Synopsis of 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

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

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 [20]. Frameworks for program equivalence proofs have been developed in interactive theorem provers like Coq [13] where correlations and invariants are manually identified during proof codification. On the other hand, programming languages like Dafny [21] 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.

We present S2C, an algorithm to automatically (push-button) search for a proof of equivalence between a functional specification 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 [25, 17], where S2C can be used to automate verification across software updates that change memory layouts for data structures. We start by formulating the problem statement and define equivalence in this context. Next, we shortly introduce bisimulation in the context of program equivalence and list our primary contributions.

1.1 Problem Setting and Bisimulation

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) [11] 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.

```
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=
A7:    mk_list_impl(n, 0i32, LNil).
```

(a) Spec Program

```
S0: List mk_list (i32 n) {
S1:   List l := LNil;
S2:   i32  i := 0i32;
S3:   while ¬(i ≥u n):
S4:     l := LCons(i, l);
S5:     i := i + 1i32;
S6:   return l;
SE: }
```

(b) (Abstracted) Spec IR

```
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: }
```

(c) C Program with `malloc()`

```
C0: i32 mk_list (i32 n) {
C1:   i32 l := 0i32;
C2:   i32 i := 0i32;
C3:   while i <u n:
C4:     i32 p := malloc_C4(sizeof lnode);
C5:     m := m[&(p →^m_lnode val) ← i]i32;
C6:     m := m[&(p →^m_lnode next) ← l]i32;
C7:     l := p;
C8:     i := i + 1i32;
C9:   return l;
CE: }
```

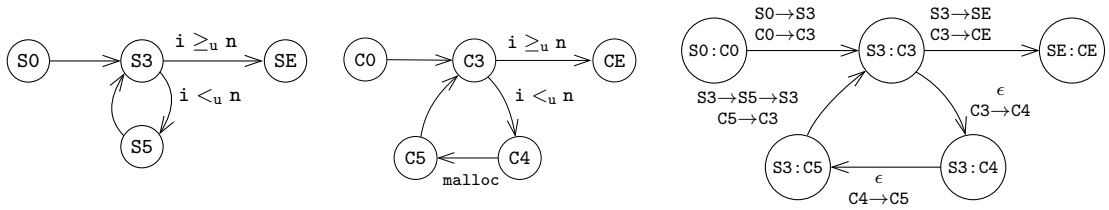
(d) (Abstracted) C IR

Figure 1: Spec and C programs for constructing a linked list along with their corresponding abstracted intermediate representations.

Figures 1a and 1c show the construction of lists in Spec and C respectively. 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 Three-Address-Code (3AC) style intermediate representation (IR) as shown in figs. 1b and 1d. For a Spec program, (a) all tail-recursive calls are converted to loops in IR while non-tail calls are preserved and (b) `match` statements are lowered to equivalent `if`-`then`-`else` conditionals. For a C program, (a) the sizes and memory layouts of both scalar (e.g., `int`) and compound (e.g., `struct`) types are concretized in IR and (b) all allocation functions (e.g., `malloc`) are annotated with their call-site i.e. IR PC (e.g., $\texttt{malloc}_{\texttt{C4}}$ in fig. 1d).

1.1.1 Equivalence Definition

S2C computes equivalence between the IRs of the Spec and C sources respectively. Henceforth, we will omit the sources and continue to refer to the IRs as Spec and C directly. Given (1) a Spec program specification S, (2) a C implementation C, (3) a precondition Pre that relates the inputs $\texttt{Input}_S$ and $\texttt{Input}_C$ to S and C respectively, and (4) a postcondition $Post$ that relates the final outputs $\texttt{Output}_S$ and $\texttt{Output}_C$ of S and C respectively: S and C are *equivalent* if for all possible inputs $\texttt{Input}_S$ and $\texttt{Input}_C$ such that $Pre(\texttt{Input}_S, \texttt{Input}_C)$ holds, S's execution is



(a) CFG of Spec Program **(b)** CFG of C Program **(c)** CFG of Product Program

Figure 2: CFG representation for Spec and C IRs shown in figs. 1b and 1d. Figure 2c shows a product-CFG between the CFGs in figs. 2a and 2b.

Table 1: Node Invariants for Product-CFG in fig. 2c

PC-Pair	Invariants
(S0:C0)	(P) $\mathtt{n}_S = \mathtt{n}_C$
(S3:C3)	(I1) $\mathtt{n}_S = \mathtt{n}_C$ (I2) $\mathtt{i}_S = \mathtt{i}_C$ (I3) $\mathtt{i}_S \leq_u \mathtt{n}_S$ (I4) $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)$
(S3:C4) (S3:C5)	(I5) $\mathtt{n}_S = \mathtt{n}_C$ (I6) $\mathtt{i}_S = \mathtt{i}_C$ (I7) $\mathtt{i}_S <_u \mathtt{n}_S$ (I8) $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)$
(SE:CE)	(E) $\mathtt{ret}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{ret}_C)$

well-defined on $\mathtt{Input}_S$ i.e. $(\mathcal{S}\ \mathtt{def})$, *and* C's memory allocation requests during its execution on $\mathtt{Input}_C$ are successful i.e. $(\mathcal{C}\ \mathtt{fits})$, then both programs S and C produce outputs that satisfy *Post*.

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

Figures 2a and 2b show the Control-Flow Graph (CFG) representation of the Spec and C programs in figs. 1b and 1d respectively. Each node represent a PC location of its corresponding program, and each edge represent transitions between PCs through instruction execution. For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 2b, the edge C5→C3 represents the path C5→C6→C7→C8→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).

1.1.2 Bisimulation Relation

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 the lockstep execution ensures identical observable behaviour. A bisimulation relation between two programs can be represented using a *product program* [26] and the CFG representation of a product program is called a *product-CFG*. Figure 2c shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 2a and 2b.

Table 1 shows the precondition (labeled Ⓟ), inductive invariants (labeled Ⓘ) and postcondition (labeled Ⓔ) for the product-CFG in fig. 2c. The precondition and postcondition are provided manually by the user while the intermediate inductive invariants are inferred automatically. The invariant labeled Ⓘ4 is an example of a *recursive relation* and represents equality between the Spec `List` variable $\texttt{l}_S$ and the `List` represented by chasing `lnode` pointers starting at $\texttt{l}_C$. Semantically $l_1 \sim l_2$ and $l_1 = l_2$ are equivalent, '$\sim$' simply emphasizes the fact that l_1 and l_2 are values of (possibly recursive) ADT types. $\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(p)$ is called a *lifting constructor* that *lifts* the C pointer value p (pointing to an object of type `struct lnode`) and the C memory state `m` to a (possibly infinite in case of a circular list) Spec `List` value, and is defined as follows:

$$\begin{aligned} U_C : \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(p\!:\!\texttt{i32}) = \underline{\texttt{if}}\ p = 0\ \underline{\texttt{then}}\ \texttt{LNil} \\ \underline{\texttt{else}}\ \texttt{LCons}(p \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{val}, \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(p \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{next})) \end{aligned} \tag{1}$$

The iterative construction of the product-CFG along with inference of inductive invariants at its nodes are based on prior work [18] and discussed briefly in section 1.2. Given a product-CFG and inferred invariants, the equivalence checker attempts to prove each inductive invariant and postcondition under the appropriate preconditions for each edge in the product-CFG. These proof obligations are expressed as relational Hoare triples [10, 19] and lowered to a first order logic predicate using weakest precondition predicate transformer. For example, $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$ represents the Hoare triple where ϕ_s and ϕ_d represents the pre- and postconditions and (ρ_S,ρ_C) represents a product-CFG edge correlating the paths ρ_S and ρ_C in S and C respectively. The above Hoare triple lowers to the following predicate:

$$(\phi_s \wedge pathcond_{\rho_S} \wedge pathcond_{\rho_C} \wedge ubfree_{\rho_S}) \Rightarrow \texttt{WP}_{\rho_S,\rho_C}(\phi_d) \tag{2}$$

We will use '`LHS`' and '`RHS`' to refer to the antecedent and consequent of the implication operator '$\Rightarrow$' in eq. (2). These proof obligations often contains recursive relations encoding equality between arbitrarily-deep recursively data structure values of S and C respectively. The handling of these proof obligations is a major challenge and forms our primary contribution as discussed next in section 1.2.

1.2 Our Contributions

As previously discussed in section 1.1.2, showing equivalence through a bisimulation proof requires three major steps: ① An algorithm for construction of a product-CFG by correlating program executions across the Spec and C programs respectively, ② An algorithm for identification of inductive invariants at correlated PCs and ③ A proof discharge algorithm for discharging proof obligations containing recursive relations. Our major contributions are as follows:

- Proof Discharge Algorithm: Discharging proof obligations (③) involving recursive relations is rather challenging 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 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 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[18] 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.

Section 2 walks through our proof discharge algorithm by demonstrating each of its components using examples. We evaluate our equivalence checking tool S2C in section 3. Finally, section 4 gives an overview of its limitations and section 5 concludes our discussion.

2 Proof Discharge Algorithm through Examples

This section demonstrates our proof discharge algorithm through example programs that construct and traverse a linked list respectively. Our equivalence checker has the property that it is safe to return false (i.e. disproven) for all proof obligations. Keeping this in mind, our proof discharge algorithm is designed to be *sound* i.e. (a) whenever it evaluates a proof obligation to true (i.e. proven), the actual proof obligation must also be proven and (b) whenever it returns false with a set of counterexamples, the counterexamples must falsify the actual proof obligation. However, it is possible for our proof discharge algorithm to return false (without a counterexample) even if the actual proof obligation is true. Our equivalence checking algorithm also ensures that, for an invariant $\phi_s = (\phi_{\mathtt{s}}^{\mathtt{1}} \wedge \phi_{\mathtt{s}}^{\mathtt{2}} \wedge ... \wedge \phi_{\mathtt{s}}^{\mathtt{k}})$, at any node s of a product-CFG, if a recursive relation appears in ϕ_s, it must be one of ϕ_s^1, ϕ_s^2, ..., or ϕ_s^k. We call this the *conjunctive recursive relation* property of an invariant ϕ_s.

A proof obligation $\{\phi_s\}(e)\{\phi_d\}$, where $e = (\rho_S, \rho_C)$, gets lowered using $\texttt{WP}_e(\phi_d)$ (as shown in eq. (2)) to a first-order logic formula of the following form:

$$(\eta_1^l \wedge \eta_2^l \wedge ... \wedge \eta_m^l) \Rightarrow (\eta_1^r \wedge \eta_2^r \wedge ... \wedge \eta_n^r) \tag{3}$$

Thus, due to the conjunctive recursive relation property of ϕ_s and ϕ_d, any recursive relation in eq. (3) must appear as one of η_i^l or η_j^r. To simplify proof obligation discharge, we break a first-order logic proof obligation P of the form in eq. (3) into multiple smaller proof obligations of the form $P_j : (\texttt{LHS} \Rightarrow \eta_{\texttt{j}}^{\texttt{r}})$, 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*.

2.1 Categorization of Proof Obligations based on Decomposition of Recursive Relations

Before diving into the proof discharge algorithm, we start with a key procedure in our proof discharge algorithm called 'decomposition of recursive relations'.

$$U_S : l = \underline{\texttt{if}}\ l \text{ is } \texttt{LNil}\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(l.\texttt{val}, l.\texttt{next}) \tag{4}$$

Firstly, an expression e whose top-level operator is a lifting constructor, (e.g., $\texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(\texttt{l}_C)$), is called a *lifted expression*. A recursive relation of the form $l_1 \sim l_2$ can be *decomposed* into an equivalent set of *decomposition clauses*, each of the form $(P \rightarrow A = B)$ or $(P \rightarrow A \sim B)$. The primary algorithm behind decomposition is an iterative unification and rewriting procedure under a set of rewriting rules E. E allows rewriting of lifted expressions by their corresponding definitions (e.g., eq. (1)), and expansion of ADT variables using $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ as shown in eq. (4). Unification proceeds by (a) recursively unifying the structures created by the top-level ADT constructors (e.g., LCons) as well as the $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ operator and (b) rewriting the top-level expressions using rules in E, as necessary. For example, $\underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(0, l_1) \sim \underline{\texttt{if}}\ c_2\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(i, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$ decomposes into $\bigwedge\{c_1 = c_2, (\neg c_1 \wedge \neg c_2) \rightarrow 0 = i, (\neg c_1 \wedge \neg c_2) \rightarrow l_1 \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2)\}$. Similarly, the decomposition of $l_1 \sim \texttt{LCons}(42, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$ is given by $\bigwedge\{l_1 \text{ is } \texttt{LCons}, (l_1 \text{ is } \texttt{LCons}) \rightarrow (l_1.\texttt{val} = 42), (l_1 \text{ is } \texttt{LCons}) \rightarrow (l_1.\texttt{next} \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))\}$. The *decomposition* of an expression e is found by decomposing each recursive relation in e.

We *unroll a recursive relation* $l_1 \sim l_2$ *with respect to* E by rewriting the top-level expressions l_1 and l_2 through E (if possible) and decomposing it. We *unroll an expression* e by unrolling each recursive relation in e with respect to E. More generally, the k-unrolling of e is found by unrolling the $(k-1)$-unrolling of e recursively. For a first order logic proof obligation $P : \texttt{LHS} \Rightarrow \texttt{RHS}$, we identify its k-unrolling (for a fixed unrolling parameter k). After unrolling, we eliminate those decomposition

```
S0: i32 sum_list (List l) {
S1:   i32 sum := 0_i32;
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

```
    unsigned sum_list (lnode* l);
C0: i32 sum_list (i32 l) {
C1:   i32 sum := 0_i32;
C2:   while l ≠ 0_i32:
C3:     sum := sum + l →^m_lnode val;
C4:     l   := l →^m_lnode next;
C5:   return sum;
CE: }
```

(b) (Abstracted) C IR

Figure 3: Spec and C Programs traversing a Linked List.

clauses $(P_i \rightarrow X_i)$ whose P_i evaluates to false under the LHS ignoring all recursive relations. For example, the one-unrolling of $P : \texttt{LHS} \Rightarrow l \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(0)$, after elimination, yields $P' : \texttt{LHS} \Rightarrow l$ is LNil. We categorize a proof obligation $P : \texttt{LHS} \Rightarrow \texttt{RHS}$ based on this k-unrolled form of decomposition of P as follows:

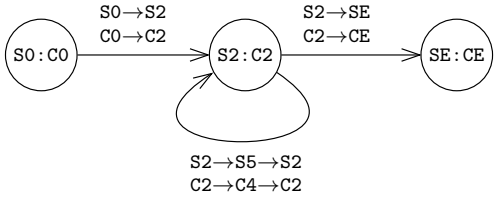
- Type I: k-unrolling of P does not contain recursive relations
- Type II: k-unrolling of P contains recursive relations only in the LHS
- Type III: k-unrolling of P contains recursive relations in the RHS

Next, we briefly describe the key ideas for each of the three types of proof obligations in sections 2.2 to 2.4.

2.2 Handling Type I Proof Obligations

In fig. 2c, consider a proof obligation generated across the product-CFG edge $(\texttt{S0:C0}) \rightarrow (\texttt{S3:C3})$ while checking if the (I4) invariant in table 1 holds at (S3:C3): $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0}\rightarrow\texttt{S3}, \texttt{C0}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)\}$. The precondition $\phi_{\texttt{S0:C0}} \equiv (\texttt{n}_S = \texttt{n}_C)$ does not contain a recursive relation. When lowered to first-order logic through $\texttt{WP}_{\texttt{S0}\rightarrow\texttt{S3},\texttt{C0}\rightarrow\texttt{C3}}$, this translates to $\texttt{n}_S = \texttt{n}_C \Rightarrow \texttt{LNil} \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(0)$. Here, LNil is obtained for $\texttt{l}_S$ and 0 (null) is obtained for $\texttt{l}_C$. The one-unrolled form of this proof obligation yields $\texttt{n}_S = \texttt{n}_C \Rightarrow \mathit{true}$ which trivially resolves to true.

Consider the following example of a proof obligation: $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0}\rightarrow\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}, \texttt{C0}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)\}$. Notice, we have changed the path in S to S0→S3→S5→S3 here. In this case, the corresponding first-order logic formula evaluates to: $\texttt{n}_S = \texttt{n}_C \wedge 0 <_u \texttt{n}_S \Rightarrow \texttt{LCons}(0, \texttt{LNil}) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(0)$. One-unrolling of this proof obligation decomposes RHS into false due to failed unification of LCons and LNil. The proof obligation is further discharged using an SMT solver which provides a counterexample (model) that evaluates the formula to false. For example, the counterexample $\{\texttt{n}_S \mapsto 42, \texttt{n}_C \mapsto 42\}$ evaluates this formula to false. These counterexamples assist in faster convergence of our invariant inference and correlation search procedures as part of the S2C tool.



(a) Product-CFG

PC-Pair	Invariants
(S0:C0)	(P) $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$
(S2:C2)	(I1) $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$ (I2) $\mathtt{sum}_S = \mathtt{sum}_C$
(SE:CE)	(E) $\mathtt{ret}_S = \mathtt{ret}_C$

(b) Node Invariants of the Product-CFG

Figure 4: Product-CFG between the IRs in figs. 3a and 3b. The inductive invariants of the Product-CFG are given in fig. 4b.

2.3 Handling Type II Proof Obligations

Consider the pair of programs in figs. 3a and 3b 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. 4a and 4b.

Consider the proof obligation originating due to (I2) invariant across edge $(\mathtt{S2:C2}) \rightarrow (\mathtt{S2:C2})$ in fig. 4a: $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2}\rightarrow\mathtt{S5}\rightarrow\mathtt{S2}, \mathtt{C2}\rightarrow\mathtt{C4}\rightarrow\mathtt{C2})\{\mathtt{sum}_S = \mathtt{sum}_C\}$, where the node invariant $\phi_{\mathtt{S2:C2}}$ contains the recursive relation $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$. The corresponding (simplified) first-order logic formula for this proof obligation is: $(\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C) \wedge \mathtt{sum}_S = \mathtt{sum}_C \wedge \neg(\mathtt{l}_S \text{ is } \mathtt{LNil}) \wedge \mathtt{l}_C \neq 0) \Rightarrow (\mathtt{sum}_S + \mathtt{l}_S.\mathtt{val}) = (\mathtt{sum}_C + \mathtt{l}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val})$. We fail to remove the recursive relation on the LHS even after k-unrolling for any finite unrolling parameter k because both sides of $\sim$ represent list values of arbitrary length. In such a scenario, we do not know of an efficient SMT encoding for the recursive relation $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$. Ignoring this recursive relation will incorrectly (although soundly) evaluate the proof obligation to false; however, for a successful equivalence proof, we need the proof discharge algorithm to evaluate it to true. Let's call this requirement (R1).

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: $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2}\rightarrow\mathtt{S5}\rightarrow\mathtt{S2}\rightarrow\mathtt{S5}\rightarrow\mathtt{S2}, \mathtt{C2}\rightarrow\mathtt{C4}\rightarrow\mathtt{C2})\{\mathtt{sum}_S = \mathtt{sum}_C\}$. Similar to the last proof obligation, its equivalent first-order logic formula contains a recursive relation in the LHS. Clearly, this proof obligation should evaluate to false. Whenever a proof obligation evaluates to false, we expect an ideal proof discharge algorithm to generate a counterexample that falsifies the proof obligation. Let's call this requirement (R2). Recall that these counterexamples help in faster convergence of our invariant inference and correlation algorithms.

To tackle requirements (R1) and (R2), our proof discharge algorithm converts the original proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$ into two approximated proof obligations $(P_{pre-o} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\})$ and $(P_{pre-u} : \{\phi_s^{u_{d_2}}\}(e)\{\phi_d\})$. Here $\phi_s^{o_{d_1}}$ and $\phi_s^{u_{d_2}}$ rep-

resent the over- and under-approximated versions of precondition ϕ_s respectively, and d_1 and d_2 represent *depth parameters* that indicate the degree of over- and under-approximation. We over (under) approximate a predicate by over (under) approximating each of its constituent recursive relation. The d-depth over- and under-approximations of a recursive relation $l_1 \sim l_2$ are written as $l_1 \sim_d l_2$ and $l_1 \approx_d l_2$ respectively. To define these two operators, we start with the notion of *depth of an ADT value*. The *depth* of an ADT value is simply the depth in its expression tree representation. We also assign a depth to each node in the said representation. The depth of the root node is defined to be zero. For example, the `List` value `LCons`(2, `LCons`(4, `LNil`)) has a depth of 2.

$l_1 \sim_d l_2$ asserts equality of the corresponding structures and scalar values (i.e. boolean and bitvector values) up to a depth of d. $l_1 \approx_d l_2$ asserts equality of the corresponding structures and scalar values up to a depth of d and bounds the depths of l_1 and l_2 to a maximum of d. $l_1 \approx_d l_2$ is equivalent to $l_1 \sim_d l_2 \wedge \Gamma_d(l_1) \wedge \Gamma_d(l_2)$, where $\Gamma_d(l)$ asserts that l has a maximum depth of d. Unlike recursive relations, these operators only equate scalar expressions and can be encoded in SMT logic. For example, the condition $l \sim_1 \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$ reduces to the SMT-encodable predicate: $(l \text{ is } \texttt{LNil} = (p = 0)) \wedge (\neg(l \text{ is } \texttt{LNil}) \rightarrow (l.\texttt{val} = p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{val}))$. Similarly, $\Gamma_2(l)$ is equivalent to $(l \text{ is } \texttt{LNil}) \vee (\neg(l \text{ is } \texttt{LNil}) \wedge (l.\texttt{tail} \text{ is } \texttt{LNil}))$ and $\Gamma_2(\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p))$ is equivalent to $(p = 0) \vee ((p \neq 0) \wedge (p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{next} = 0))$.

Thus, for a *Type II* proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$, we first submit the over-approximated proof obligation P_{pre-o} and return true if it evaluates to true. Otherwise, we submit the under-approximated proof obligation P_{pre-u}. If it returns false, then we return false with the counterexample. If both approximations fail, we soundly return false with no counterexamples.

Revisiting our proof obligations, $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}, \texttt{C2}\rightarrow\texttt{C4}\rightarrow\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$ is provable using a depth-1 overapproximation of the precondition $\phi_{\texttt{S2:C2}}$. Similarly, the proof obligation $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}, \texttt{C2}\rightarrow\texttt{C4}\rightarrow\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$ evaluates to false (with a counterexample) using a depth-2 underapproximation of the precondition $\phi_{\texttt{S2:C2}}$. The following is a possible counterexample for a depth-2 underapproximation.

$$
\begin{array}{ll}
\{ & \texttt{sum}_S \mapsto 3, \\
& \texttt{sum}_C \mapsto 3, \\
& \texttt{l}_S \mapsto \texttt{LCons(42,LCons(43,LNil))}, \\
& \texttt{l}_C \mapsto \texttt{0x123}, \\
& \mathbb{m} \mapsto \left\{ \begin{array}{l} \texttt{0x123} \mapsto_{\texttt{lnode}} (.\texttt{val} \mapsto 42, .\texttt{next} \mapsto \texttt{0x456}), \\ \texttt{0x456} \mapsto_{\texttt{lnode}} (.\texttt{val} \mapsto 43, .\texttt{next} \mapsto 0), \\ () \mapsto 77 \end{array} \right\} \\
\}
\end{array}
$$

2.4 Handling Type III Proof Obligations

In fig. 2c, consider a proof obligation generated across the product-CFG edge $(\texttt{S3:C5}) \rightarrow (\texttt{S3:C3})$ while checking if the (I4) invariant, $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$, holds at $(\texttt{S3:C3})$: $\{\phi_{\texttt{S3:C5}}\}(\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}, \texttt{C5}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)\}$. Here, a recursive relation is present both in the precondition $\phi_{\texttt{S3:C5}}$ ((I8)) and in the postcondition ((I4)) and we are unable to remove them after k-unrolling. When lowered to first-order logic through $\texttt{WP}_{\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3},\texttt{C5}\rightarrow\texttt{C3}}$, this translates to (showing only relevant relations):

$$\begin{aligned}(\texttt{i}_S = \texttt{i}_C \wedge \texttt{p}_C = \texttt{malloc}() \wedge \texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)) \\ \Rightarrow (\texttt{LCons}(\texttt{i}_S, \texttt{l}_S) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}'}(\texttt{p}_C))\end{aligned} \tag{5}$$

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

$$\texttt{m}' \equiv \texttt{m}[\&(\texttt{p}_C \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{val}) \leftarrow \texttt{i}_C]_{\texttt{i32}}[\&(\texttt{p}_C \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{next}) \leftarrow \texttt{l}_C]_{\texttt{i32}} \tag{6}$$

Here, $\texttt{m}[a \leftarrow v]_{\texttt{T}}$ represents an array that is equal to $\texttt{m}$ everywhere except at addresses starting at a which contains the value v of type T. We refer to these memory writes that distinguish $\texttt{m}$ and $\texttt{m}'$, as the *distinguishing writes*.

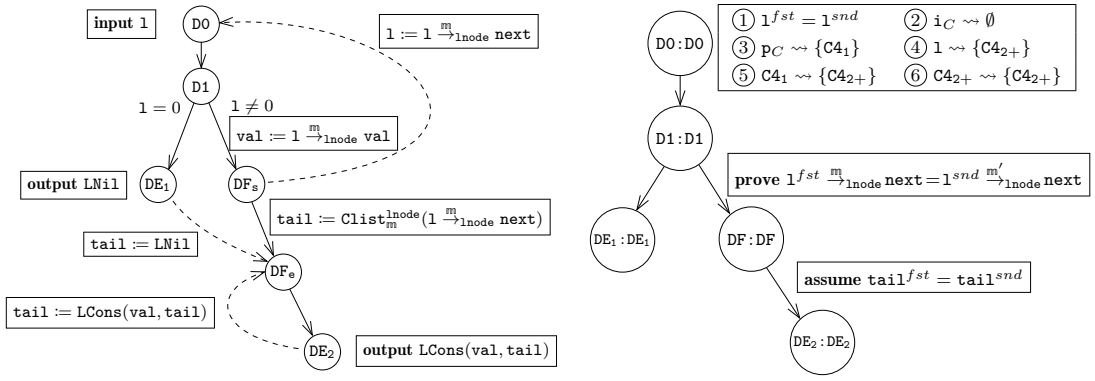
2.4.1 LHS-to-RHS Substitution and RHS Decomposition

We start by utilizing the $\sim$ relationships in the LHS (antecedent) of '$\Rightarrow$' to rewrite eq. (5) so that the ADT variables (e.g., $\texttt{l}_S$) in its RHS (consequent) are substituted with the lifted C values (e.g., $\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$). Thus, we rewrite eq. (5) to:

$$\begin{aligned}(\texttt{i}_S = \texttt{i}_C \wedge \texttt{p}_C = \texttt{malloc}() \wedge \texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)) \\ \Rightarrow (\texttt{LCons}(\texttt{i}_S, \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}'}(\texttt{p}_C))\end{aligned} \tag{7}$$

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)): (a) $\neg(\texttt{p}_C = 0)$, (b) $\neg(\texttt{p}_C = 0) \rightarrow (\texttt{i}_S = \texttt{p}_C \xrightarrow{\texttt{m}'}_{\texttt{lnode}} \texttt{val})$, and (c) $\neg(\texttt{p}_C = 0) \rightarrow (\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}'}(\texttt{p}_C \xrightarrow{\texttt{m}'}_{\texttt{lnode}} \texttt{next}))$ The first two proof obligations fall in *Type II* and are discharged through over- and under-approximation schemes (as discussed in section 2.3). Note that the first proof obligation is provable due to the ($\mathcal{C}$ fits) assumption which implies that pointer returned by malloc must be non-null. For ease of exposition, we simplify the postcondition of the third proof obligation using the ($\mathcal{C}$ fits) assumption and eq. (6) to:

$$\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}'}(\texttt{l}_C) \tag{8}$$



(a) Deconstruction Program **(b)** Decons-PCFG

Figure 5: Deconstruction program for $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(l)$ and decons-PCFG between deconstruction programs of $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C)$ and $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\texttt{l}_C)$ respectively. In fig. 5a, the square boxes show the transfer functions based on eq. (1). The dashed edges represent a function call. In fig. 5b, the square box to the right of node (D0:D0) contains the inferred invariants for this decons-PCFG.

Hence, we are interested in proving equality between two `List` values in C under different memory states $\mathtt{m}$ and $\mathtt{m}'$. Next, we show how the above can be posed as a bisimilarity check between two programs.

2.4.2 Equality of Values to Equivalence of Programs

Consider a program that recursively calls the definition (body) of $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}$ to deconstruct $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C)$. For example, $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C)$ may yield a recursive call to $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C \xrightarrow{\mathtt{m}}_{\texttt{lnode}} \texttt{next})$ and so on, until the argument becomes zero. This program essentially deconstructs $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C)$ into its terminal (scalar) values and reconstructs a `List` value equal to the value represented by $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C)$. We call this program a *deconstruction program* based on the lifting constructor $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}$.

To check if $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\texttt{l}_C)$, we instead check if a bisimulation relation exists between the two respective deconstruction programs (assuming `LHS` as the precondition at entries). Similar to the top-level bisimulation search, we use a product-CFG to represent this bisimulation relation. To distinguish this product-CFG from the top-level product-CFG that relates S and C, we call this product-CFG that relates two deconstruction programs, a *deconstruction product-CFG* or *decons-PCFG* for short. The deconstruction program and the decons-PCFG for our $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}$ example are shown in fig. 5. We distinguish states between the first and second programs using superscripts: fst and snd respectively. However, these are omitted in case the states are equal in both programs (e.g., $\texttt{p}_C$). Since both $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C)$ and $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\texttt{l}_C)$ use the same lifting constructor $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}$, the PC-transition correlations of both programs are trivially obtained by unifying

the program structures. A node is created in the decons-PCFG that encodes the correlation of the entries to both programs, we call this node the *recursive-node* in the decons-PCFG (e.g., D0:D0 in fig. 5b). A recursive call becomes a back-edge in the decons-PCFG that terminates at the recursive-node. At the start of both deconstruction programs, $\mathtt{l}^{fst} = \mathtt{l}^{snd} = \mathtt{l}_C$ — the same $\mathtt{l}_C$ is passed to both deconstruction programs, only the memory states $\mathtt{m}$ and $\mathtt{m}'$ are different. The bisimulation check thus involves checking that if the invariant $\mathtt{l}^{fst} = \mathtt{l}^{snd}$ holds at the recursive-node, then during one iteration of the static programs:

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

The first check succeeds due to the invariant $\mathtt{l}^{fst} = \mathtt{l}^{snd}$. For the second and third checks, we additionally need to reason that the memory objects $\mathtt{l} \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{l} \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next}$ cannot alias with the writes (in $\mathtt{m}'$ in eq. (6)) to the newly allocated objects $\mathtt{p}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{p}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next}$. We capture this aliasing information by running an interprocedural allocation-site based points-to analysis on the original C program as well as the deconstruction programs being checked for equivalence. Our points-to analysis also splits each allocation site region, say C4 in fig. 1d, into two regions $\mathtt{C4}_1$ and $\mathtt{C4}_{2+}$, where $\mathtt{C4}_1$ represents the region of the most recent allocation and $\mathtt{C4}_{2+}$ represents the region of all other allocations through $\mathtt{malloc}_{\mathtt{C4}}$ respectively. We write $p \rightsquigarrow \{\mathtt{R}_1, \mathtt{R}_2\}$ to represent the condition that value p *may point to* an object belonging to one of the region labels $\mathtt{R}_1$ or $\mathtt{R}_2$ (but may not point to any object outside of $\mathtt{R}_1$ and $\mathtt{R}_2$). We compute may-point-to information for all program variables as well as the region labels themselves.

Our points-to analysis on the C program determines that at PC C5 of fig. 1d (the start of the product-CFG edge (S3:C5)→(S3:C3) across which the proof condition is being evaluated), the pointer to the *head* of the list, i.e., $\mathtt{l}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$. It also determines that the distinguishing writes modify memory regions belonging to $\mathtt{C4}_1$ only. Further, we get $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}\}$ 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, $\mathtt{l}^{snd} \rightsquigarrow \{\mathtt{C4}_{2+}\}$, at the recursive-node of the decons-PCFG (shown in fig. 5b). To identify such points-to invariant, we run our points-to analysis on the deconstruction

programs (fig. 5a) before comparing them for equivalence. To model procedure calls, a *supergraph* is created with control flow to and from the entry and exit of the program (e.g., dashed edges in fig. 5). We use the results of the points-to analysis on C at the PC where the proof obligation is being discharged (C5 of fig. 1c in our case). To see why $\texttt{l}^{snd} \rightsquigarrow \{\texttt{C4}_{2+}\}$ is an inductive invariant at the recursive-node: (base case) the invariant holds at entry to the decons-PCFG since it holds for $\texttt{l}_C$ and (induction step) if $\texttt{l}^{snd} \rightsquigarrow \{\texttt{C4}_{2+}\}$ holds at the entry node, it also holds at the start of a recursive call. This follows from $\texttt{C4}_{2+} \rightsquigarrow \{\texttt{C4}_{2+}\}$ (points-to information at PC C5), which ensures that $\texttt{l}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{next}$ may point to only $\texttt{C4}_{2+}$ objects. During proof obligation discharge (e.g., during the bisimulation check on decons-PCFG), the points-to invariants are encoded as SMT constraints. This allows us to successfully complete the bisimulation proof on the decons-PCFG, and consequently successfully discharge the proof obligation $\{\phi_{\texttt{S3:C5}}\}(\texttt{S3}{\rightarrow}\texttt{S5}{\rightarrow}\texttt{S3}, \texttt{C5}{\rightarrow}\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)\}$ generated due to (I4) invariant in table 1.

2.5 Summary of Proof Discharge Algorithm

Let $Solve(\texttt{LHS}, \texttt{RHS}, k, d_o, d_u)$ be the top-level procedure for discharging proof obligations. $\texttt{LHS} \Rightarrow \texttt{RHS}$ represents the proof obligation, where k, d_o and d_u are the categorization parameter, over- and under-approximation depths respectively. *Solve* returns either T or $\texttt{F}(\Gamma)$ signifying proven and disproven respectively where Γ is a set of counterexamples. We perform RHS-breaking on the lowered proof obligations and invoke *Solve* for each smaller proof obligations. Figure 6 gives a broad overview of our proof discharge algorithm.

```
Function Solve(LHS, RHS, k, d_o, d_u)
    (LHS_k, RHS_k) ↤ DecomposeAndUnroll(LHS, RHS, k);
    switch Categorize(LHS_k, RHS_k) do
        case Type I do return SMTSolve(LHS_k ⇒ RHS_k);
        case Type II do
            (LHS_o, LHS_u) ↤ Approximate(LHS, d_o, d_u);
            if SMTSolve(LHS_o ⇒ RHS_k) ≡ T then return T;
            if SMTSolve(LHS_u ⇒ RHS_k) ≡ F(Γ) then return F(Γ);
            else return F(∅);
        case Type III do
            foreach P_i ⇒ RHS_i : DecomposeAndRHSBreak(LHS, RHS) do
                if RHS_i ≡ l_1 ∼ l_2 then
                    (D_1, D_2) ↤ GetDeconstructionPrograms(l_1, l_2);
                    if CheckBisimilarity(LHS ∧ P_i, D_1, D_2) ≡ F then return F(∅);
                else
                    if Solve(LHS ∧ P_i, RHS_i, k, d_o, d_u) ≡ F(Γ) then return F(Γ);
                end
            end
            return T;
        end
    end
end
```

Figure 6: Summary of the Proof Discharge Algorithm

Table 2: String lifting constructors and their definitions.

Lifting Constructor	Definition
(T1) `Str = SInvalid | SNil | SCons(i8, Str)`	
$\texttt{Cstr}_{\mathbb{m}}^{\texttt{u8[]}}(p:\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{SInvalid}$ $\underline{\texttt{elif}}\ p[0_{\texttt{i32}}]_{\mathbb{m}}^{\texttt{i8}} = 0_{\texttt{i8}}\ \underline{\texttt{then}}\ \texttt{SNil}$ $\underline{\texttt{else}}\ \texttt{SCons}(p[0_{\texttt{i32}}]_{\mathbb{m}}^{\texttt{i8}}, \texttt{Cstr}_{\mathbb{m}}^{\texttt{u8[]}}(p + 1_{\texttt{i32}}))$
$\texttt{Cstr}_{\mathbb{m}}^{\texttt{lnode(u8)}}(p:\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{SInvalid}$ $\underline{\texttt{elif}}\ p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val} = 0_{\texttt{i8}}\ \underline{\texttt{then}}\ \texttt{SNil}$ $\underline{\texttt{else}}\ \texttt{SCons}(p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val}, \texttt{Cstr}_{\mathbb{m}}^{\texttt{lnode(u8)}}(p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{next}))$
$\texttt{Cstr}_{\mathbb{m}}^{\texttt{clnode(u8)}}(p:\texttt{i32}, i:\texttt{i2})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \underline{\texttt{then}}\ \texttt{SInvalid}$ $\underline{\texttt{elif}}\ p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{chunk}[i]_{\mathbb{m}}^{\texttt{i8}} = 0_{\texttt{i8}}\ \underline{\texttt{then}}\ \texttt{SNil}$ $\underline{\texttt{else}}\ \texttt{SCons}(p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{chunk}[i]_{\mathbb{m}}^{\texttt{i8}}, \texttt{Cstr}_{\mathbb{m}}^{\texttt{clnode(u8)}}(i = 3_{\texttt{i2}} ? p \overset{\mathbb{m}}{\rightarrow}_{\texttt{clnode}} \texttt{next} : p, i + 1_{\texttt{i2}}))$

3 Evaluation

We have implemented S2C on top of the Counter tool [18]. 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` [15], `Yices2-45e38fc` [16], 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 (specification), a C implementation, and a file that contains the precondition and postcondition. All inductive invariants at intermediate nodes in the product-CFG are inferred automatically. We consider programs involving four distinct ADTs, namely, (T1) `String`, (T2) `List`, (T3) `Tree` and (T4) `Matrix`.

3.1 Experiments

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.

```
S0: i32 strlen (Str s) {
S1:   i32 len := 0_i32;
S2:   while ¬(s is SNil):
S3:     assume ¬(s is SInvalid);
S4:     // (s is SCons)
S5:     s   := s.tail;
S6:     len := len + 1_i32;
S7:   return len;
SE: }
```

(a) Strlen Specification

```
   size_t strlen(char* s);
C0: i32 strlen (i32 s) {
C1:   i32 i := 0_i32;
C2:   while s[0_i32]^i8_m ≠ 0_i8:
C3:     s := s + 1_i32;
C4:     i := i + 1_i32;
C5:   return i;
CE: }
```

(b) Strlen Implementation using Array

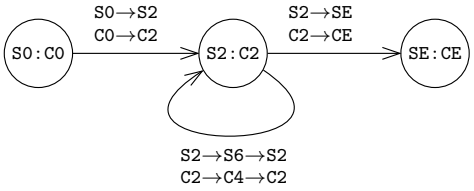
```
   typedef struct clnode {
     char chunk[4]; struct clnode* next; } clnode;
   size_t strlen(clnode* cl);
C0 : i32 strlen (i32 cl) {
C1 :   i32 hi := 0x80808080_i32; i32 lo := 0x01010101_i32;
C2 :   i32 i  := 0_i32;
C3 :   while true:
C4 :     i32 dword_ptr := &cl →^m_clnode chunk;
C5 :     i32 dword     := dword_ptr[0_i32]^i32_m;
C6 :     if ((dword − lo) & (∼ dword) & hi) ≠ 0_i32:
C7 :       if dword_ptr[0_i32]^i8_m = 0_i8: return i;
C8 :       if dword_ptr[1_i32]^i8_m = 0_i8: return i + 1_i32;
C9 :       if dword_ptr[2_i32]^i8_m = 0_i8: return i + 2_i32;
C10:       if dword_ptr[3_i32]^i8_m = 0_i8: return i + 3_i32;
C11:     cl := cl →^m_clnode next; i  := i + 4_i32;
CE : }
```

(c) Optimized Strlen Implementation using Chunked Linked List

Figure 7: Specification of Strlen along with two possible C implementations. Figure 7b is a generic implementation using a null-terminated array for `String`. Figure 7c is an optimized implementation using a chunked linked list for `String`.

3.1.1 String

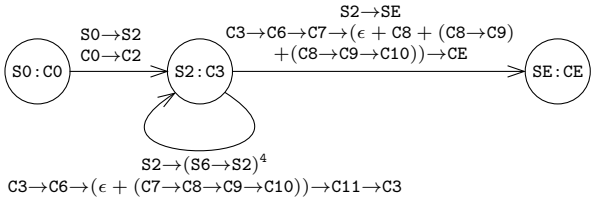
We wrote a single specification in Spec for each of the following common string library functions: `strlen`, `strchr`, `strcmp`, `strspn`, `strcspn`, and `strpbrk`. For each specification program, we took multiple C implementations of that program, drawn from popular libraries like `glibc` [3], `klibc` [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 $\texttt{Cstr}^{\texttt{u8}[]}_{\texttt{m}}$. `u<N>` represents the N-bit unsigned integer type in C. For example, `u8` represents `unsigned char` type.



(a) Product CFG for programs figs. 7a and 7b

PC-Pair	Invariants
(S0:C0)	(P) $\mathtt{s}_S \sim \mathtt{Cstr}_{\mathtt{m}}^{\mathtt{char[]}}(\mathtt{s}_C)$
(S2:C2)	(I1) $\mathtt{s}_S \sim \mathtt{Cstr}_{\mathtt{m}}^{\mathtt{char[]}}(\mathtt{s}_C)$
	(I2) $\mathtt{len}_S = \mathtt{i}_C$
(SE:CE)	(E) $\mathtt{ret}_S = \mathtt{ret}_C$

(b) Invariants Table for fig. 8a



(c) Product CFG for programs figs. 7a and 7c

PC-Pair	Invariants
(S0:C0)	(P) $\mathtt{s}_S \sim \mathtt{Cstr}_{\mathtt{m}}^{\mathtt{clnode}}(\mathtt{cl}_C, 0)$
(S2:C3)	(I1) $\mathtt{s}_S \sim \mathtt{Cstr}_{\mathtt{m}}^{\mathtt{clnode}}(\mathtt{cl}_C, 0)$
	(I2) $\mathtt{len}_S = \mathtt{i}_C$
(SE:CE)	(E) $\mathtt{ret}_S = \mathtt{ret}_C$

(d) Invariants Table fig. 8c

Figure 8: Product CFGs and Invariants Tables showing bisimulation between Strlen specification in fig. 7a and two C implementations in figs. 7b and 7c

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 $\mathtt{Cstr}_{\mathtt{m}}^{\mathtt{lnode(u8)}}$ and $\mathtt{Cstr}_{\mathtt{m}}^{\mathtt{clnode(u8)}}$ respectively. These lifting constructors are defined in table 2. $\mathtt{Cstr}_{\mathtt{m}}^{\mathtt{lnode(u8)}}$ requires a single argument p representing the pointer to the list node. On the other hand, $\mathtt{Cstr}_{\mathtt{m}}^{\mathtt{clnode(u8)}}$ 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 7 shows the `strlen` specification and two vastly different C implementations. Figure 7b 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. 7c differs from fig. 7b 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 8 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, $\mathtt{Cstr}^{\mathtt{u8[]}}$ 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[]`). `lnode(T)` represents a linked list node type containing a value of type `T`. Similarly, `clnode(T)` and `tnode(T)` represent a chunked linked list and a tree node with values of type `T`

respectively.

Table 3: List lifting constructors and their definitions.

Lifting Constructor	Definition
(T2) `List = LNil | LCons(i32, List)`	
$\texttt{Clist}_{\texttt{m}}^{\texttt{u32[]}}(p\ i\ n{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u n\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p[i]_{\texttt{m}}^{\texttt{i32}}, \texttt{Clist}_{\texttt{m}}^{\texttt{u32[]}}(p, i + 1_{\texttt{i32}}, n))$
$\texttt{Clist}_{\texttt{m}}^{\texttt{lnode(u32)}}(p{:}\texttt{i32})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val}, \texttt{Clist}_{\texttt{m}}^{\texttt{lnode}}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{next}))$
$\texttt{Clist}_{\texttt{m}}^{\texttt{clnode(u32)}}(p{:}\texttt{i32}, i{:}\texttt{i2})$	$\underline{\texttt{if}}\ p = 0_{\texttt{i32}}\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p \overset{\texttt{m}}{\rightarrow}_{\texttt{clnode}} \texttt{chunk}[i]_{\texttt{m}}^{\texttt{i32}}, \texttt{Clist}_{\texttt{m}}^{\texttt{clnode}}(i = 3_{\texttt{i2}} ? p \overset{\texttt{m}}{\rightarrow}_{\texttt{clnode}} \texttt{next} : p, i + 1_{\texttt{i2}}))$

3.1.2 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 ($\texttt{Clist}_{\texttt{m}}^{\texttt{u32[]}}$), linked list ($\texttt{Clist}_{\texttt{m}}^{\texttt{lnode(u32)}}$), and a chunked linked list ($\texttt{Clist}_{\texttt{m}}^{\texttt{clnode(u32)}}$). The lifting constructors are shown in table 3. Although similar to the String lifting constructors, these lifting constructors differ widely in their data encoding. For example, $\texttt{Clist}_{\texttt{m}}^{\texttt{u32[]}}(p, i, n)$ represents a `List` value constructed from a C array p of size n starting at the i^{th} index. The list becomes empty when we are at the end of the array. ($\texttt{Clist}_{\texttt{m}}^{\texttt{lnode(u32)}}$) and ($\texttt{Clist}_{\texttt{m}}^{\texttt{clnode(u32)}}$), 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 4: Tree lifting constructors and their definitions.

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

3.1.3 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 ($\texttt{Ctree}_{\texttt{m}}^{\texttt{u32[]}}$), and (2) a linked tree node with two pointers for the left and right children ($\texttt{Ctree}_{\texttt{m}}^{\texttt{tnode(u32)}}$) (shown in table 4). 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 5: Matrix and auxiliary List lifting constructors and their definitions.

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

3.1.4 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 `List`s ((T4) in table 5). The C implementations for a `Matrix` object include (a) a two-dimensional array ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[][]}}$), (b) a flattened row-major array ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[r]}}$), (c) a flattened column-major array ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[c]}}$), (d) a linked list of 1D arrays ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{lnode(u32[])}}$), (e) a 1D array of linked lists ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{lnode(u32)[]}}$) and (f) a 1D array of chunked linked list ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{clnode(u32)[]}}$) data layouts. Note that both `T[r]` and `T[c]` represent a 1D array of type `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, $\texttt{Clist}_{\mathbb{m}}^{\texttt{u32[r]}}$ and $\texttt{Clist}_{\mathbb{m}}^{\texttt{u32[c]}}$ for lifting each row of matrices lifted using the corresponding $\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[r]}}$ and $\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[c]}}$ `Matrix` lifting constructors. These constructors are listed in table 5.

3.2 Results

Table 6 lists the various C implementations and the time it took to compute equivalence with their specifications. For functions that take two or more data struc-

tures as arguments, we show results for different combinations of data layouts for

Table 6: Equivalence checking times and minimum under- and over-approximation depth values at which equivalence checks succeeded.

Data Layout	Variant	Time(s)	($\mathbf{d}_u, \mathbf{d}_o$)	Data Layout	Variant	Time(s)	($\mathbf{d}_u, \mathbf{d}_o$)
	list				**tree**		
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)		**matfreq**		
	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)
	strlen			lnode(u8)[]	naive	1.5k	(1,2)
u8[]	dietlibc$_s$	9	(1,2)		opt	2.3k	(4,6)
	dietlibc$_f$	44	(3,2)	clnode(u8)[]	opt	1.8k	(4,6)
	glibc	52	(3,2)		**strpbrk**		
	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)
	strchr			lnode(u8),clnode(u8)	opt	572	(4,2)
u8[]	dietlibc$_s$	16	(1,1)		**strcspn**		
	dietlibc$_f$	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$_s$	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)
	strcmp				opt	502	(4,2)
u8[],u8[]	dietlibc$_s$	39	(1,1)	lnode(u8),clnode(u8)	opt	595	(4,2)
	freebsd	39	(1,1)		**strspn**		
	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$_s$	42	(1,1)	u8[],clnode(u8)	opt	535	(4,2)
	newlib$_f$	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)				

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).

4 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 [18] 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.

5 Conclusion

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

Prior work on push-button verification of specific systems [12, 24, 22, 23] involves a combination of careful system design and automatic verification tools like SMT solvers. Constrained Horn Clause (CHC) Solvers [14] 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.

6 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. We summarize our approach and finish with the major contributions.

Chapter 2 begins with an introduction to a minimal function language 'Spec' and an intermediate representation (IR). The rest of this chapter provides a background on bisimulation relation and product program, as well as introduce terminology used in the rest of the thesis. We finish with a formal definition of equivalence.

Chapter 3 starts with proof obligations and their properties. 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 introduced in the last two chapters. We also introduce a program representation of values, called 'deconstruction program'.

Chapter 4 contains a discussion on the two major components of our algorithm: (a) a counterexample-guided correlation algorithm to search for a bisimulation relation and (b) a counterexample-guided invariant inference algorithm. These two components along with our proof discharge algorithm allow automatic end-to-end equivalence checking. We formalize handling of procedure calls, and finish with a dataflow formulation of a pointer analysis used by our equivalence checker.

Chapter 5 introduces a program graph representation of values, called 'value graphs', similar to 'deconstruction program'. 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 C implementations, each varying in its data layout and algorithmic strategy.

Finally, **Chapter 7** discusses the limitations of our algorithm and draws comparison with some related work. We note our key ideas and finish with potential improvements to our algorithm.

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] **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.

[11] **C., B.**, *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0262162098.

[12] **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.

[13] Coq:Equiv (2023). Program Equivalence in Coq. https://softwarefoundations.cis.upenn.edu/plf-current/Equiv.html.

[14] **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.

[15] **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/ETAPS'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.

[16] **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.

[17] **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.

[18] **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.

[19] **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.

[20] **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.

[21] **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.

[22] **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.

[23] **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.

[24] **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.

[25] **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.

[26] **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.