after  $k$  unrollings), and discharge the resulting proof obligations (that relate bitvector and array values) using an SMT solver. Please refer to **Chapter XXX** of the thesis for the intricacies of (a) translation of the formula to SMT logic and (b) reconstruction of counterexamples from the models returned by the SMT solver. Assuming a capable enough SMT solver, all proof obligations in Type I can be discharged precisely, i.e., we can always decide whether  $P$  evaluates to true or false. If it evaluates to false, we also obtain a counterexample.

```

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) => sum_list_impl(rest, sum + x).
A6:
A7: fn sum_list (l:List) : i32 = sum_list_impl(l, 0i32).

```

(a) Spec Program

```

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: }

```

(b) C Program

**Figure 12:** Spec and C Programs traversing a Linked List.

### 3.8 Type II: $k$ -unrolling of $P$ contains recursive relations only in the LHS

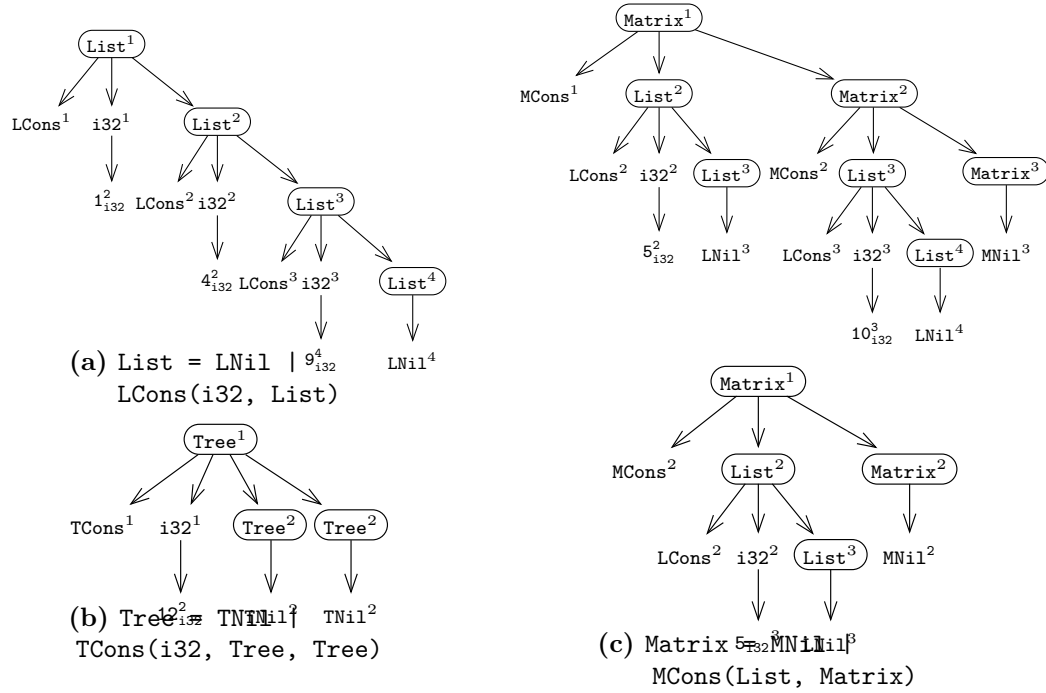
Consider the pair of programs in figs. 12a and 12b that traverse a list to compute the sum of all elements. The corresponding product-CFG and its node invariants that ensure observable equivalence are shown in figs. 9a and 9b.

Consider the proof obligation originating due to  $\textcircled{\text{I2}}$  invariant across edge  $(\text{S2:C2}) \rightarrow (\text{S2:C2})$  in fig. 12:  $\{\phi_{\text{S2:C2}}\}(\text{S2} \rightarrow \text{S5} \rightarrow \text{S2}, \text{C2} \rightarrow \text{C4} \rightarrow \text{C2})\{sum_S = sum_C\}$ , where the node invariant  $\phi_{\text{S2:C2}}$  contains the recursive relation  $l_S \sim Clist_m^{\text{lnode}}(l_C)$ . The corresponding (simplified) first-order logic condition for this proof obligation is:  $(l_S \sim Clist_m^{\text{lnode}}(l_C) \wedge sum_S = sum_C \wedge \neg(l_S \text{ is LNil}) \wedge l_C \neq 0) \Rightarrow ((sum_S + l_S.val) = (sum_C + l \xrightarrow{m}_{\text{lnode}} val))$ . We fail to remove the recursive relation on the LHS even after  $k$ -unrolling for any finite depth  $k$  because both sides of  $\sim$  represents list values of arbitrary length. In such a scenario, we do not know of an efficient SMT encoding for the recursive relation  $(l_S \sim Clist_m^{\text{lnode}}(l_C))$ . 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  $\textcircled{\text{R1}}$ .

Now, consider the proof obligation formed by correlating two iterations of the loop in program  $S$  with one iteration of the loop in program  $C$ ,  $\{\phi_{\text{S2:C2}}\}(\text{S2} \rightarrow \text{S5} \rightarrow \text{S2} \rightarrow \text{S5} \rightarrow \text{S2}, \text{C2} \rightarrow \text{C4} \rightarrow \text{C2})\{sum_S = sum_C\}$ . Similar to the last proof obligation, its equivalent first-order logic condition contains a recursive relation in the LHS. Clearly, this proof obligation is false. Whenever a proof obligation evaluates to false, we expect an ideal proof discharge algorithm to generate a counterexample that falsifies the proof obligation. Let's call this requirement  $\textcircled{\text{R2}}$ . Recall that such counterexamples help in faster convergence of our invariant inference and correlation algorithms.

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:  $(P_{\text{pre-o}} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\})$  and  $(P_{\text{pre-u}} : \{\phi_s^{u_{d_2}}\}(e)\{\phi_d\})$ . Here  $\phi_s^{o_{d_1}}$  and  $\phi_s^{u_{d_2}}$  represent the over- and under-approximated versions of precondition  $\phi_s$  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 the *depth of an ADT value*.



**Figure 13:** Parsed Trees with depths (shown as superscript) for three values, each of ADT type List, Tree, and Matrix respectively.

### 3.8.1 Depth of an ADT value

To define the depth of an ADT value, we view the ADT as a context-free grammar:

- The set of terminals consist of (a) scalar values, e.g., 42 for the `i32` type, and (b) the ADT value constructors (e.g., `LNil`, `LCons`).
- The set of non-terminals are all the scalar type identifiers (`i32`, `bool`, `unit`) and all the ADT type identifiers (e.g., `List`, `Tree`, `Matrix2D`).
- Productions of this context-free grammar specify the values that may be constructed for each type identifier: (a) For each basic type (e.g. `i32`), we add productions from its type identifier to all values of that type, e.g.,  $i32 \rightarrow 0 | 1 | 2 | \dots | (2^{32} - 1)$ ; (b) For an ADT, each constructor represents a separate production rule, e.g., ADT `List` has two productions: `List`  $\rightarrow$  `LNil` | `LCons i32 List`.



- The start non-terminal is the top-level ADT identifier (e.g., `List`).

In this context-free grammar interpretation of an ADT, a value of this ADT type can be viewed as a *parse tree* (also called a derivation tree) of the grammar. The *depth* of a node in this parse tree is the number of ADT identifiers (but not scalar type identifiers) in the path from the root node to the node representing that terminal value (both inclusive). Figure 13 shows examples of values of the `List`, `Tree`, and `Matrix` ADTs, and the depth values of the parse tree nodes. The *depth of an ADT value* is the maximum depth of any node in the parse tree of that value.

### 3.8.2 Overapproximation and Underapproximation

To overapproximate (underapproximate) a precondition  $\phi$ , each conjunctive recursive relation in  $\phi$  is overapproximated (underapproximated) individually.

The  $d$ -depth overapproximated version of a recursive relation  $l_1 \sim l_2$  is written as  $l_1 \sim_d l_2$ , where  $\sim_d$  represents the condition that the two ADT values  $l_1$  and  $l_2$  are *recursively equal up to depth  $d$* . i.e., all *terminals* at depth  $\leq d$  in the parse trees of both values are identical; however, terminals at depths  $> d$  can have different values.  $l_1 \sim_d l_2$  (for finite  $d$ ) is a weaker condition than  $l_1 \sim l_2$  (overapproximation);  $l_1 \sim l_2$  is equivalent to  $l_1 \sim_\infty l_2$ .

The  $d$ -depth underapproximated version 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 the two ADT values  $l_1$  and  $l_2$  are *recursively equal and bounded to depth  $d$* , i.e.,  $l_1, l_2$  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  $(l_1 \sim_d l_2) \wedge \Gamma_d(l_1) \wedge \Gamma_d(l_2)$ , 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  $l_1 \sim l_2$  (underapproximation) as it ensures both equality and max-depth of both values. For arbitrary depths  $a$  and  $b$  ( $a \leq b$ ), the approximate versions of recursive relation are related as follows:

$$l_1 \approx_a l_2 \Rightarrow l_1 \approx_b l_2 \Rightarrow l_1 \sim l_2 \Rightarrow l_1 \sim_b l_2 \Rightarrow l_1 \sim_a l_2$$

### 3.8.3 SMT encoding of overapproximate and underapproximate proof obligations

Unlike the original recursive relation  $l_1 \sim l_2$ ,  $l_1 \sim_d l_2$  and  $l_1 \approx_d l_2$  can be encoded using SMT as shown below:

- $l_1 \sim_d l_2$  is equivalent to the condition that the parse tree structures of the two values  $l_1$  and  $l_2$  (after  $d$ -unrolling) are isomorphic till depth  $d$  and the corresponding values in both ( $d$ -depth) isomorphic structures are also equal.  $l_1 \sim_d l_2$  can be identified through a  $d$ -depth bounded version of iterative unrolling and unification procedure described in section 3.5 followed by keeping only those correlation tuples that equate scalar expressions during decomposition. Along with the expressions and path conditions, we also keep track of the depth parameter which is incremented by one for each successfully unified ADT value constructors. We terminate early if the current depth is greater than  $d$ . In case both expressions are atoms, we eagerly unroll both sides and unify further as long as the current depth is within bounds.

For example, the condition  $l \sim_1 \text{Clist}_m^{\text{lnode}}(\mathbf{p})$  can be computed through one-depth bounded iterative unrolling and unification. During decomposition, keeping only correlation tuples that equate scalar expressions, the condition above reduces to the SMT-encodable predicate:

$$((l \text{ is LNil}) \Leftrightarrow (\mathbf{p} == 0)) \wedge (\neg(l \text{ is LNil}) \Rightarrow (l.\text{val} = p \xrightarrow{m}_{\text{lnode}} \text{val}))$$

- $\Gamma_d(l)$  is equivalent to the condition that the parse-tree nodes at depths  $> d$  are unreachable. This is achieved by unrolling a recursive relation till depth  $d$  and then asserting the unreachability of if-then-else paths that reach nodes with depth  $> d$  (by checking the satisfiability of their expression path conditions). For example, for a **List** value  $l$ , the condition  $\Gamma_2(l)$  is equivalent to  $(l \text{ is LNil}) \vee (\neg(l \text{ is LNil}) \wedge (l.\text{tail} \text{ is LNil}))$ . Similarly,  $\Gamma_2(\text{Clist}_m^{\text{lnode}}(\mathbf{p}))$  is equivalent to  $(\mathbf{p} = 0) \vee (\neg(\mathbf{p} \neq 0) \wedge (\mathbf{p} \xrightarrow{m}_{\text{lnode}} \text{next} = 0))$ .

### 3.8.4 Proof discharge algorithm for Type II proof obligations

Thus, for a *Type II* proof obligation  $P : \{\phi_s\}(e)\{\phi_d\}$ , we first submit the proof obligation  $(P_{\text{pre-o}} : \{\phi_s^{o_{d1}}\}(e)\{\phi_d\})$  to the SMT solver. Recall that the precondition  $\phi_s^{o_{d1}}$  is the overapproximated version of  $\phi_s$ . If the SMT solver evaluates  $P_{\text{pre-o}}$  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  $P_{\text{pre-o}}$  to false, then we submit the proof obligation  $(P_{\text{pre-u}} : \{\phi_s^{u_{d2}}\}(e)\{\phi_d\})$  to the SMT solver. Recall that the precondition  $\phi_s^{u_{d2}}$  is the underapproximated version of  $\phi_s$ . If the SMT solver evaluates  $P_{\text{pre-u}}$  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  $P_{\text{pre-u}}$  would also falsify  $P$ , and is thus usable in invariant inference and correlation procedures.

Finally, if the SMT solver evaluates  $P_{\text{pre-u}}$  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 a counterexample). Note that both approximations of  $P$  strictly fall in Type I and are discharged as discussed in section 3.7. Revisiting our examples, the proof obligation  $\{\phi_{\text{S2:C2}}\}(\text{S2} \rightarrow \text{S5} \rightarrow \text{S2}, \text{C2} \rightarrow \text{C4} \rightarrow \text{C2}) \{sum_S = sum_C\}$  is provable using a depth-1 overapproximation of the precondition  $\phi_{\text{S2:C2}}$  — the depth-1 overapproximation retains the information that the first value in lists  $l_S$  and  $l_C$  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  $\phi_{\text{S2:C2}}$ ).

Similarly, the proof obligation  $\{\phi_{\text{S2:C2}}\}(\text{S2} \rightarrow \text{S5} \rightarrow \text{S2} \rightarrow \text{S5} \rightarrow \text{S2}, \text{C2} \rightarrow \text{C4} \rightarrow \text{C2}) \{sum_S = sum_C\}$  evaluates to false (with a counterexample) using a depth-2 underapproximation of the precondition  $\phi_{\text{S2:C2}}$ . In the depth-2 underapproximate version, we try to prove that if the equal lists  $l_S$  and  $\text{Clist}_m^{\text{lnode}}(l_C)$  have exactly two nodes<sup>9</sup>, then the sum of the values in the two nodes of  $l_S$  is equal to the value

---

<sup>9</sup>The underapproximation restricts both lists to have at most two nodes; the path condition for  $\text{S2} \rightarrow \text{S5} \rightarrow \text{S2} \rightarrow \text{S5} \rightarrow \text{S2}$  additionally restricts  $l_S$  to have at least two nodes; together, this is equivalent to the list having exactly two nodes

---

stored in the first node in  $l_C$ . This proof obligation will return a counterexample that maps program variables to their concrete values. We show a possible counterexample to this proof obligation below.

$$\left. \begin{array}{l} \text{sum}_S \mapsto 3 \quad \text{sum}_C \mapsto 3 \\ l_S \mapsto \text{LCons}(42, \text{LCons}(43, \text{LNil})) \\ l_C \mapsto 0x123 \end{array} \right| m \mapsto \left( \begin{array}{l} 0x123 \mapsto_{\text{lnode}} (.value \mapsto 42, .next \mapsto 0x456), \\ 0x456 \mapsto_{\text{lnode}} (.value \mapsto 43, .next \mapsto 0x000), \\ () \mapsto 77 \end{array} \right)$$

This counterexample maps variables to values (e.g.,  $\text{sum}_C$  maps to an `i32` value 3 and  $l_S$  maps to a `List` value  $\text{LCons}(42, \text{LCons}(43, \text{LNil}))$ ). It also maps the C program's memory state  $m$  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 `value` and `next` fields shown for each object). For all other addresses (except the ones for which an explicit mapping is available),  $m$  maps them to the default byte-value 77 (shown as  $() \mapsto 77$ ) in this counterexample.

This counterexample satisfies the preconditions  $l_S \approx_2 \text{Clist}_m^{\text{lnode}}(l_C)$  and  $\text{sum}_S = \text{sum}_C$ . Further, when the paths  $(S2 \rightarrow S5 \rightarrow S2 \rightarrow S5 \rightarrow S2, C2 \rightarrow C4 \rightarrow C2)$  are executed starting at the machine state represented by this counterexample, the resulting values of  $\text{sum}_S$  and  $\text{sum}_C$  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).

### 3.9 Type III: $k$ -unrolling $P$ contains recursive relations in the RHS

In fig. 11, consider a proof obligation generated across the product-CFG edge  $(S3 : C5) \rightarrow (S3 : C3)$  while checking if the  $\textcircled{I4}$  invariant,  $l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)$ , holds at  $(S3 : C3) : \{\phi_{S3:C5}\}(S3 \rightarrow S5 \rightarrow S3, C5 \rightarrow C3)\{l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)\}$ . Here, a recursive relation is present both in the precondition  $\phi_{S3:C5}$  ( $\textcircled{I8}$ ) and in the postcondition ( $\textcircled{I4}$ ) 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):

$$(i_S = i_C \wedge p_C = \text{malloc}() \wedge l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)) \Rightarrow (\text{LCons}(i_S, l_S) \sim \text{Clist}_{m'}^{\text{lnode}}(p_C)) \quad (7)$$

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

$$m' \equiv m[\&(p_C \xrightarrow{m}_{\text{lnode}} \text{value}) \leftarrow i_C]_{i32}[\&(p_C \xrightarrow{m}_{\text{lnode}} \text{next}) \leftarrow l_C]_{i32} \quad (8)$$

Here,  $m[a \leftarrow v]$  represents an array that is equal to  $m$  everywhere except at address  $a$  which contains the value  $v$ . We also refer to these memory writes that distinguish  $m$  from  $m'$ , the *distinguishing writes*.

### 3.9.1 LHS-to-RHS Substitution and RHS Decomposition

S2C utilizes the  $\sim$  relationships in the LHS (antecedent) of “ $\Rightarrow$ ” to rewrite eq. (7) so that the recursive `List` values in its RHS (consequent) are substituted with the lifted  $C$  values (lifted using the `Clist` constructor). Thus, we rewrite eq. (7) to:

$$(i_S = i_C \wedge p_C = \text{malloc}() \wedge l_S \sim \text{Clist}_m^{\text{lnode}}(l_C)) \Rightarrow (\text{LCons}(i_S, \text{Clist}_m^{\text{lnode}}(l_C)) \sim \text{Clist}_{m'}^{\text{lnode}}(p_C)) \quad (9)$$

Next, we decompose the RHS by decomposing the recursive relation in the RHS followed by RHS-breaking. This process reduces eq. (7) into the following smaller proof obligations (showing only the RHS, the LHS is the same as in eq. (7)): (1)  $\neg(p_C = 0)$ , (2)  $\neg(p_C = 0) \rightarrow i_S = (p_C \xrightarrow{m'}_{\text{lnode}} \text{value})$ , and (3)  $\neg(p_C = 0) \rightarrow \text{Clist}_m^{\text{lnode}}(l_C) \sim \text{Clist}_{m'}^{\text{lnode}}(p_C \xrightarrow{m'}_{\text{lnode}} \text{next})$ . The first two proof obligations fall in *Type II* and are discharged through over- and under-approximation schemes (as discussed in section 3.8):

1. The first proof obligation with postcondition  $\neg(p_C = 0)$  evaluates to true because the LHS ensures that  $p_C$  is the return value of an allocation function (`malloc`) which must be non-null due to the ( $C$  fits) assumption.
2. The second proof obligation with postcondition  $(i_S = (p_C \xrightarrow{m'}_{\text{lnode}} \text{value}))$  also evaluates to true because  $i_C$  is written to address  $\&(p_C \xrightarrow{m'}_{\text{lnode}} \text{value})$  in  $m'$  (eq. (8)) and the LHS ensures that  $i_S = i_C$ .

For ease of exposition, we simplify the postcondition of the third proof obligation from  $\neg(p_C = 0) \rightarrow (\text{Clist}_m^{\text{lnode}}(l_C) \sim \text{Clist}_{m'}^{\text{lnode}}(p_C \xrightarrow{m'}_{\text{lnode}} \text{next}))$  to

---

$(\text{Clist}_m^{\text{lnode}}(\text{l}_C) \sim \text{Clist}_{m'}^{\text{lnode}}(\text{l}_C))$ . This simplification is valid because  $\text{l}_C$  is written to address  $\&(\text{p}_C \xrightarrow{m'} \text{lnode next})$  in  $m'$  (eq. (8)). Also, we have already shown that  $\neg(\text{p}_C = 0)$  holds. Thus, the third proof obligation can be rewritten as a recursive relation between two lifted expressions<sup>10</sup>:

$$\text{Clist}_m^{\text{lnode}}(\text{l}_C) \sim \text{Clist}_{m'}^{\text{lnode}}(\text{l}_C) \quad (10)$$

Hence, we are interested in proving equality between two **List** values in  $C$  under different memory states  $m$  and  $m'$ . Next, we show how the above can be reformulated to the problem of showing equivalence between two procedures through bisimulation.

### 3.9.2 Conversion of recursive equality between lifted expressions to a bisimulation

Consider a program that recursively calls the unrolling procedure in eq. (4) to deconstruct  $\text{Clist}_m^{\text{lnode}}(\text{l}_C)$ . For example,  $\text{Clist}_m^{\text{lnode}}(\text{l}_C)$  may yield a recursive call to the unrolling procedure  $\text{Clist}_m^{\text{lnode}}(\text{l}_C \xrightarrow{m} \text{lnode next})$  and so on, until the argument to the unrolling procedure becomes zero. This program essentially deconstructs  $\text{Clist}_m^{\text{lnode}}(\text{l}_C)$  into its terminal (scalar) values and reconstructs a **List** value equal to the value represented by  $\text{Clist}_m^{\text{lnode}}(\text{l}_C)$ . We call this program a *reconstruction program* based on the unrolling procedure of  $\text{Clist}_m^{\text{lnode}}(\text{l}_C)$ .

**Theorem 1.** *Under the antecedent  $(\text{l}_S \sim \text{Clist}_m^{\text{lnode}}(\text{l}_C))$ :*

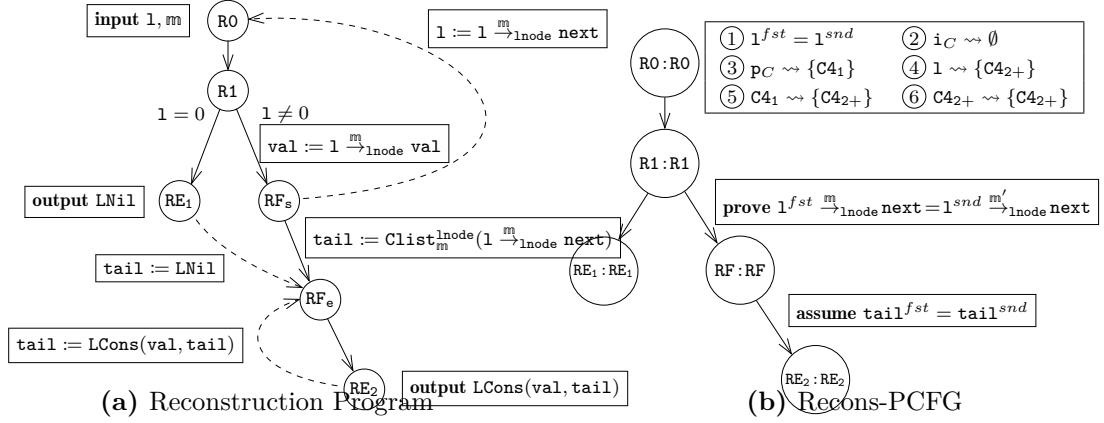
*$(\text{Clist}_m^{\text{lnode}}(\text{l}_C) \sim \text{Clist}_{m'}^{\text{lnode}}(\text{l}_C))$  holds iff a bisimulation relation exists between the reconstruction programs based on  $\text{Clist}_m^{\text{lnode}}(\text{l}_C)$  and  $\text{Clist}_{m'}^{\text{lnode}}(\text{l}_C)$ . The bisimulation relation must ensure that the observables generated by both procedures are identical.*

*Proof.* The “if” case of this “iff” relation follows from noting that the observables of a reconstruction program are the generated **List** values. Thus, a successful bisim-

---

<sup>10</sup>This simplification-based rewriting is only shown for ease of exposition, and has no effect on the operation of the algorithm. Even if the proof obligation is not simplified, the unification-based proof discharge algorithm will generate proof conditions of the form  $\neg(\text{p}_C = 0) \Rightarrow ((\text{p}_C \xrightarrow{m'} \text{lnode next}) = \text{l}_C)$  which will be successfully discharged by the SMT solver.

---



**Figure 14:** The reconstruction program for  $Clist_m^{lnode}(1_C)$  and recons-PCFG between reconstruction programs of  $Clist_m^{lnode}(1_C)$  and  $Clist_{m'}^{lnode}(1_C)$ . In fig. 14a,  $DO$  represents the unrolling procedure entry node, and the square boxes show the transfer functions of the unrolling procedure (eq. (4)). The dashed edges represent a recursive function call. In fig. 14b, the square box to the right of node  $DO:DO$  contains the inferred invariants for this recons-PCFG.

ulation check ensures equal `List` values upon termination. Termination follows from the antecedent because `Spec` values (such as  $l_S$ ) must be finite.

The “only if” case follows from the unification of the unrolling procedure (in eq. (4)) for  $Clist_m^{lnode}(1_C)$  and  $Clist_{m'}^{lnode}(1_C)$ .

Thus, to check if  $Clist_m^{lnode}(1_C) \sim Clist_{m'}^{lnode}(1_C)$ , we check if a bisimulation exists between the two respective reconstruction programs (potentially under an antecedent). Theorem 1 generalizes to equality of arbitrary lifted expressions constructed from potentially different  $C$  values and memory states.

### 3.9.3 Checking bisimulation between reconstruction programs

To check bisimulation, we attempt to show that both reconstructions 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 — to distinguish this product-CFG from the top-level product-CFG that relates  $S$  and  $C$ , we call this product-CFG that relates two reconstruction programs, a *reconstruction product-CFG* or *recons-PCFG* for short.

The reconstruction program and the recons-PCFG for our `Clist` example are shown in fig. 14. To check bisimulation between the programs that deconstruct

$\text{Clist}_m^{\text{lnode}}(\mathbf{l}_C)$  and  $\text{Clist}_{m'}^{\text{lnode}}(\mathbf{l}_C)$ , the recons-PCFG correlates one unrolling of the first program with one unrolling of the second program. An unrolling of each reconstruction program is based on the unrolling procedure in eq. (4). Thus, the PC-transition correlations of both programs are trivially obtained by unifying the unrolling procedure with itself. A node is created in the recons-PCFG that encodes the correlation of the entries of the unrolling procedure in both programs, we call this node the *recursive-node* in the recons-PCFG, e.g., the recursive node in fig. 14b is  $\text{R0}:\text{R0}$ . A recursive call becomes a back-edge in the recons-PCFG that terminates at the recursive-node. A candidate invariant at the recursive-node is obtained by equating the pair of corresponding  $\mathbf{l}_C$  variables across the first and second programs, i.e.,  $\mathbf{l}_C^{fst} = \mathbf{l}_C^{snd}$ . At the start of both reconstruction programs,  $\mathbf{l}_C^{fst} = \mathbf{l}_C^{snd} = \mathbf{l}_C^{start}$  — the same  $\mathbf{l}_C^{start}$  is passed to both reconstruction programs, only the memory states  $m$  and  $m'$  are different. The bisimulation check thus involves checking that if the invariant  $\mathbf{l}_C^{fst} = \mathbf{l}_C^{snd}$  holds at the recursive-node, then during one iteration of the unrolling procedure in both programs:

1. The if condition ( $\mathbf{l}_C^{fst} = 0$ ) in the first program is equal to the corresponding if condition ( $\mathbf{l}_C^{snd} = 0$ ) in the second program.
2. If the if condition evaluates to false in both programs, then the observable values (that are used in the construction of the list) are equal, i.e.,  $((\mathbf{l}_C^{fst} \neq 0) \wedge (\mathbf{l}_C^{snd} \neq 0)) \Rightarrow (\mathbf{l}_C^{fst} \xrightarrow{m}_{\text{lnode}} \text{val} = \mathbf{l}_C^{snd} \xrightarrow{m'}_{\text{lnode}} \text{val})$ .
3. If the if condition evaluates to false in both programs, then the invariant holds at the beginning of the unrolling procedure invoked through the recursive call. This involves checking equality of the arguments to the recursive call, i.e.,  $((\mathbf{l}_C^{fst} \neq 0) \wedge (\mathbf{l}_C^{snd} \neq 0)) \Rightarrow \mathbf{l}_C^{fst} \xrightarrow{m}_{\text{lnode}} \text{next} = \mathbf{l}_C^{snd} \xrightarrow{m'}_{\text{lnode}} \text{next}$ .

The first check succeeds due to the invariant  $\mathbf{l}_C^{fst} = \mathbf{l}_C^{snd}$ . For the second and third checks, we additionally need to reason that the memory objects  $\mathbf{l}_C^{fst} \xrightarrow{m}_{\text{lnode}} \text{val}$  and  $\mathbf{l}_C^{fst} \xrightarrow{m}_{\text{lnode}} \text{next}$  cannot alias with the writes (in  $m'$  in eq. (8)) to the newly allocated objects  $\text{p}_C \xrightarrow{m}_{\text{lnode}} \text{val}$  and  $\text{p}_C \xrightarrow{m}_{\text{lnode}} \text{next}$ . This aliasing information is captured using a points-to analysis, described next in section 3.9.4.

Notice that a bisimulation check between the reconstruction programs is significantly easier than the top-level bisimulation check between Spec and C programs:



here, the correlation of PC transitions is trivially identified by unifying the unrolling procedure with itself, and the candidate invariants are obtained by equating each corresponding pair of variables across the two programs.

#### 3.9.4 Points-to Analysis

To reason about aliasing (as required during the bisimulation check in section 3.9.3), we conservatively compute the *may-point-to* information for each program value using Andersen’s algorithm [10]. The range of this computed may-point-to function are *sets 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 the *allocation sites* of the program, i.e., PCs where a call to `malloc` exists, e.g., **C4** in fig. 2b is an allocation site. For each allocation site  $A$ , we create two region labels: (1) the first region label, called  $A_1$ , identifies the set of memory objects that were allocated by the most recent execution of  $A$ . (2) The second region label, called  $A_{2+}$ , identifies the set of memory objects that were allocated by older (not the most recent) executions of  $A$ .

For example, at the start of PC **C7** in fig. 2b,  $i_C \rightsquigarrow \emptyset$ ,  $n_C \rightsquigarrow \{\mathbf{C4}_1\}$ , and  $l_C \rightsquigarrow \{\mathbf{C4}_{2+}\}$ . Because the may-point-to analysis determines the sets of objects pointed-to by  $n_C$  and  $l_C$  to be disjoint ( $\{\mathbf{C4}_1\}$  vs.  $\{\mathbf{C4}_{2+}\}$ ), any memory accesses through  $n_C$  and  $l_C$  cannot alias at **C7** (for an access offset that is within the bounds of the allocation size ‘`sizeof lnode`’).

The may-point-to information is computed not just for scalar program values ( $n_C, l_C, \dots$ ) but also for each region label. For region labels  $A1_{r1}, A2_{r2}, A3_{r3}$ :  $A1_{r1} \rightsquigarrow \{A2_{r2}, A3_{r3}\}$  represents the condition that the values (pointers) stored in objects identified by  $A1_{r1}$  may point to an object identified by either  $A2_{r2}$  or  $A3_{r3}$  (but not to any object outside  $A2_{r2}$  and  $A3_{r3}$ ). In fig. 2b, at PC **C7**, we get  $\mathbf{C4}_1 \rightsquigarrow \{\mathbf{C4}_{2+}\}$  and  $\mathbf{C4}_{2+} \rightsquigarrow \{\mathbf{C4}_{2+}\}$ . The condition  $\mathbf{C4}_1 \rightsquigarrow \{\mathbf{C4}_{2+}\}$  holds because the `next` pointer of the object pointed-to by  $n_C$  (which is a  $\mathbf{C4}_1$  object) may point

---

to a  $\mathbf{C4}_{2+}$  object (e.g., object pointed-to by  $1_C$ ). Similarly,  $\mathbf{C4}_{2+} \rightsquigarrow \{\mathbf{C4}_{2+}\}$  says that a pointer within a  $\mathbf{C4}_{2+}$  object may point to a  $\mathbf{C4}_{2+}$  object (but not to a  $\mathbf{C4}_1$  object).

### 3.9.5 Transferring points-to information to the recons-PCFG

Recall that in section 3.9.2, we reduce a validity check of the condition  $\mathbf{Clist}_m^{\text{lnode}}(1_C) \sim \mathbf{Clist}_{m'}^{\text{lnode}}(1_C)$  to a bisimulation check. Also, recall that we discharge the bisimulation check through the construction of a recons-PCFG that compares the unrolling procedure with itself (executing on memory states  $m$  and  $m'$ ). During this bisimulation check, we need to prove that for each execution of the unrolling procedure,  $1_C \xrightarrow{m}_{\text{lnode}} \{\mathbf{val}, \mathbf{next}\}$  and  $1_C \xrightarrow{m'}_{\text{lnode}} \{\mathbf{val}, \mathbf{next}\}$ <sup>11</sup> are equal. To successfully discharge these proof obligations, it suffices to show  $1_C$  cannot alias with the memory writes that distinguish  $m$  from  $m'$ .

Our points-to analysis on the C program determines that at PC C5 (the start of the product-CFG edge  $(\mathbf{S3}:\mathbf{C5}) \rightarrow (\mathbf{S3}:\mathbf{C3})$  across which the proof condition is being evaluated), the pointer to the *head* of the list, i.e.,  $1_C^{\text{start}}$  points to  $\mathbf{C4}_{2+}$ . It also determines that the distinguishing writes modify memory regions belonging to  $\mathbf{C4}_1$ . Further, we get  $\mathbf{C4}_{2+} \rightsquigarrow \{\mathbf{C4}_{2+}\}$  at PC C5. 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,  $1_C \rightsquigarrow \{\mathbf{C4}_{2+}\}$ , at the recursive-node of the recons-PCFG (shown in fig. 14b). To see why  $1_C \rightsquigarrow \{\mathbf{C4}_{2+}\}$  is an inductive invariant at the recursive-node:

- (Base case) The invariant holds at entry to the recons-PCFG (because it holds for  $1_C^{\text{start}}$ ).
- (Induction step) If  $1_C \rightsquigarrow \{\mathbf{C4}_{2+}\}$  holds at the start of an unrolling procedure, it also holds at the start of a recursive call to the unrolling procedure. This follows from  $\mathbf{C4}_{2+} \rightsquigarrow \{\mathbf{C4}_{2+}\}$  (points-to information at PC C5), which ensures that  $1_C \rightarrow \mathbf{next}$  may point to only  $\mathbf{C4}_{2+}$  objects.

---

<sup>11</sup>Here, we use the symbol  $1_C$  to refer to equal values  $1_C^{\text{fst}}$  and  $1_C^{\text{snd}}$ .

---

To identify this points-to invariant, we run our points-to analysis (the same analysis that is run on the C program) on the reconstruction programs (fig. 14a) before comparing them for equivalence. The boundary condition for the points-to analysis at the entry node of the reconstruction program (e.g., R0 in fig. 14) is based on the results of the points-to analysis on  $C$  at the PC where the proof obligation is being discharged (e.g., C5 in our fig. 1b). The points-to invariants at a node  $(R_i^{fst}, R_j^{snd})$  of a recons-PCFG are derived from the results of the points-to analysis on the individual reconstruction programs at nodes  $R_i^{fst}$  and  $R_j^{snd}$  respectively.

During proof obligation discharge (e.g., during the bisimulation check on recons-PCFG), the points-to invariants are encoded as SMT constraints. This allows us to successfully complete the bisimulation proof on the recons-PCFG, and consequently successfully discharge the proof obligation  $\{\phi_{S3:C5}\}$   $(S3 \rightarrow S5 \rightarrow S3, C5 \rightarrow C3)\{l_S \sim Clist_m^{lnode}(l_C)\}$  in table 2. The points-to analysis is described more formally in section 4.5.

### 3.9.6 Proof discharge algorithm for Type III obligations

Before the start of an equivalence check, a points-to analysis is run on the  $C$  IR once. During the equivalence check, to discharge a Type III proof obligation  $P : LHS \Rightarrow RHS$  (expressed in first-order logic), we first replace the recursive values of program  $S$  in the RHS with lifted C values, based on the equalities present in the LHS, to obtain  $P_2$ . This is followed by decomposition and RHS-breaking of  $P_2$ .

Upon successful decomposition, we obtain several smaller proof obligations. To prove  $P$ , we require all these smaller proof obligations to be provable. If any of these smaller proof obligations is not provable, we are unable to prove  $P$ . If we obtain a counterexample to any of these smaller proof obligations, then that counterexample also falsifies  $P$ . Let  $P_3$  represent any such smaller proof obligation. RHS of  $P_3$ , being a decomposition clause, must relate atomic expressions on the RHS. If  $P_3$  relates two scalar values in the RHS, then it is a Type II proof obligation and can be discharged using the algorithm in section 3.8.4.

If  $P_3$  relates two lifted expressions in the RHS, we check if the reconstruction programs of the two lifted ADT values being compared can be proven to be bisimilar (assuming that LHS of  $P_3$  holds at the correlated entry nodes in the recons-PCFG).

To improve the bisimulation check's precision, we transfer the points-to information of the  $C$  program (at the PC where the proof obligation is being discharged) to the entry of the reconstruction programs. The same points-to analysis is ran on the reconstruction programs to populate the points-to function at all PCs.

These queries generated by a bisimulation check are discharged by a recursive call to the proof discharge procedure. The depth of these recursive calls to the proof discharge procedure is determined by the maximum *recursion nest depth* (similar to loop nest depth) of the decomposition program.

If the bisimilarity check succeeds, the proof procedure returns true for  $P$ . If the bisimilarity check fails, we imprecisely return false for  $P$  (without a counterexample).

Finally, if  $P_3$  neither relates two scalar values, nor relates two lifted expressions, we attempt to prove that LHS of  $P_3$  imply **false**. If successfully disproven, we return false for  $P$  with the counterexamples. Otherwise, we imprecisely return false for  $P$  (without a counterexample).

Please refer to **Chapter XXX** of the thesis for a detailed discussion on the algorithms introduced in this section along with their pseudo-code.

## 4 Formalism

### 4.1 The Spec Language

We briefly discuss the properties of the Spec language in this section. Spec supports recursive algebraic data types (ADT) similar to the ones available in most functional languages. The types in Spec can be represented in *first order recursive types* with **Product** and **Sum** type constructors and **Unit**, **Bool**,  $i\langle N \rangle$  types (i.e., nullary type constructors) as follows:

$$T \rightarrow \mu\alpha \mid \text{Product}(T, \dots, T) \mid \text{Sum}(T, \dots, T) \mid \text{Unit} \mid \text{Bool} \mid i\langle N \rangle \mid \alpha$$

For example, the **List** type can be written as  $\mu\alpha.\text{Sum}(\text{Unit}, \text{Prod}(i32, \alpha))$ .

The language also borrows its expression grammar heavily from functional languages. This includes the usual constructs like **let-in**, **if-then-else**, function

---

application and the **match** statement for pattern-matching (i.e. deconstructing) sum and product values. Unlike functional languages, Spec only supports first order functions. Also, Spec does not support partial function application. Hence, we constrain our attention to C programs containing only first order functions. Spec is equipped with a special **assuming-do** construct for explicitly providing UB conditions. These assumptions become part of  $(S \text{ def})$  as discussed in section 2.6. Spec also provides the typical boolean and bitvector operators for expressing computation in C succinctly yet explicitly. This includes logical operators (e.g., **and**), bitvector arithmetic operators (e.g., **bvadd(+)**) and relational operators for comparing bitvectors interpreted as signed or unsigned integers (e.g.,  $\leq_{u,s}$ ).

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

**Figure 15:** Simplified expression grammar of Spec language

## 4.2 Counterexample-guided Best-First Search Algorithm for a Product-CFG

S2C constructs a product-CFG incrementally to search for an observably-equivalent bisimulation relation between the individual CFGs of a Spec program  $S$  and a C program  $C$ . Multiple candidate product-CFGs are partially constructed during this search; the search completes when one of these candidates yields an

equivalence proof.

*Anchor nodes* in the CFG of the  $C$  program are identified to ensure that every cycle in the CFG contains at least one anchor node. Also, for every procedure call in the CFG, anchor nodes are created just before and just after the callsite, e.g., in fig. 10b, **C4** and **C5** are anchor nodes around the call to `malloc()`. Our algorithm ensures that for each anchor node in  $C$ , we identify a correlated node in  $S$  — if a product-CFG  $\pi$  contains a product-CFG node  $(n_S, n_C)$ , then  $\pi$  correlates node  $n_C$  in  $C$  with node  $n_S$  in  $S$ . The first partially-constructed product-CFG contains a single entry node that encodes the correlation of the entry nodes (**S0:C0**) of the two input CFGs.

At each step of the incremental construction algorithm, a node  $(n_S, n_C)$  is chosen in a product-CFG  $\pi$  and a path  $\rho_C$  in  $C$ 's CFG starting at  $n_C$  (and ending at an anchor node in  $C$ ) is selected. Then, the potential correlations  $\rho_C$  with paths in  $S$ 's CFG are enumerated. For example, in fig. 11, at product-CFG node (**S3:C3**), we first select the  $C$  path **C3**→**C4**, and its potential correlation possibilities with paths  $\epsilon$ , **S3** → **S5**, **S3** → **S5** → **S3**, **S3** → **S5** → **S3** → **S5**, ... in  $S$  are enumerated (up to an unroll factor  $\mu$ ).

For each enumerated correlation possibility  $(\rho_S, \rho_C)$ , a separate product-CFG  $\pi'$  is created (by cloning  $\pi$ ) and a new product-CFG edge  $e = (\rho_S, \rho_C)$  is added to  $\pi'$ . The head of the product-CFG edge  $e$  is the (potentially newly added) product-CFG node representing the correlation of the end-points of paths  $\rho_S$  and  $\rho_C$ . For example, the node (**S3:C4**) is added to the product-CFG if it correlates paths  $\epsilon$  and **C3** → **C4** starting at (**S3:C3**). For each node  $s$  in a product-CFG  $\pi$ , we maintain a small number of concrete machine state pairs (of  $S$  and  $C$ ) at  $s$ . The concrete machine state pairs at  $s$  are obtained as counterexamples to an unsuccessful proof obligation  $\{\phi_s\}(s \rightarrow d)\{\phi_d\}$  (for some edge  $s \rightarrow d$  and node  $d$  in  $\pi$ ). Thus, by construction, these counterexamples represent concrete state pairs that may potentially occur at  $s$  during the lockstep execution encoded by  $\pi$ .

To evaluate the promise of a possible correlation  $(\rho_S, \rho_C)$  starting at node  $s$  in product-CFG  $\pi$ , we examine the execution behavior of the counterexamples at  $s$  on the product-CFG edge  $e = (s \rightarrow d) = (\rho_S, \rho_C)$ . 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 [22]. A best-first search (BFS)

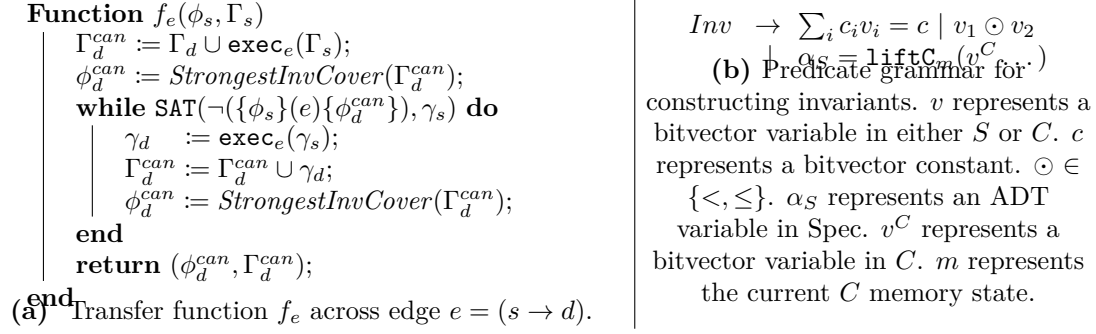
procedure based on this ranking criterion is used to incrementally construct a product-CFG that proves bisimulation. For each intermediate candidate product-CFG  $\pi$  generated during this search procedure, an automatic invariant inference procedure is used to identify invariants at all the nodes in  $\pi$ . The counterexamples obtained from the proof obligations created by this invariant inference procedure are added to the respective nodes in  $\pi$ ; these counterexamples help rank future correlations starting at those nodes.

If after invariant inference, we realize that an intermediate candidate product-CFG  $\pi_1$  is not promising enough, we backtrack and choose another candidate product-CFG  $\pi_2$  and explore the potential correlations that can be added to  $\pi_2$ . Thus, a product-CFG is constructed one edge at a time. If at any stage, the invariants inferred for a product-CFG  $\pi_i$  ensure equal observables, we have successfully demonstrated equivalence.

This counterexample-guided BFS procedure is similar to the one described in prior work on the Counter algorithm [22]. Our primary contribution is a proof discharge algorithm for proof obligations containing recursive relations (sections 3.3, 3.4 and 3.7 to 3.9). These proof obligations may be generated either at the intermediate (search) or the final (check) phases of the BFS procedure.

**Table 3:** Dataflow formulation for the Invariant Inference Algorithm.

Domain	$\left\{ \begin{array}{l} \phi_n \text{ is a conjunction of predicates drawn from} \\ \text{grammar in 16b, } \Gamma_n \text{ is a set of counterexamples} \end{array} \right\}$
Direction	Forward
Transfer function across edge $e = (s \rightarrow d)$	$(\phi_d, \Gamma_d) = f_e(\phi_s, \Gamma_s)$ (fig. 16a)
Meet operator $\otimes$ $(\phi_n, \Gamma_n) \leftarrow (\phi_n^1, \Gamma_n^1) \otimes (\phi_n^2, \Gamma_n^2)$	$\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2, \quad \phi_n \leftarrow \text{StrongestInvCover}(\Gamma_n)$
Boundary condition	$\text{out}[n^{start}] = (Pre, \Gamma_{n^{start}})$
Initialization to $\top$	$\text{in}[n] = (\text{False}, \{\})$ for all non-start nodes



**Figure 16:** Transfer function  $f_e$  and Predicate grammar  $Inv$  for invariant inference dataflow analysis in table 3. Given invariants  $(\phi_s)$  and counterexamples  $(\Gamma_s)$  at node  $s$ ,  $f_e$  returns the updated invariants  $(\phi_d)$  and counterexamples  $(\Gamma_d)$  at node  $d$ .

$\mathbf{StrongestInvCover}(\Gamma)$  computes the strongest invariant cover for counterexamples  $\Gamma$ .  $\mathbf{exec}_e(\Gamma)$  (concretely) executes counterexamples  $\Gamma$  over edge  $e$ .  $\mathbf{SAT}(\phi, \gamma)$  determines the satisfiability of  $\phi$ ; if satisfiable, the models (counterexamples) are returned in output parameter  $\gamma$ .

### 4.3 Invariant Inference and Counterexample Generation

Table 3 presents our dataflow analysis for inferring invariants  $\phi_n$  at each node  $n$  of a product-CFG, while also generating a set of counterexamples  $\Gamma_n$  at node  $n$  that represents the potential concrete machine states at  $n$ .

Given the invariants and counterexamples at node  $s$   $(\phi_s, \Gamma_s)$ , the transfer function initializes the new candidate set of counterexamples at  $d$  ( $\Gamma_d^{can}$ ) to the current set of counterexamples at  $d$  ( $\Gamma_d$ ) union-ed with the counterexamples obtained by executing  $\Gamma_s$  on edge  $e$  ( $\mathbf{exec}_e$ ). The candidate invariant at  $d$  ( $\phi_d^{can}$ ) is computed as the strongest cover of  $\Gamma_d^{can}$  ( $\mathbf{StrongestInvCover}()$ ). At each step, the transfer function attempts to prove  $\{\phi_s\}(e)\{\phi_d^{can}\}$  (by checking SATisfiability of its negation). If the proof succeeds, the candidate invariant  $\phi_d^{can}$  is returned alongwith the counterexamples  $\Gamma_d^{can}$  learned so far. Else the candidate invariant  $\phi_d^{can}$  is weakened using the counterexamples obtained from the SAT query ( $\gamma$ ) and the proof attempt is repeated.

The predicate grammar allows the automatic inference of affine and inequality relations between bitvector values of both programs, as well as, recursive relations between an ADT value in Spec ( $\alpha_S$ ) and a *lifted* ADT value from C ( $\mathbf{liftC}_m(p_C)$ ). We enumerate these recursive relation guesses for all bitvector variables  $v^C$  in  $C$



and candidate **liftC** lifting constructor. In our implementation, the candidate **liftC** constructors are derived from the constructors present in the precondition *Pre* and the postcondition *Post*. More sophisticated strategies for automatic guessing of these lifting constructors are possible.

*StrongestInvCover()* for affine relations involves identifying the basis vectors of the kernel of the matrix formed by the counterexamples in the bitvector domain [31, 15]. For inequality relations, *StrongestInvCover*( $\Gamma$ ) returns false iff any counterexample in  $\Gamma$  evaluates the relation to false — this effectively simulates the Houdini approach [21]. In case of recursive relations, *StrongestInvCover*( $\Gamma$ ) attempts to disprove the recursive relation  $l_1 \sim l_2$  by evaluating its depth- $\eta$  underapproximation  $l_1 \sim_\eta l_2$  for each counterexample in  $\Gamma$  and returns false if any one of them successfully evaluates to false.  $\eta$  is a constant parameter of the algorithm.

## 4.4 Modeling Procedure Calls

A top-level procedure  $\delta$  in  $S$  or  $C$  may make non-tail recursive calls, e.g., for traversing a tree data structure. Our correlation algorithm (section 4.2) ensures that the anchor nodes around such a callsite are correlated one-to-one across both programs. For example, let there be a recursive call in  $S$  at PC  $A_S$ , i.e.,  $A_S$  is the callsite. Then we denote the program points just before and just after this callsite as  $A_S^b$  and  $A_S^a$  respectively. Let  $\mathbf{args}_{A_S}$  represent the values of the actual arguments of this procedure call. Let  $\mathbf{ret}_{A_S}$  represent the values returned by this procedure call. Similarly, for a procedure call at PC  $A_C$  in  $C$ , let  $A_C^b$ ,  $A_C^a$ ,  $\mathbf{args}_{A_C}$  and  $\mathbf{ret}_{A_C}$  represent the before-callsite program point, after-callsite program point, arguments and return values respectively. Our algorithm ensures that the only correlations possible in a product-CFG  $\pi$  for these  $S$  and  $C$  program points are  $A_\pi^b = (A_S^b, A_C^b)$  and  $A_\pi^a = (A_S^a, A_C^a)$ .

Recall that the recursive call at  $A_S$  (or  $A_C$ ) must be a call to the top-level procedure  $\delta$ . We utilize the user-supplied *Pre* and *Post* conditions for  $\delta$  to obtain the desired invariants at nodes  $A_\pi^b$  and  $A_\pi^a$  in the product-CFG. We require a successful proof to *ensure* that  $Pre(A_S^{\mathbf{args}_S}, A_C^{\mathbf{args}_C}, m_b)$  holds at  $A_\pi^b$ . Further, the proof can *assume* that  $Post(A_S^{\mathbf{ret}_S}, A_C^{\mathbf{ret}_C}, m_a)$  holds at  $A_\pi^a$ . Here,  $m_b$  and  $m_a$  represent the memory states in  $C$  at  $A_C^b$  and  $A_C^a$  respectively. Thus, for such recursive calls to

---

the top-level function, we inductively prove the precondition (on the arguments of the procedure call) at  $A_\pi^b$  and assume the postcondition (on the return values of the procedure call) at  $A_\pi^a$ .

## 4.5 Points-to Analysis

We formulate our points-to analysis as a dataflow analysis as discussed below. We first identify the set  $R_C$  of all region labels representing mutually non-overlapping regions of the  $C$  memory state  $m$ . For each call to `malloc()` at PC  $A$ , we add  $A_1$  and  $A_{2+}$  to  $R_C$ .  $R_C = \bigcup_A \{A_1, A_{2+}\} \cup \{\text{heap}\}$ , where `heap` represents all *other* memory regions that are not captured by the region labels associated with allocation sites.

Let  $S_C$  be the set of all scalar pseudo-registers in  $C$ 's IR. We use a forward dataflow analysis to identify a may-point-to function  $\Delta : (S_C \cup R_C) \mapsto 2^{R_C}$  at each program point. For an IR instruction  $\mathbf{x} := c$ , for constant  $c$ , the transfer function updates  $\Delta(\mathbf{x}) := \emptyset$ . For instruction  $\mathbf{x} := \mathbf{y} \text{ op } \mathbf{z}$  (for some arithmetic or logical operand `op`), we update  $\Delta(\mathbf{x}) := \Delta(\mathbf{y}) \cup \Delta(\mathbf{z})$ . For a load instruction  $\mathbf{x} := *y$ , we update  $\Delta(\mathbf{x})$  to  $\bigcup_{R_C \in \Delta(y)} \Delta(R_C)$ . For a store instruction  $*\mathbf{x} := y$ , for all  $R_C \in \Delta(\mathbf{x})$ , we update  $\Delta(R_C) := \Delta(R_C) \cup \Delta(y)$ . For recursive procedure 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 return to the returning point of the call-site (copying returned value to the variable assigned at the callsite), e.g., in fig. 14, the dashed edges represent supergraph edges. For a `malloc` instruction  $\mathbf{x} := \text{malloc}_A()$  (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  $r \in S_C \cup R_C$ , if  $A_1 \in \Delta(r)$ , then update  $\Delta(r) := (\Delta(r) \setminus \{A_1\}) \cup \{A_{2+}\}$ .
  2. Update  $\Delta(\mathbf{x}) := \{A_1\}$
  3. Update  $\Delta(A_{2+}) := \Delta(A_{2+}) \cup \Delta(A_1)$ .
  4. Update  $\Delta(A_1) := \emptyset$  (empty set).
-

The meet operator is set-union. For a C program  $C$ , the boundary condition at entry is given by  $\Delta_C^{entry}(r) = R_C$  for all  $r \in S_C \cup R_C$ , where  $\Delta_P^{pc}$  represents the may-point-to function for program  $P$  at PC  $pc$ .

In case of a reconstruction program  $R$ , the domain of  $\Delta$  contains the pseudo-registers in  $C$ 's IR ( $S_C$ ) as well as any region labels ( $R_C$ ). In addition to these, the domain also contains the pseudo-registers of the reconstruction program itself, say  $R_R$ . For a reconstruction program  $R$  originating from a proof obligation at a product program PC  $(n_S, n_C)$ , the boundary condition is given by:

$$\Delta_R^{entry}(r) = \begin{cases} \Delta_C^{n_C}(r) & \text{for all } r \in S_C \cup R_C \\ \emptyset & \text{for all } r \in R_R \end{cases}$$

Hence, for a reconstruction program, we use the results of the points-to analysis on  $C$  at the PC where the proof obligation is being discharged. This is a crucial step for proving equality of  $C$  values under different  $C$  memory state as seen in section 3.9.5.

## 5 Evaluation

We have implemented S2C on top of the Counter tool [22]. 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** [18], **Yices2-45e38fc** [19], 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 value of our unrolling parameter  $k$  (used for categorization of proof obligations) is *five*.

S2C requires the user to provide a Spec program  $S$  (specification), a C implementation  $C$ , and a file that contains the precondition  $Pre$  and postcondition  $Post$ . An equivalence check requires the identification of lifting constructors to relate C values to the ADT values in Spec 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., in the 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) **String**, (T2) **List**, (T3) **Tree** and (T4) **Matrix**. For each Spec program specification, we consider multiple C implementations that differ in their (a) layout and representation 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 4:** String lifting constructors and their definitions.

Lifting Constructor	Definition
(T1) <b>Str</b> = SInvalid   SNil   SCons(i8, Str)	
$\text{Cstr}_m^{\text{u8}}(p:i32)$	<pre> if p = 0<sub>i32</sub> then SInvalid elif p[0<sub>i32</sub>]<sub>m</sub><sup>i8</sup> = 0<sub>i8</sub> then SNil else SCons(p[0<sub>i32</sub>]<sub>m</sub><sup>i8</sup>, Cstr<sub>m</sub><sup>u8</sup>(p + 1<sub>i32</sub>)) </pre>
$\text{Cstr}_m^{\text{lnode}(\text{u8})}(p:i32)$	<pre> if p = 0<sub>i32</sub> then SInvalid elif p <math>\xrightarrow{m}</math><sub>lnode</sub> val = 0<sub>i8</sub> then SNil else SCons(p <math>\xrightarrow{m}</math><sub>lnode</sub> val, Cstr<sub>m</sub><sup>lnode(u8)</sup>(p <math>\xrightarrow{m}</math><sub>lnode</sub> next)) </pre>
$\text{Cstr}_m^{\text{clnode}(\text{u8})}(p:i32, i:i2)$	<pre> if p = 0<sub>i32</sub> then SInvalid elif p <math>\xrightarrow{m}</math><sub>lnode</sub> chunk[i]<sub>m</sub><sup>i8</sup> = 0<sub>i8</sub> then SNil else SCons(p <math>\xrightarrow{m}</math><sub>lnode</sub> chunk[i]<sub>m</sub><sup>i8</sup>, Cstr<sub>m</sub><sup>clnode(u8)</sup>(i = 3<sub>i2</sub>?p <math>\xrightarrow{m}</math><sub>clnode</sub> next : p, i + 1<sub>i2</sub>)) </pre>

### 5.1.1 String

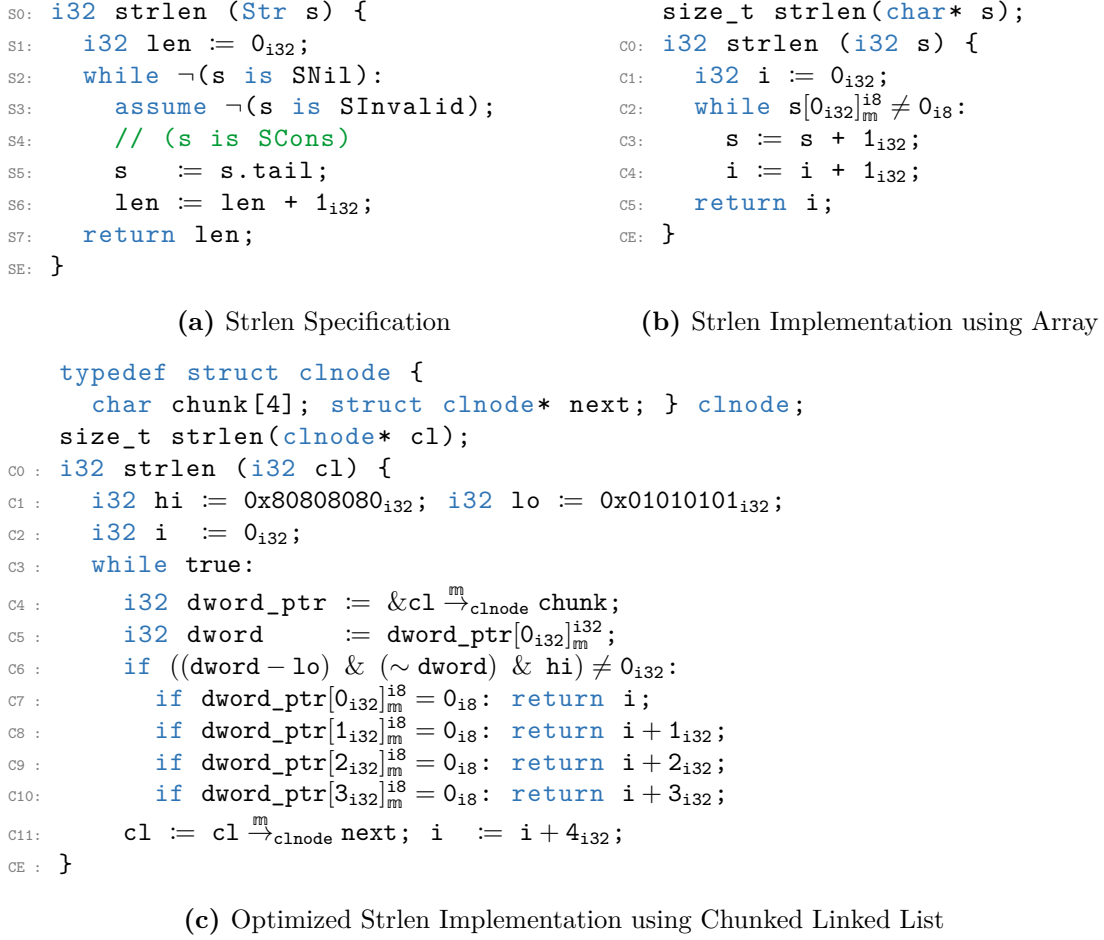
We wrote a single specification in Spec for each of the following common string library functions: **strlen**, **strchr**, **strcmp**, **strspn**, **strcspn**, and **strpbrk**. For each specification program, we took multiple C implementations of that program, drawn from popular libraries like **glibc** [3], **klibc** [4], **newlib** [7], **openbsd** [8],

`uClibc` [9], `dietlibc` [2], `musl` [5], and `netbsd` [6]. Some of these libraries implement the same function in two ways: one that is optimized for code size and another that is optimized for runtime. All these library implementations use a *null character* terminated array to represent a string, and the corresponding lifting constructor is  $\text{Cstr}_m^{\text{u8}[]}$ .  $\text{u<N>}$  represents the N-bit unsigned integer type in C. For example, `u8` represents `unsigned char` type.

Further, we implemented custom C programs for all of these functions that used 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 4.  $\text{Cstr}_m^{\text{lnode}(\text{u8})}$  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.

Figure 17 shows the `strlen` specification and two vastly different C implementations. Figure 17b is a generic implementation using a null character terminated array to represent a string similar to a C-style string. The second implementation in fig. 17c differs from fig. 17b in the following: (a) it uses a chunked linked list data layout for the input string and (b) it uses specialized bit manipulations to identify a null character in a chunk at a time. S2C is able to automatically find a bisimulation relation for both implementations against the unaltered specification. Figure 18 shows the product-CFG and invariants for each implementation.

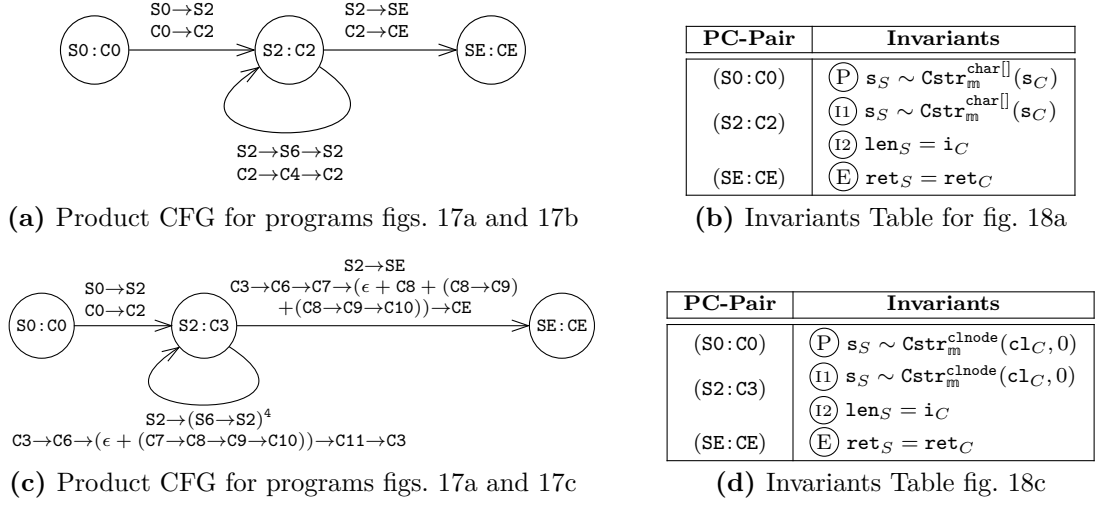
Lifting constructors are named based on the C data layout being lifted and the Spec ADT type of the lifted value. For example,  $\text{Cstr}_m^{\text{u8}[]}$  represents a `String` lifting constructor for an array layout. In general, we use the following naming convention for different C data layouts: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



**Figure 17:** Specification of Strlen along with two possible C implementations. Figure 17b is a generic implementation using a null-terminated array for `String`. Figure 17c is an optimized implementation using a chunked linked list for `String`.

**Table 5:** List lifting constructors and their definitions.

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



**Figure 18:** Product CFGs and Invariants Tables showing bisimulation between Strlen specification in fig. 17a and two C implementations in figs. 17b and 17c

### 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 ( $\text{Clist}_m^{\text{cnode}(\text{u32})}$ ). The lifting constructors are shown in table 5. 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  $p$  of size  $n$  starting at the  $i^{\text{th}}$  index. The list becomes empty when we are at the end of the array. ( $\text{Clist}_m^{\text{lnode}(\text{u32})}$ ) and ( $\text{Clist}_m^{\text{cnode}(\text{u32})}$ ), on the other hand, encodes empty lists (LNil) using *null pointers*. These layouts are in contrast to the String layouts, all of which uses a *null character* to indicate the empty string.

**Table 6:** Tree lifting constructors and their definitions.

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

### 5.1.3 Tree

We wrote a Spec program that sums all the nodes in a tree through an inorder traversal using recursion. We use two different data layouts for a tree: (1) a flat array where a complete binary tree is laid out in breadth-first search order commonly used for heaps ( $\text{Ctree}_m^{u32[]}$ ), and (2) a linked tree node with two pointers for the left and right children ( $\text{Ctree}_m^{\text{tnode}(u32)}$ ) (shown in table 6). 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*. 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 returned states.

**Table 7:** Matrix and auxiliary List lifting constructors and their definitions.

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

### 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** ( $(\text{T4})$  in table 7). The C implementations for a **Matrix** object include (a) a two-dimensional array ( $\text{Cmat}_m^{u32[]}$ ), (b) a flattened row-major array ( $\text{Cmat}_m^{u32[r]}$ ), (c) a flattened



---

column-major array ( $\mathbf{Cmat}_m^{u32[c]}$ ), (d) a linked list of 1D arrays ( $\mathbf{Cmat}_m^{1node(u32[])}$ ), (e) a 1D array of linked lists ( $\mathbf{Cmat}_m^{1node(u32)[]}$ ) and (f) a 1D array of chunked linked list ( $\mathbf{Cmat}_m^{c1node(u32)[]}$ ) data layouts. Note that both  $\mathbf{T}[r]$  and  $\mathbf{T}[c]$  represent a 1D array of type  $\mathbf{T}$ . The  $r$  and  $c$  simply emphasizes that these arrays are used to represent matrices in row-major and column-major encodings respectively. We also introduce two auxiliary lifting constructors,  $\mathbf{Clist}_m^{u32[r]}$  and  $\mathbf{Clist}_m^{u32[c]}$  for lifting each row of matrices lifted using the corresponding  $\mathbf{Cmat}_m^{u32[r]}$  and  $\mathbf{Cmat}_m^{u32[c]}$  **Matrix** lifting constructors. These constructors are listed in table 7.

---

**Table 8:** Equivalence checking times and minimum under- and over-approximation depth values at which equivalence checks succeeded.

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

## 5.2 Results

Table 8 lists the various C implementations and the time it 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).

During the verification of `strchr` and `strpbrk` implementations, we identified an interesting subtlety. Since `strchr` and `strpbrk` return null pointers to signify absence of the required character(s) in the input string, we additionally need to model the UB assumption that the zero address does not belong to the null character terminated array representing the string. We use an explicit constructor `SInvalid` to expose this well-formedness property in a Spec `String`. Furthermore, we relate `SInvalid` to the condition of C character pointer being null using the lifting constructors  $\text{Cstr}_m^T(p:\text{i32}, \dots)$  (as defined in table 5). These lifting constructors are used as part of *Pre* to equate *S* and *C* input strings. Finally in *S*, we model the absence of `SInvalid` in the input string as a UB assumption using the `assuming-do` statement introduced in section 4.1. Due to the (*S def*) assumption, this constraints the inputs to *S* as well as *C* to well-formed strings only. This is an example where (*S def*) and *Pre* can be used to model wellformedness of values in *C*.

TODO: add strlen spec atleast, show the strchr also!! maybe some matrix data layouts (only layouts)

## 6 Limitations

Our proof discharge algorithm is not without limitations. For a recursive relation relating values of a non-linear ADT such as `Tree`, a *d*-depth approximation results in  $\sim 2^d$  smaller equalities. This is a major cause of inefficiency due to generation of large queries which slows down SMT solvers and counterexample-guided algorithms for large values of *d*.

---

S2C is only interested in finding a bisimulation relation and hence equivalence of non-bisimilar programs is beyond our scope. S2C currently only supports bitvector affine and inequality relations along with recursive relations provided as part of *Pre* and *Post*. Consequently, non-linear bitvector invariants (e.g. polynomial invariants) as well as custom recursive relations are not supported. While our correlation and invariant inference algorithms based on the Counter tool [22] are designed for translation validation between (C-like) unoptimized IR and assembly, we found them to be surprisingly good for Spec to (C-like) IR as well. Rather unsurprisingly, S2C suffers from the same limitations of these algorithms. For example, S2C supports path specializations from Spec to C, it does not search for path merging correlations.

## 7 Conclusion

As introduced in ??, 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 [14, 36, 34, 35] involves a combination of careful system design and automatic verification tools like SMT solvers. Constrained Horn Clause (CHC) Solvers [17] 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 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.

## 8 Outline of the Thesis

**Chapter 1** of the thesis contains a general introduction to the research problem of verification C programs against a functional specification. We take a C program and its analogue in a safe functional language, and contrast their differences. This helps us motivate the problem and its solution. We finish with our contributions.

In **Chapter 2**, we constrain the programs being considered by formulating the problem statement. This helps us define the scope of our solution. We introduce a custom minimal functional language called ‘Spec’ and define the necessary terminology used in the rest of the thesis.

**Chapter 3** starts with background on program equivalence, bisimulation relation and product program. The rest of the chapter gradually introduces our first contribution: A Proof Discharge Algorithm and related sub-procedures with the help of two example programs. We also introduce a program representation of values, called ‘reconstruction program’.

Next, we formalize previously discussed topics in **Chapter 4**. We begin with the specification of our custom language ‘Spec’. We give a detailed description of (a) a counterexample-guided search algorithm for finding a bisimulation relation and (b) a counterexample-guided invariant inference procedure. There two procedures with our proof discharge algorithm allow automatic end-to-end equivalence checking. We formalize handling of procedure calls in ‘Spec’ as well as C, and finish with a dataflow analysis formulation of a pointer analysis used by our equivalence checker.

**Chapter 5** introduces a program graph representation of values, called ‘value graphs’, similar to ‘reconstruction procedure’ as introduced in **Chapter 3**. We motivate it by listing its advantages and give an algorithm to convert expressions to this representation. This helps us simplify our proof discharge algorithm.

In **Chapter 6**, we introduce our automatic equivalence checker tool named S2C, based on our proof discharge algorithm and counterexample-guided search procedures. S2C is evaluated on a large variety of C programs involving lists, strings, trees and matrices. This includes C programs taken from C library implementations as well as manually written programs. We show that our equivalence checker is able to prove equivalence of a single specification with multiple of C implementations, each varying in its data layouts and algorithmic strategies.

---

---

Finally, **Chapter 7** discusses the limitations of our algorithm and compares it with some related work. We note our major ideas and finish with some potential future improvements to our algorithm.

---

## **Publications Based on Research Work**

---

## References

- [1] (2023). Cvc4 theorem prover webpage. <https://cvc4.github.io/>.
  - [2] (2023). diet libc webpage. <https://www.fefe.de/dietlibc/>.
  - [3] (2023). Gnu libc sources. <https://sourceware.org/git/glibc.git>.
  - [4] (2023). klibc libc sources. <https://git.kernel.org/pub/scm/libs/klibc/klibc.git>.
  - [5] (2023). musl libc sources. <https://git.musl-libc.org/cgit/musl>.
  - [6] (2023). Netbsd libc sources. <http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/>.
  - [7] (2023). Newlib libc sources. <https://www.sourceware.org/git/?p=newlib-cygwin.git>.
  - [8] (2023). Openbsd libc sources. <https://github.com/openbsd/src/tree/master/lib/libc>.
  - [9] (2023). uclibc libc sources. <https://git.uclibc.org/uClibc/>.
  - [10] **Andersen, L. O.** (1994). Program analysis and specialization for the C programming language. Technical report.
  - [11] **Barrett, C., Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck,** Tvoc: A translation validator for optimizing compilers. In **K. Etessami and S. K. Rajamani** (eds.), *Computer Aided Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31686-2.
  - [12] **Benton, N.**, 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. Association for Computing Machinery, New York, NY, USA, 2004. ISBN 158113729X. URL <https://doi.org/10.1145/964001.964003>.
  - [13] **Burstall, R. M., D. B. MacQueen, and D. T. Sannella,** Hope: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80. Association for Computing Machinery, New York, NY, USA, 1980. ISBN 9781450373968. URL <https://doi.org/10.1145/800087.802799>.
-



- 
- [14] **Chen, H., D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich**, Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450338349. URL <https://doi.org/10.1145/2815400.2815402>.
- [15] **Churchill, B., O. Padon, R. Sharma, and A. Aiken**, Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*. ACM, New York, NY, USA, 2019. ISBN 978-1-4503-6712-7. URL <http://doi.acm.org/10.1145/3314221.3314596>.
- [16] Coq:Equiv (2023). Program Equivalence in Coq. <https://softwarefoundations.cis.upenn.edu/plf-current/Equiv.html>.
- [17] **De Angelis, E., F. Fioravanti, A. Pettorossi, and M. Proietti**, Relational verification through horn clause transformation. In **X. Rival** (ed.), *Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-53413-7.
- [18] **De Moura, L. and N. 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/E-TAPS'08*. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 3-540-78799-2, 978-3-540-78799-0. URL <http://dl.acm.org/citation.cfm?id=1792734.1792766>.
- [19] **Dutertre, B.**, Yices 2.2. In **A. Biere and R. Bloem** (eds.), *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.
- [20] **Felsing, D., S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich**, Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-3013-8. URL <http://doi.acm.org/10.1145/2642937.2642987>.
- [21] **Flanagan, C. and K. R. 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*. Springer-Verlag, Berlin, Heidelberg, 2001. ISBN 3540417915.
- [22] **Gupta, S., A. Rose, and S. Bansal** (2020). Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4(OOPSLA). URL <https://doi.org/10.1145/3428289>.
-

- 
- [23] **Hoare, C. A. R.** (1969). An axiomatic basis for computer programming. *Commun. ACM*, **12**(10), 576–580. ISSN 0001-0782. URL <https://doi.org/10.1145/363235.363259>.
- [24] **Kanade, A., A. Sanyal, and U. P. Khedker** (2009). Validation of gcc optimizers through trace generation. *Softw. Pract. Exper.*, **39**(6), 611–639. ISSN 0038-0644. URL <http://dx.doi.org/10.1002/spe.v39:6>.
- [25] **Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood**, Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605587523. URL <https://doi.org/10.1145/1629575.1629596>.
- [26] **Kundu, S., Z. Tatlock, and S. Lerner**, Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-392-1. URL <http://doi.acm.org/10.1145/1542476.1542513>.
- [27] **Leino, K. R. M.**, Dafny: An automatic program verifier for functional correctness. In **E. M. Clarke and A. Voronkov** (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-17511-4.
- [28] **Leung, A., D. Bounov, and S. Lerner**, C-to-verilog translation validation. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*. 2015.
- [29] LLVM LangRef (2023). LLVM language reference manual. <http://llvm.org/docs/LangRef.html>.
- [30] **Lopes, N. P. and J. Monteiro** (2016). Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *Int. J. Softw. Tools Technol. Transf.*, **18**(4), 359–374. ISSN 1433-2779. URL <http://dx.doi.org/10.1007/s10009-015-0366-1>.
- [31] **Müller-Olm, M. and H. Seidl**, Analysis of modular arithmetic. In **M. Sagiv** (ed.), *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31987-0.
- [32] **Namjoshi, K. and L. Zuck**, Witnessing program transformations. In **F. Logozzo and M. Fähndrich** (eds.), *Static Analysis*, volume 7935
-

- of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38855-2, 304–323. URL [http://dx.doi.org/10.1007/978-3-642-38856-9\\_17](http://dx.doi.org/10.1007/978-3-642-38856-9_17).
- [33] **Necula, G. C.**, Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00. ACM, New York, NY, USA, 2000. ISBN 1-58113-199-2. URL <http://doi.acm.org/10.1145/349299.349314>.
- [34] **Nelson, L., J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang**, Scaling symbolic evaluation for automated verification of systems code with serval. In **T. Brecht** and **C. Williamson** (eds.), *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019. URL <https://doi.org/10.1145/3341301.3359641>.
- [35] **Nelson, L., J. V. Geffen, E. Torlak, and X. 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*. USENIX Association, 2020. URL <https://www.usenix.org/conference/osdi20/presentation/nelson>.
- [36] **Nelson, L., H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang**, Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-5085-3. URL <http://doi.acm.org/10.1145/3132747.3132748>.
- [37] **Poetzsch-Heffter, A.** and **M. Gawkowski** (2005). Towards proof generating compilers. *Electron. Notes Theor. Comput. Sci.*, **132**(1), 37–51. ISSN 1571-0661. URL <http://dx.doi.org/10.1016/j.entcs.2005.03.023>.
- [38] **Sewell, T. A. L., M. O. Myreen, and G. Klein**, Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450320146. URL <https://doi.org/10.1145/2491956.2462183>.
- [39] **Sharma, R., E. Schkufza, B. Churchill, and A. Aiken**, Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13*. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2374-1. URL <http://doi.acm.org/10.1145/2509136.2509509>.
-

- 
- [40] **Stepp, M., R. Tate, and S. Lerner**, Equality-based translation validator for llvm. In *Proceedings of the 23rd International Conference on Computer Aided Verification, CAV'11*. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-22109-5. URL <http://dl.acm.org/citation.cfm?id=2032305>. 2032364.
  - [41] **Strichman, O. and B. Godlin**, Regression verification - a practical way to verify programs. In **B. Meyer and J. Woodcock** (eds.), *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69147-1, 496–501. URL [http://dx.doi.org/10.1007/978-3-540-69149-5\\_54](http://dx.doi.org/10.1007/978-3-540-69149-5_54).
  - [42] **Tate, R., M. Stepp, Z. Tatlock, and S. 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*. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-379-2. URL <http://www.cs.cornell.edu/~ross/publications/eqsat/>.
  - [43] **Tristan, J.-B., P. Govereau, and G. Morrisett**, Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0663-8. URL <http://doi.acm.org/10.1145/1993498.1993533>.
  - [44] **Zaks, A. and A. Pnueli**, Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods, FM '08*. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-68235-6. URL [http://dx.doi.org/10.1007/978-3-540-68237-0\\_5](http://dx.doi.org/10.1007/978-3-540-68237-0_5).
  - [45] **Zuck, L., A. Pnueli, Y. Fang, and B. Goldberg** (2003). Voc: A methodology for the translation validation of optimizing compilers. **9**(3), 223–247.
  - [46] **Zuck, L., A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu** (2005). Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, **27**(3), 335–360. ISSN 0925-9856. URL <http://dx.doi.org/10.1007/s10703-005-3402-z>.
-