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

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 [32, 42, 39, 41, 26, 44, 45, 36, 43, 28, 24, 29, 11, 38, 15, 22, 37, 31] across low level programming languages such as C and assembly. In most of these applications, soundness in critial, 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 [40, 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.

```
type List = LNil | LCons (val:i32, tail:List).

fn mk_list_impl (n:i32) (i:i32) (l:List) : List =
  if i ≥_u n then l
             else make_list_impl(n, i+1_i32, LCons(i, l)).

fn mk_list (n:i32) : List = mk_list_impl(n, 0_i32, LNil).
```

(a) Spec Program

```
typedef struct lnode {
  unsigned val; struct lnode* next; } lnode;

lnode* mk_list(unsigned n) {
  lnode* l = NULL;
  for (unsigned i = 0; i < n; ++i) {
    lnode* p = malloc(sizeof lnode);
    p→val = i; p→next = l; l = p;
  }
  return l;
}
```

(b) C Program with `malloc()`

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

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

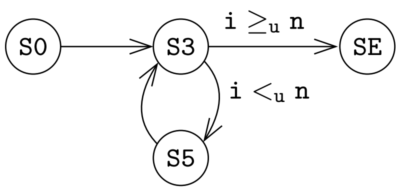
(a) (Abstracted) Spec IR

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

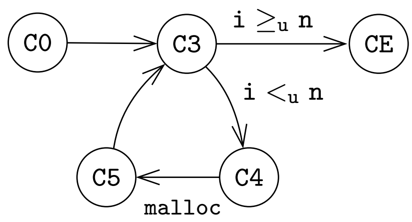
(b) (Abstracted) C IR

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 in defined at line `A0` in fig. 1a. An empty `List` is represented by the constructor `LNil`, where as 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 3a and 3b show the Control-Flow Graph (CFG) representation of the Spec and C IR programs in figs. 2a and 2b respectively. Each node represent a PC

Table 1: Node Invariants for Product-CFG in fig. 4

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) $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$
(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) $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$
(SE:CE)	(E) $\mathtt{ret}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{ret}_C)$

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. 3b, the edge C5→C3 represents the path C5→C6→C7→C8→C3.

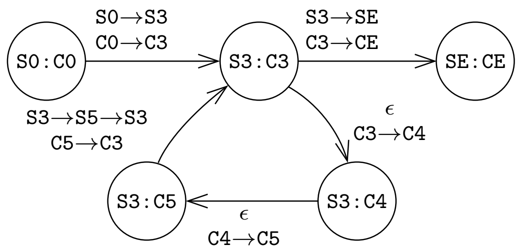


Figure 4: Product-CFG between the CFGs in figs. 3a and 3b

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* [43] and the CFG representation of a product program is called a *product-CFG*. Figure 4 shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 3a and 3b.

At each node of the product-CFG, invariants relate the states of the Spec and C program respectively. Table 1 lists invariants for the product-CFG in fig. 4. At the start node S0:C0 of the product-CFG, the precondition *Pre* (labeled (P)) ensures equality of input arguments $\mathtt{n}_S$ and $\mathtt{n}_C$ 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) $\mathtt{i}_S = \mathtt{i}_C$ is an inductive invariant.

In table 1, the invariant (I4) $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$ is an example of a recursive relation and represents equality between the Spec `List` variable $\texttt{l}_S$ and the `List` represented by chasing the `lnode` pointers starting at $\texttt{l}_C$. $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}$ 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 $\mathtt{m}$ to a Spec `List` value, and is defined as follows:

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

Product-CFG invariants involving recursive relations (e.g., (I4)) allow us to express equality between native Spec values with the C program state. Assuming that the precondition *Pre* ((P)) holds at the entry node `S0:C0`, a bisimulation check involves checking that the inductive invariants hold too, and consequently the postcondition *Post* ((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: (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 intermediate correlated PCs. (3) An algorithm for solving proof obligations containing recursive relations. Our major contributions are as follows:

- Proof Discharge Algorithm: Solving proof obligations ((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

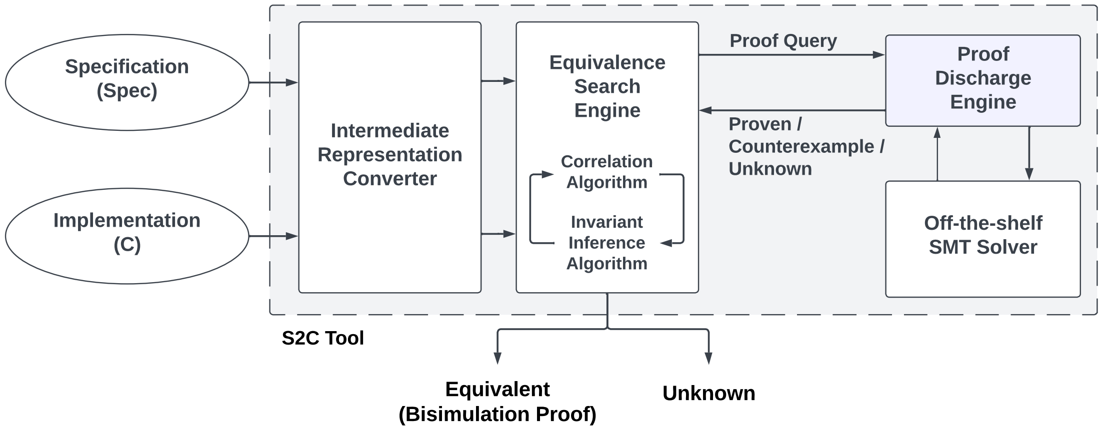


Figure 5: A brief overview of our equivalence checker algorithm S2C. The inputs to the algorithm are the Spec and C programs. S2C either successfully finds a bisimulation proof implying equivalence or soundly fails with an unknown verdict.

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 Languages and Equivalence

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:

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

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

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 succintly yet explicitly. This includes logical operators (e.g., `and`), bitvector arithmatic operators (e.g., `bvadd(+)`) and relational operators for comparing bitvectors interpreted as signed or unsigned integers (e.g., $\leq_{u,s}$).

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

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 7a and 7b 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 7a 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 7b 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 concretized,

```
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, 0_i32).
```

(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

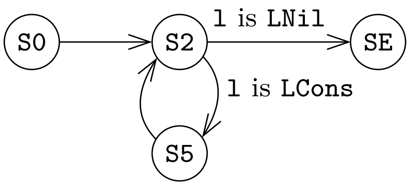
Figure 7: Spec and C Programs traversing a Linked List.

```
S0: i32 sum_list (List l) {
S1:   i32 sum := 0_i32;
S2:   while ¬(l is LNil):
S3:     // (l is LCons);
S4:     sum := sum + l.val;
S5:     l   := l.next;
S6:   return sum;
SE: }
```

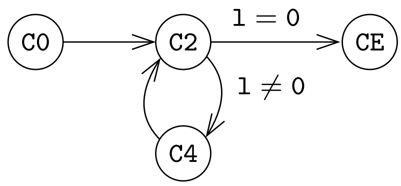
(a) (Abstracted) Spec IR

```
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



(c) CFG of Spec Program



(d) CFG of C Program

Figure 8: IRs and CFGs of the Spec and C Programs in figs. 7a and 7b respectively.

(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., $\texttt{malloc}_{\texttt{C4}}$ 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:

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

In this expanded representation of l, the *sum-deconstruction* operator '$\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$' [1] conditionally deconstructs the sum type into its variants `LNil` and `LCons`. Equation (2) 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 $\mathbb{m}$ in the IR and pointers are converted to bitvectors. Memory reads are represented using the following two C-like syntaxes: (a) "$p \xrightarrow{\mathbb{m}}_{\texttt{T}} \texttt{f}$" is equivalent to "$*(\texttt{typeof(T.f)}*)(\&\mathbb{m}[p{+}\texttt{offsetof(T,f)}])$" i.e., it returns the bytes in the memory array $\mathbb{m}$ starting at address '$p{+}\texttt{offsetof(T,f)}$' and interpreted as an object of type '`typeof(T.f)`' and (b) "$p[i]^{\texttt{T}}_{\mathbb{m}}$" is equivalent to "$*(\mathrm{T}*)(\&\mathbb{m}[p + i{\times}\texttt{sizeof(T)}])$" i.e., it returns the bytes in the memory array $\mathbb{m}$ starting at address '$p + i{\times}\texttt{sizeof(T)}$' and interpreted as an object of type '`T`'. "$\mathbb{m}[a \leftarrow v]_{\texttt{T}}$" represents an array that is equal to $\mathbb{m}$ everywhere except at addresses $[a,\ a{+}\texttt{sizeof(T)})$ which contains the value v of type '`T`'. Recall that the size and memory layout of each type is concretized in the IR, and hence the values '`offsetof(T,f)`' and '`sizeof(T)`' are constants.

Figures 8c and 8d show the Control-Flow Graph (CFG) representation of the

[1] The sum-deconstruction operator '$\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$' for an ADT T must contain exactly one branch for each value constructor of T. For example, '$\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$' for the `List` type must have exactly two branches of the form `LNil` and $\texttt{LCons}(e_1, e_2)$ for some expressions e_1 and e_2.

Spec and C IRs in figs. 8a and 8b respectively. Each CFG node represents a 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. 3b, the edge C5→C3 represents the path C5→C6→C7→C8→C3. In such a case, the transfer function of the edge is the composition of the sequence of instructions. Henceforth, We refer to the IR programs as Spec and C directly unless a distinction is necessary.

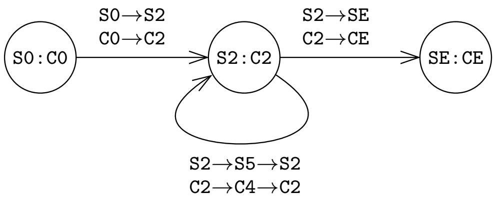
2.3 Equivalence Definition

Given (1) a Spec program specification S, (2) a C implementation C, (3) a precondition Pre that relates the initial 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[2]: 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 well-defined on $\texttt{Input}_S$, *and* C's memory allocation requests during its execution on $\texttt{Input}_C$ are successful, then both programs S and C produce outputs such that $Post(\texttt{Output}_S, \texttt{Output}_C)$ holds.

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

The $(S\ \texttt{def})$ antecedent states that we are only interested in proving equivalence for well-defined executions of S, i.e., executions that satisfy all assertions expressed using the `assuming-do` statement. Sometimes, the user may be interested in constraining the nature of inputs to C for the purpose of checking equivalence only for *well-defined* inputs. In these cases, we use a combination of Pre and $(S\ \texttt{def})$ to constrain the execution of C to inputs for which we are interested in proving equivalence. For example, the C library function $\texttt{strlen(char* str}_C\texttt{)}$ is

[2] $\texttt{Input}_C$ and $\texttt{Output}_C$ include the initial and final memory state of C respectively.



(a) Product-CFG

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

(b) Node Invariants of the Product-CFG

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

well-defined only if $\mathtt{str}_C$ represents a valid null character terminated string. This includes the assumption that the pointer $\mathtt{str}_C$ may not be null. Since Spec has no notion of pointers, we expose this conditional well-definedness of C strings through an explicit constructor e.g. `SInvalid` for the `String` ADT defined as $\mathtt{String} = \mathtt{SInvalid} \mid \mathtt{SNil} \mid \mathtt{SCons}(\mathtt{i8}, \mathtt{String})$. ($S$ `def`) asserts $\neg(\mathtt{str}_S$ is `SInvalid`) and the precondition *Pre* contains the relation ($\mathtt{str}_S$ is `SInvalid`) $\Leftrightarrow$ $(\mathtt{str}_C = 0)$. Hence, (S `def`) and *Pre* ensures that we compute equivalence only for those executions of S and C where the input strings are well-defined. A similar strategy is employed for other functions as detailed in section 5.2.

The (C `fits`) antecedent states that we prove equivalence under the assumption that C'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 ADT values. For C, observables include the returned value along with the implicit memory state at program exit. The postcondition *Post* relates these outputs of the two programs. In general, the Spec and C sources may contain multiple procedures and these procedures may have calls to each other. In that scenario, we are interested in proving equivalence of each S and C procedure pair.

2.4 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 behavior. A bisimulation relation between two programs can be represented using a *product program* [43] and the CFG representation of a product program is called a *product-CFG*. Figure 9a shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 8c and 8d.

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

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 its correlated paths. For example, the product-CFG edge (`S2:C2`)→(`S2:C2`) is formed by pairing `S2`→`S5`→`S2` and `C2`→`C4`→`C2` in figs. 8c and 8d, and represents that when S makes the transition `S2`→`S5`→`S2`, C makes the transition `C2`→`C4`→`C2` in lockstep. In general, a product-CFG edge e may correlate a finite path ρ_S in S with a finite path ρ_C in C, written $e = (\rho_S, \rho_C)$. The empty path ϵ in S may be correlated with a finite path in C However, a product-CFG is only well-formed (i.e. represents a valid bisimulation relation) if no loop path in C is correlated with ϵ in S. For example, fig. 4 shows the correlation of ϵ with the paths `C3`→`C4` and `C4`→`C5`. Since the loop path `C3`→`C4`→`C5`→`C3` in C is still correlated with the non-empty path `S3`→`S5`→`S3` in S, it represents a valid bisimulation relation.

At the start node `S0:C0` of the product-CFG in fig. 4, the precondition *Pre* (labeled P) ensures equality of input arguments $\mathtt{n}_S$ and $\mathtt{n}_C$ at programs' entry. *Inductive invariants* (labeled I) are inferred at each intermediate product-CFG node that relate the values of S with values and memory state of C. The inductive invariants are identified by running an invariant inference algorithm on the product-CFG, which is further discussed in section 4.2. At the exit node `SE:CE` of the product-CFG, the postcondition *Post* (labeled P) represents equality of observable outputs and forms our primary proof obligation. Assuming that the precondition *Pre* (P) holds at the entry node `S0:C0`, a bisimulation check involves checking that the inductive invariants (I) hold too, and consequently the postcondition *Post* (E) holds at the exit node SE:CE.

2.5 Recursive Relation

In fig. 9b, the precondition (P) is an example of a *recursive relation*: "$\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)$" where $\mathtt{l}_S$ and $\mathtt{l}_C$ represent the input variables to the Spec and C programs respectively, `lnode` is the C `struct` type that contains the `val` and `next` fields, and $\mathfrak{m}$ is the byte-addressable array representing the current memory state of the C program. $l_1 \sim l_2$ is read l_1 *is recursively equal to* l_2 and is semantically equivalent to $l_1 = l_2$. The '$\sim$' simply emphasizes that l_1 and l_2 are (possibly recursive) ADT values. $\mathtt{Clist}^{\mathtt{lnode}}$ is called a *lifting constructor* that 'lifts' a C pointer value p (pointing to an object of type `struct lnode`) and a C memory state $\mathfrak{m}$ to a (possibly infinite in case of a circular list) `List` value, and is defined through its *unrolling procedure* as follows:

$$
\begin{aligned}
U_C : \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(p\!:\!\mathtt{i32}) = {} & \underline{\mathtt{if}}\ p = 0\ \underline{\mathtt{then}}\ \mathtt{LNil} \\
& \underline{\mathtt{else}}\ \mathtt{LCons}(p \overset{\mathfrak{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{val}, \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(p \overset{\mathfrak{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{next}))
\end{aligned} \tag{3}
$$

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

2.6 Proof Obligations

The counterexample-guided algorithms for construction of the product-CFG and inference of inductive invariants are discussed later in **??**. For now, we discuss the proof obligations that arise from a given product-CFG. Consider the product-CFG in fig. 4. Recall that a bisimulation check involves checking that all inductive invariants and the postcondition *Post* hold at each product-CFG node.

We use relational Hoare triples to express these proof obligations [12, 23]. If ϕ denotes a predicate relating the machine states of S and C, then for a product-CFG edge $e = (\rho_S, \rho_C)$, $\{\phi_s\}(e)\{\phi_d\}$ denotes the condition: if any machine states σ_S and σ_C of programs S and C are related through precondition $\phi_s(\sigma_S, \sigma_C)$ and the paths ρ_S and ρ_C are executed in S and C respectively, then execution terminates normally in states $\sigma_S^{'}$ (for S) and $\sigma_C^{'}$ (for C) and postcondition $\phi_d(\sigma_S^{'}, \sigma_C^{'})$ holds.

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

$$(\phi_s \land pathcond_{\rho_S} \land pathcond_{\rho_C} \land ubfree_{\rho_S}) \Rightarrow \mathtt{WP}_{\rho_S,\rho_C}(\phi_d) \tag{4}$$

Here, $pathcond_{\rho_X}$ represent the condition that path ρ is taken in program X and $ubfree_{\rho_S}$ represents the condition that execution of S along path ρ_S is free of undefined behaviour. $\mathtt{WP}_{\rho_S,\rho_C}(\phi_d)$ represents the weakest-precondition of the predicate ϕ_d across the product-CFG edge $e = (\rho_S, \rho_C)$. We will use '`LHS`' and '`RHS`' to refer to the antecedent and consequent of the implication operator '$\Rightarrow$' in eq. (4).

For example, checking that the loop invariant (I2) $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$ holds at `S2:C2` in fig. 9a requires us to prove the following two proof obligations: (1) $\{\phi_{\mathtt{S0:C0}}\}(\mathtt{S0}{\rightarrow}\mathtt{S2}, \mathtt{C0}{\rightarrow}\mathtt{C2})\{\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)\}$ and (2) $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2}{\rightarrow}\mathtt{S5}{\rightarrow}\mathtt{S2}, \mathtt{C2}{\rightarrow}\mathtt{C4}{\rightarrow}\mathtt{C2})\{\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)\}$. The proof obligation (2) reduces to the following first-order logic proof obligation:

$$\begin{aligned} &\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C) \land \mathtt{sum}_S = \mathtt{sum}_C \land (\mathtt{l}_S \text{ is } \mathtt{LCons}) \land (\mathtt{l}_C \neq 0) \\ &\qquad \Rightarrow \mathtt{l}_S.\mathtt{next} \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next}) \end{aligned} \tag{5}$$

Due to the presence of recursive relations, these proof queries (e.g., eq. (5)) cannot be solved directly by off-the-shelf solvers and require special handling. The next chapter illustrates our proof discharge algorithm for solving proof queries involving recursive relations.

3 Proof Discharge Algorithm through Illustative Examples

This chapter demonstrates our proof discharge algorithm through examples. We consider proof obligations generated due to invariants shown in table 1 and fig. 9b.

3.1 Properties of Proof Discharge Algorithm

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

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

Resolving the truth value of a proof obligation that contains a recursive relation such as $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$ is unclear. Fortunately, the shapes of the proof obligations generated by our equivalence checker are restricted. Our equivalence checking algorithm ensures that, for an invariant $\phi_s = (\phi^{\mathtt{1}}_{\mathtt{s}} \wedge \phi^{\mathtt{2}}_{\mathtt{s}} \wedge \ldots \wedge \phi^{\mathtt{k}}_{\mathtt{s}})$, 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 $\mathtt{WP}_e(\phi_d)$ (as shown in eq. (4)) to a first-order logic formula of the following form:

$$(\eta_1^l \wedge \eta_2^l \wedge \ldots \wedge \eta_m^l) \Rightarrow (\eta_1^r \wedge \eta_2^r \wedge \ldots \wedge \eta_n^r) \tag{6}$$

Thus, due to the conjunctive recursive relation property of ϕ_s and ϕ_d, any recursive relation in eq. (6) 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. (6) into multiple smaller proof obligations of the form $P_j : (\mathtt{LHS} \Rightarrow \eta_{\mathtt{j}}^{\mathtt{r}})$, 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*.

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

3.2 Iterative Unification and Rewriting Procedure

We begin with some definitions. An expression e whose top-level constructor is a lifting constructor, e.g., $e = \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$, is called a *lifted expression*. An expression e of the form $v.\mathtt{a_1}.\mathtt{a_2}...\mathtt{a_n}$ i.e. a variable with *zero* or more *accessor*-operators applied on it, is called a *pseudo-variable*. Note that, a variable v is a pseudo-variable. An expression e in which (a) all accessors (e.g., '_`.tail`') appear in a pseudo-variable and (b) each *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. The internal nodes of e represents 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 and ADT type), the scalar expressions (of `Unit`, `Bool` and `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 the node v. The *expression path condition* represents the conjunction of all the **if** conditions (if the **then** branch of taken along the path), or their negation (if the **else** branch is taken along the path). For example, in the 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 ADT 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 constructor in e_1 and e_2 is the same value constructor (in which case an unification is attempted for each of their children), *or* the top-level constructor in one of e_1 or e_2 is **if**-**then**-**else**.

If the top-level constructor of exactly one of e_1 and e_2 (say e_1) is **if**-**then**-**else**, then e_2 must have a value constructor at its root. In such a case, we *rewrite* e_2 using **if**-**then**-**else** such that one of the branches contain $e2$ under the condition *true* and all other branches have a *false* condition. the condition of the branch containing e_2 is *true* while all other branches have a *false* condition. For example, we can rewrite `LCons`(e_1, e_2) to **if** $false$ **then** `LNil` **else** `LCons`(e_1, e_2). Next, we unify each child (condition and branch expressions) of the top-level **if**-**then**-**else** operators of (possibly rewritten) e_1 and e_2. Whenever we descend down an **if**-**then**-**else** operator, we keep track of the expression path conditions for both expressions. Recall that the **if**-**then**-**else** operator for an ADT T must have exactly one branch for each value constructor of T. Moreover, the branch associated with the value constructor V must contain an expression whose top-level constructor is V.

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 of a lifted expression. Every (pseudo-)variable is associated with its unrolling procedure governed by its ADT.

For example, the unrolling procedure for `List` variable l is U_S (eq. (2)). For lifted expressions, the unrolling procedure is given by the its definition, e.g., U_C (eq. (3)) for the lifting constructor $\texttt{Clist}^{\texttt{lnode}}$.

Given two expressions e_a and e_b at expression path conditions p_a and p_b respectively, an *iterative unification and rewriting 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 begins with an attempt to unify e_a and e_b. If this unification fails, we return a failure for the original expressions e_a and e_b. Else, we obtain correlation tuples between atoms and expressions (with their expression path conditions). If the unification correlates an atom a_1 at expression path condition p_1 with another atom a_2 at expression path condition p_2, we add (p_1, a_1, p_2, a_2) to the final output. Otherwise, if the unification correlates an atom a_1 at expression path condition p_1 to a non-atomic expression e_2 at expression path condition p_2, we *rewrite* a_1 using its unrolling procedure to obtain expression e_1. The unification algorithm then proceeds by unifying e_1 and e_2 through a recursive call to $\Theta(e_1, e_2, p_1, p_2)$. The maximum number of rewrites performed by $\Theta(e_a, e_b, p_a, p_b)$ (before termination) is upper bounded by the sum of number of ADT value constructors in e_a and e_b.

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

$$l_1 \sim l_2 \Leftrightarrow \bigwedge_{i=1}^{n} (p_1^i \wedge p_2^i \rightarrow (a_1^i = a_2^i)) \tag{7}$$

For example, the unification of '$\underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(0, l_1)$' and '$\underline{\texttt{if}}\ c_2\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(i, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$' yields the correlation tuples: $(true, true, c_1, c_2)$, $(\neg c_1, \neg c_2, 0, i)$ and $(\neg c_1, \neg c_2, l_1, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$. Hence, the recursive relation "$\underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(0, l_1) \sim \underline{\texttt{if}}\ c_2\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(i, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$" decomposes into $(c_1 = c_2) \wedge (\neg c_1 \wedge \neg c_2 \rightarrow 0 = i) \wedge (\neg c_1 \wedge \neg c_2 \rightarrow l_1 \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$. Similarly, the decomposition of $l_1 \sim \texttt{LCons}(42, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$ is given by $(l_1\ \text{is}\ \texttt{LCons}) \wedge (l_1\ \text{is}\ \texttt{LCons} \rightarrow l_1.\texttt{val} = 42) \wedge (l_1\ \text{is}\ \texttt{LCons} \rightarrow l_1.\texttt{next} \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$. In case of a failed unification, the *decomposition* is defined to be *false*, e.g., $\texttt{LNil} \sim \texttt{LCons}(0, l)$ decomposes into *false*.

Each conjunctive clause of the form $(p_1^i \wedge p_2^i \rightarrow (a_1^i = a_2^i))$[3] in the decomposition is called a *decomposition clause*. A decomposition clause may relate only atomic values, i.e., it may relate either (a) two scalars or (b) two ADT variable(s) and/or lifted expression(s). However, we restrict recursive relation invariants to a shape such that each recursive relation in its decomposition strictly relates ADT values to lifted expressions only. This is discussed in more detail along with all other invariant shapes in section 4.2. We *decompose* a recursive relation by replacing it with its decomposition. We *decompose* a proof obligation P to P_D by decomposing all recursive relations in P.

3.3 Categorization of Proof Obligations

We *unroll a recursive relation* $l_1 \sim l_2$ by rewriting the top-level expressions l_1 and l_2 through their unrolling procedures (if possible) and decomposing it. We *unroll an expression* e by unrolling each recursive relation in e. More generally, the k-unrolling of e is found by unrolling the $(k-1)$-unrolling of e recursively. For a decomposed proof obligation P_D : $\texttt{LHS} \Rightarrow \texttt{RHS}$, we identify its k-unrolling (say P_K), where k is a fixed parameter called the *unrolling parameter*. After k-unrolling, we *eliminate* those decomposition clauses $(p_1 \wedge p_2 \rightarrow (a_1 = a_2))$ in P_K whose $(p_1 \wedge p_2)$ evaluates to false under LHS ignoring all recursive relations, yielding an equivalent proof obligation, say P_E. For example, the one-unrolling of P : $\texttt{LHS} \Rightarrow l \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(0)$, after elimination, yields P_E : $\texttt{LHS} \Rightarrow l$ is LNil. We categorize a proof obligation P : $\texttt{LHS} \Rightarrow \texttt{RHS}$ based on the k-unrolled form of its decomposition (i.e. P_E) as follows:

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

The categorization method is *sound* as long as the elimination of decomposition clauses is sound (but possibly not precise). In other words, it is possible that we

[3]If a_1^i and a_2^i are ADT values, then we replace $a_1^i = a_2^i$ with $a_1^i \sim a_2^i$.

are unable to eliminate a recursive relation in P_K, due to an imprecise algorithm for elimination of decomposition clauses. However, our proof discharge algorithm remains sound irrespective of such imprecision during categorization. Henceforth, we will simply use k-unrolling of P to refer to P_E directly. Next, we describe the algorithm for each type of proof obligations in sections 3.4 to 3.6.

3.4 Handling Type I Proof Obligations

In fig. 4, 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, $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$ holds at $(\texttt{S3:C3})$: $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0}\rightarrow\texttt{S3}, \texttt{C0}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)\}$. 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, $\texttt{LNil}$ is obtained for $\texttt{l}_S$ and $\texttt{0}$ (null) is obtained for $\texttt{l}_C$. The one-unrolled form of this proof obligation yields $\texttt{n}_S = \texttt{n}_C \Rightarrow \textit{true}$ 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 (with CFG fig. 3a) to $\texttt{S0}\rightarrow\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}$ here. In this case, the corresponding first-order logic formula evaluates to: $\texttt{n}_S = \texttt{n}_C \wedge 0 <_u \texttt{n}_S \Rightarrow \texttt{LCons}(0, \texttt{LNil}) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(0)$., where $(0 <_u \texttt{n}_S)$ is the path condition for the path $\texttt{S0}\rightarrow\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}$. One-unrolling of this proof obligation decomposes RHS into false due to failed unification of LCons and LNil. The proof obligation is further discharged using an SMT solver which provides a counterexample (model) that evaluates the formula to false. For example, the counterexample $\{\texttt{n}_S \mapsto 42, \texttt{n}_C \mapsto 42\}$ evaluates this formula to false. These counterexamples assist in faster convergence of our correlation search and invariant inference procedures (as we will discuss later in **??** and section 4.2).

Thus for type I queries, k-unrolling reduces all recursive relations in the original proof obligation into scalar equalities. The resulting query is further discharged using an SMT solver. Please refer to **????** 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 the proof obligation evaluates to true or false. If it evaluates to false, we also obtain counterexamples.

3.5 Handling Type II Proof Obligations

Consider the proof obligation originating due to (I2) invariant $\texttt{sum}_S = \texttt{sum}_C$ across edge $(\texttt{S2:C2}) \rightarrow (\texttt{S2:C2})$ in fig. 9a: $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}, \texttt{C2}\rightarrow\texttt{C4}\rightarrow\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$, where the node invariant S2:C2 contains the recursive relation $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$. The corresponding (simplified) first-order logic formula for this proof obligation is: $(\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \wedge \texttt{sum}_S = \texttt{sum}_C \wedge \texttt{l}_S \text{ is } \texttt{LCons} \wedge \texttt{l}_C \neq 0) \Rightarrow (\texttt{sum}_S + \texttt{l}_S.\texttt{val}) = (\texttt{sum}_C + \texttt{l}_C \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val})$. 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 $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$. 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 CFG fig. 8c) with one iteration of the loop in program C (with CFG fig. 8d): $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}, \texttt{C2}\rightarrow\texttt{C4}\rightarrow\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$. The equivalent first-order logic formula is: $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \wedge \texttt{sum}_S = \texttt{sum}_C \wedge \texttt{l}_S \text{ is } \texttt{LCons} \wedge \texttt{l}_S.\texttt{tail} \text{ is } \texttt{LCons} \Rightarrow (\texttt{sum}_S + \texttt{l}_S.\texttt{val} + \texttt{l}_S.\texttt{tail}.\texttt{val}) = (\texttt{l}_C + \texttt{l}_C \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val})$. Similar to the prior proof obligation, its equivalent first-order logic formula contains a recursive relation in the LHS. Clearly, this proof obligation should evaluate to false. Whenever a proof obligation evaluates to false, we expect an ideal proof discharge algorithm to generate counterexamples that falsify the proof obligation. Let's call this requirement (R2). Recall that these counterexamples help in faster convergence of our correlation search and invariant inference procedures.

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 $\phi_s^{u_{d_2}}$ represent the over- and under-approximated versions of precondition ϕ_s respectively, and d_1 and d_2 represent *depth parameters* that indicate the degree of over- and under-approximation. To explain our over- and under-approximation scheme, we first introduce the notion of *depth of an ADT value*.

3.5.1 Depth of ADT Values

To define the depth of an ADT value v, we view the value as a tree $\mathcal{T}(v)$. This tree representation is similar to the one briefly introduced in section 3.2. The internal nodes of $\mathcal{T}(v)$ represent ADT value constructors and the leafs (also called *terminals*) represent scalar values (i.e. boolean and bitvector literals). The depth of a value constructor or a scalar in v is simply the depth of its associated node in $\mathcal{T}(v)$. The *depth of ADT value* v is defined as the depth of $\mathcal{T}(v)$. For example, the depth of $\texttt{LCons}(1, \texttt{LCons}(4, \texttt{LNil}))$ is 2, where as the depth of the literal 1 is 1. **??** shows the tree representation and depths for different values.

3.5.2 Overapproximation and Underapproximation of Recursive Relations

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

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

$$l_1 \approx_a l_2 \Rightarrow l_1 \approx_b l_2 \Rightarrow l_1 \sim l_2 \Rightarrow l_1 \sim_b l_2 \Rightarrow l_1 \sim_a l_2 \tag{8}$$

3.5.3 SMT Encoding of Approximate Recursive Relations

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

- $l_1 \sim_d l_2$ is equivalent to the condition that the tree structures of l_1 and l_2 are isomorphic till depth d *and* the corresponding terminal values in both d-depth isomorphic structures are also equal. Note that these conditions only require scalar equalities. $l_1 \sim_d l_2$ can be identified through a *d-depth bounded* iterative unification and rewriting procedure described in section 3.2. In this modified algorithm, We eagerly expand both expressions through rewriting and collect all correlation tuples till depth d. Finally, we only keep those correlation tuples that relate scalar values and discard the recursive relations.

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

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

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

$p \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next} = 0)$. Finally, $l \approx_2 \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(p) \Leftrightarrow \Gamma_2(l) \wedge \Gamma_2(\mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(p)) \wedge l \sim_2 \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(p)$.

3.5.4 Summary of Type II Proof Discharge Algorithm

We over- (under-) approximate a precondition ϕ till depth d by d-depth over- (under-) approximating each recursive relation occuring in ϕ. Due to the conjunctive recursive relation property (section 3.1), the over- and under-approximation of ϕ are also weaker and stronger conditions compared to ϕ respectively. For a type II proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$, we first submit the proof obligation $(P_{pre-o} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\})$ to the SMT solver. Recall that the precondition $\phi_s^{o_{d_1}}$ is the d_1-depth overapproximated version of ϕ_s. If the SMT solver evaluates P_{pre-o} to true, then we return true for the original proof obligation P — if the Hoare triple with an overapproximate precondition holds, then the original Hoare triple also holds.

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

Finally, if the SMT solver evaluates P_{pre-u} to true, then we have neither proven nor disproven P. In this case, we imprecisely (but soundly) return false for the original proof obligation P (without counterexamples). Note that both approximations of P strictly fall in type I and are discharged as discussed in section 3.4. Revisiting our examples, the proof obligation $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2} \rightarrow \mathtt{S5} \rightarrow \mathtt{S2}, \mathtt{C2} \rightarrow \mathtt{C4} \rightarrow \mathtt{C2})\{\mathtt{sum}_S = \mathtt{sum}_C\}$ is provable using a depth $\mathtt{1}$ overapproximation of the precondition $\phi_{\mathtt{S2:C2}}$ — the depth $\mathtt{1}$ overapproximation retains the information that the first value in lists $\mathtt{l}_S$ and $\mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)$ are equal, and that is sufficient to prove that the new values of $\mathtt{sum}_S$ and $\mathtt{sum}_C$ are also equal (given that the old values are equal, as encoded in $\phi_{\mathtt{S2:C2}}$).

Similarly, the proof obligation $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\to\texttt{S5}\to\texttt{S2}\to\texttt{S5}\to\texttt{S2}, \texttt{C2}\to\texttt{C4}\to\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$ evaluates to false (with counterexamples) using a depth 2 underapproximation of the precondition $\phi_{\texttt{S2:C2}}$. In the depth |tt 2 underapproximate version, we try to prove that if the equal lists $\texttt{l}_S$ and $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$ have exactly two nodes[4], then the sum of the two values in $\texttt{l}_S$ is equal to the value stored in the first node in $\texttt{l}_C$. This proof obligation will return counterexample(s) that map program variables to their concrete values. The following is a possible counterexample to the depth 2 underapproximate proof obligation.

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

This counterexample maps variables to values (e.g., $\texttt{sum}_S$ maps to an `i32` value 3 and $\texttt{l}_S$ maps to a `List` value LCons(42,LCons(43,LNil)). It also maps the C program's memory state $\mathbb{m}$ 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). All other addresses (except the ones for which an explicit mapping is available), $\mathbb{m}$ provides a default byte-value 77 (shown as `()` $\mapsto$ `77`) in this counterexample.

This counterexample satisfies the preconditions $\texttt{l}_S \approx_2 \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$, $\texttt{sum} = \texttt{sum}_{CS}$ and the path conditions. Further, when the paths $\texttt{S2}\to\texttt{S5}\to\texttt{S2}\to\texttt{S5}\to\texttt{S2}$ and $\texttt{C2}\to\texttt{C4}\to\texttt{C2}$ are executed starting at the machine state represented by this counterexample, the resulting values of $\texttt{sum}_S$ and $\texttt{sum}_C$ 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).

[4]The underapproximation restricts both lists to have at most two nodes; the path condition for $\texttt{S2}\to\texttt{S5}\to\texttt{S2}\to\texttt{S5}\to\texttt{S2}$ additionally restricts $\texttt{l}_S$ to have at least two nodes. Together, this is equivalent to the list having exactly two nodes

3.6 Handling Type III Proof Obligations

In fig. 4, consider a proof obligation generated across the product-CFG edge $(\mathtt{S3}\!:\!\mathtt{C5}) \rightarrow (\mathtt{S3}\!:\!\mathtt{C3})$ while checking if the (I4) invariant, $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)$, holds at $(\mathtt{S3}\!:\!\mathtt{C3})$: $\{\phi_{\mathtt{S3:C5}}\}(\mathtt{S3}\rightarrow\mathtt{S5}\rightarrow\mathtt{S3}, \mathtt{C5}\rightarrow\mathtt{C3})\{\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)\}$. Here, a recursive relation is present both in the precondition $\phi_{\mathtt{S3:C5}}$ ((I8)) and in the postcondition ((I4)) and we are unable to remove them after k-unrolling. When lowered to first-order logic through $\mathtt{WP}_{\mathtt{S3}\rightarrow\mathtt{S5}\rightarrow\mathtt{S3},\mathtt{C5}\rightarrow\mathtt{C3}}$, this translates to (showing only relevant relations):

$$\begin{aligned}(\mathtt{i}_S = \mathtt{i}_C \wedge \mathtt{p}_C = \mathtt{malloc}() \wedge \mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C))\\ \Rightarrow (\mathtt{LCons}(\mathtt{i}_S, \mathtt{l}_S) \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}'}(\mathtt{p}_C))\end{aligned} \tag{9}$$

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

$$\mathfrak{m}' \equiv \mathfrak{m}[\&(\mathtt{p}_C \overset{\mathfrak{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{val}) \leftarrow \mathtt{i}_C]_{\mathtt{i32}}[\&(\mathtt{p}_C \overset{\mathfrak{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{next}) \leftarrow \mathtt{l}_C]_{\mathtt{i32}} \tag{10}$$

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

3.6.1 LHS-to-RHS Substitution and RHS Decomposition

We start by utilizing the $\sim$ relationships in the LHS (antecedent) of '$\Rightarrow$' to rewrite eq. (9) so that the ADT variables (e.g., $\mathtt{l}_S$) in its RHS (consequent) are substituted with the lifted C values (e.g., $\mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)$). Thus, we rewrite eq. (9) to:

$$\begin{aligned}(\mathtt{i}_S = \mathtt{i}_C \wedge \mathtt{p}_C = \mathtt{malloc}() \wedge \mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C))\\ \Rightarrow (\mathtt{LCons}(\mathtt{i}_S, \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)) \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}'}(\mathtt{p}_C))\end{aligned} \tag{11}$$

Next, we decompose the RHS by decomposing the recursive relation in the RHS followed by RHS-breaking. This process reduces eq. (11) into the following smaller proof obligations (LHS denotes the antecedent of the proof obligation in eq. (11)): (a) $\mathtt{LHS} \Rightarrow \neg(\mathtt{p}_C = 0)$, (b) $\mathtt{LHS} \wedge \neg(\mathtt{p}_C = 0) \Rightarrow (\mathtt{i}_S = \mathtt{p}_C \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val})$, and (c) $\mathtt{LHS} \wedge \neg(\mathtt{p}_C = 0) \Rightarrow (\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C) \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{p}_C \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{next}))$

The first two proof obligations fall in type II and are discharged through over- and under-approximation schemes as discussed in section 3.5:

1. The first proof obligation with postcondition $\neg(\mathtt{p}_C = 0)$ evaluates to *true* because the LHS ensures that $\mathtt{p}_C$ is the return value of an allocation function (`malloc`) which must be non-zero due to the (C `fits`) assumption.
2. The second proof obligation with postcondition $(\mathtt{i}_S = \mathtt{p}_C \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val})$ also evaluates to *true* because $\mathtt{i}_C$ is written at address $\&\mathtt{p}_C \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{val}$ in $\mathbb{m}'$ (eq. (10)) and the LHS ensures that $\mathtt{i}_S = \mathtt{i}_C$.

For ease of exposition, we simply the postcondition of the third proof obligation by rewriting $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{p}_C \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{next}))$ to $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C)$. This simplification is valid because $\mathtt{l}_C$ is written to address $\&\mathtt{p}_C \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{next}$ in $\mathbb{m}'$ (eq. (10)). Also, we have already shown that $\neg(\mathtt{p}_C = 0)$ holds due to the (C `fits`) assumption. This simplification-based rewriting is only done for ease of exposition, and has no effect on the operation of the algoritm. Thus, the third proof obligation can be rewritten as a recursive relation between two lifted expressions:

$$\mathtt{LHS} \Rightarrow \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C) \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C) \tag{12}$$

Hence, we are interested in proving equality between two `List` values lifted from C values under a precondition. Next, we show how the above can be reposed as the problem of showing equivalence between two procedures through bisimulation.

3.6.2 Equality of Values to Equivalence of Programs

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

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

Proof Sketch. The proof follows from noting that the only observables of the deconstruction programs are their output `List` values. Also, the value represented by a lifted expression is identical to the output of its deconstruction program. Thus, a successful equivalence proof ensures equal values represented by the lifting constructors and vice versa. □

Thus, to check if $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C)$ holds; we instead check if a bisimulation relation exists between their respective deconstruction programs (implying equivalence). Theorem 1 generalizes to arbitrary lifting constructors with potentially different C values and memory states.

3.6.3 Checking Bisimulation between Deconstruction Programs

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

The deconstruction programs and their decons-PCFG for the proof obligation eq. (12) are shown in fig. 10. 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., $\texttt{p}_C$). To check bisimulation

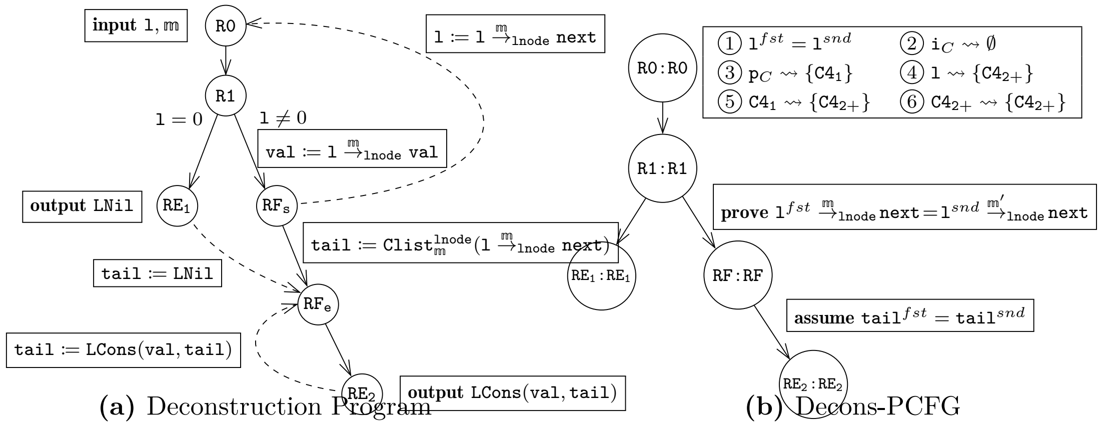


Figure 10: The deconstruction program for $\mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{l}_C)$ and decons-PCFG between deconstruction programs of $\mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{l}_C)$ and $\mathtt{Clist}_{m'}^{\mathtt{lnode}}(\mathtt{l}_C)$. In fig. 10a, D0 represents the unrolling procedure entry node, and the square boxes show the transfer functions of the unrolling procedure (eq. (3)). The dashed edges represent a recursive function call. In fig. 10b, the square box to the right of node D0:D0 contains the inferred invariants for this decons-PCFG.

between the programs that deconstruct $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$ and $\mathtt{Clist}_{\mathbb{m}'}^{\mathtt{lnode}}(\mathtt{l}_C)$, the decons-PCFG correlates one unrolling of the first program with one unrolling of the second program, as defined by the unrolling procedure in eq. (3). Thus, the PC-transition correlations of both programs are trivially obtained by unifying the static program structures. A node is created in the decons-PCFG that encodes the correlation of the entries of both programs; we call this node the *recursive-node* in the decons-PCFG (e.g., R0:R0 in fig. 10b). A recursive call becomes a back-edge in the recons-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 $\mathbb{m}^{fst} = \mathbb{m}$ and $\mathbb{m}^{snd} = \mathbb{m}'$ 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 unrolling procedure in both 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 observable values (that are used in the construction of the list) are equal: $((\mathtt{l}^{fst} \neq 0) \wedge (\mathtt{l}^{snd} \neq 0)) \Rightarrow (\mathtt{l}^{fst} \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{val} = \mathtt{l}^{snd} \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val})$.

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: $((\mathtt{l}^{fst} \neq 0) \wedge (\mathtt{l}^{snd} \neq 0)) \Rightarrow (\mathtt{l}^{fst} \overset{\mathtt{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{next} = \mathtt{l}^{snd} \overset{\mathtt{m}'}{\rightarrow}_{\mathtt{lnode}} \mathtt{next})$.

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 $\mathtt{l} \overset{\mathtt{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{l} \overset{\mathtt{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{next}$ cannot alias with the writes (in $\mathtt{m}'$ in eq. (10)) to the newly allocated objects $\mathtt{p}_C \overset{\mathtt{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{p}_C \overset{\mathtt{m}}{\rightarrow}_{\mathtt{lnode}} \mathtt{next}$. We capture this aliasing information using a points-to analysis described next in section 3.6.4.

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

3.6.4 Points-to Analysis

To reason about aliasing (as required during bisimulation check in section 2.4), we conservatively compute 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 $p \rightsquigarrow \{\mathtt{R}_1, \mathtt{R}_2\}$ 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 populate the set of all region labels using *allocation sites* of the C program i.e., PCs where a call to `malloc` occurs. For example, `C4` in fig. 2b is an allocation site. For each allocation site A, we create two region labels: (a) the first region label, called $\mathtt{A}_1$, identifies the set of memory objects that were allocated by the most recent execution of A, and (b) the second region label, called $\mathtt{A}_{2+}$, identifies the set of memory objects that were allocated by older (not the most recent) executions of A. We also include a special heap region, $\mathcal{H}$ to represent the rest of the memory not covered by the allocation site regions.

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

The may-point-to information is computed not just for program values (e.g., $\texttt{p}_C$, $\texttt{l}_C$,...) but also for each region label. For region labels $\texttt{R1}$, $\texttt{R2}$ and $\texttt{R3}$: $\texttt{R1} \rightsquigarrow \{\texttt{R2}, \texttt{R3}\}$ represents the condition that the values (pointers) stored in objects identified by $\texttt{R1}$ may point to objects identified by either $\texttt{R2}$ or $\texttt{R3}$ (but not to any other object outside $\texttt{R2}$ and $\texttt{R3}$). In fig. 2b, at PC $\texttt{C7}$, we get $\texttt{C4}_1 \rightsquigarrow \{\texttt{C4}_{2+}\}$ and $\texttt{C4}_{2+} \rightsquigarrow \{\texttt{C4}_{2+}, \mathcal{H}\}$. The condition $\texttt{C4}_1 \rightsquigarrow \{\texttt{C4}_{2+}\}$ holds because the $\texttt{next}$ pointer of the object pointed-to by $\texttt{p}_C$ (which is a $\texttt{C4}_1$ object at $\texttt{C7}$) may point to a $\texttt{C4}_{2+}$ object (e.g., object reachable from chasing the pointer $\texttt{l}_C$). On the other hand, pointers within a $\texttt{C4}_{2+}$ object may not point to a $\texttt{C4}_1$ object.

3.6.5 Transferring Points-to Information to Recons-PCFG

Recall that in section 3.6.2, we reduce the condition $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C)$ to an equivalence check. Also, recall that we discharge the equivalence check through construction of a decons-PCFG encoding the lockstep execution between the two deconstruction programs. During this bisimulation check, we need to prove that, $\texttt{l} \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \{\texttt{val}, \texttt{next}\}$ and $\texttt{l} \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \{\texttt{val}, \texttt{next}\}$ are equal. To successfully discharge these proof obligations, it suffices to show $\texttt{l}_C$ cannot alias with the memory writes that distinguish $\mathbb{m}$ and $\mathbb{m}'$.

Our points-to analysis on the C program (in fig. 2b) determines that at PC $\texttt{C5}$ (i.e. start of the product-CFG edge $(\texttt{S3:C5}) \rightarrow (\texttt{S3:C3})$ across which the proof obligation is generated), the pointer to the *head* of the list, i.e. $\texttt{l}_C \rightsquigarrow \{\texttt{C4}_{2+}\}$. It also determines that the distinguishing writes modify memory regions belonging to $\texttt{C4}_1$ only. Further, we get $\texttt{C4}_{2+} \rightsquigarrow \{\texttt{C4}_{2+}\}$ at PC $\texttt{C5}$. 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: $\texttt{l}^{snd} \rightsquigarrow \{\texttt{C4}_{2+}\}$, at the recursive-node of the decons-PCFG (shown in fig. 10b). To identify such points-to invariant, we run our points-to analysis on the deconstruction programs (fig. 10a) before comparing them for equivalence. To model procedure calls, A *supergraph* is created with edges representing control flow to (and from) the entry (and exits) of the program respectively (e.g., dashes edges in fig. 10). To see why $\mathtt{l}^{snd} \rightsquigarrow \{\mathtt{C4}_{2+}\}$ is an inductive invariant at the recursive-node:

(Base case) The invariant holds at entry of the decons-PCFG because it holds for $\mathtt{l}_C$.

(Inductive step) If $\mathtt{l}^{snd} \rightsquigarrow \{\mathtt{C4}_{2+}\}$ holds at the entry node, it also holds at the start of a recursive call. This follows from $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}\}$ (points-to information at PC $\mathtt{C5}$), which ensures that $\mathtt{l}_C \xrightarrow{\mathtt{m}'}_{\mathtt{lnode}} \mathtt{next}$ may point to only $\mathtt{C4}_{2+}$ objects.

To identify such points-to invariants, we run our points-to analysis on the deconstruction programs before comparing them for equivalence. The same analysis is run for both C and the deconstruction programs. For a reconsturction program, the boundary condition (at entry) for the points-to analysis is based on the results of the points-to analysis on C at the PC where the proof obligation is being discharged. For example, the points-to information of C PC $\mathtt{C5}$ (in fig. 1b) is used during the points-to analysis on the reconstruction programs in fig. 10.

During proof query discharge, the points-to invariants are encoded as SMT constraints. This allows us to complete the bisimulation proof on the decons-PCFG in fig. 10b, and consequently, successfully discharge the proof obligation $\{\phi_{\mathtt{S3:C5}}\}(\mathtt{S3}\to\mathtt{S5}\to\mathtt{S3}, \mathtt{C5}\to\mathtt{C3})\{\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)\}$ in table 1. The points-to analysis is further discussed in section 4.4.

3.6.6 Summary of Type III Proof Discharge Algorithm

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

Upon $\mathtt{RHS}$-breaking, we obtain several smaller proof obligations, say $P_1^i : \mathtt{LHS}_1^i \Rightarrow \mathtt{RHS}_1^i$ (for $i = 1 \ldots n$). To prove P, we require *all* of these smaller proof obligations P_1^i to be provable. However, a counterexample to *any* one of these small proof obligations would also be a counterexample to the original proof obligation P. Due to decomposition and RHS-breaking, each $\texttt{RHS}_1^i$ must be a decomposition clause and relate atomic expressions. If $\texttt{RHS}_1^i$ relate two scalar values, then P_1^i is a type II proof obligation and discharged using the algorithm in **??**.

If $\texttt{RHS}_1^i$ relates two lifted expressions (i.e. a recursive relation), we check if the deconstruction programs of the two ADT values being compared can be proven to be bisimilar (assuming $\texttt{LHS}_1^i$ holds at the correlated entry nodes in the decons-PCFG). To improve the precision during bisimulation check, we transfer the points-to invariants of the C program (at the PC where the proof obligation is being discharged) to the entry of the deconstruction programs. Next, the same points-to analysis is run on the reconstruction programs to identify points-to invariants in the deconstruction programs.

If the bisimilarity check succeeds, we return *true* for P; otherwise, we imprecisely return *false* (without counterexamples). Figure 11 gives a broad overview of the entire proof discharge algorithm. The proof discharge algorithm is further discussed in **??**.

Function $\mathit{Solve}(\texttt{LHS}, \texttt{RHS}, k, d_o, d_u)$
  $(\texttt{LHS}_k, \texttt{RHS}_k) \hookleftarrow \mathit{DecomposeAndUnroll}(\texttt{LHS}, \texttt{RHS}, k)$;
  switch $\mathit{Categorize}(\texttt{LHS}_k, \texttt{RHS}_k)$ **do**
    case Type I **do return** $\mathit{SMTSolve}(\texttt{LHS}_k \Rightarrow \texttt{RHS}_k)$;
    case Type II **do**
      $(\texttt{LHS}_o, \texttt{LHS}_u) \hookleftarrow \mathit{Approximate}(\texttt{LHS}, d_o, d_u)$;
      if $\mathit{SMTSolve}(\texttt{LHS}_o \Rightarrow \texttt{RHS}_k) \equiv \texttt{T}$ **then return** T;
      if $\mathit{SMTSolve}(\texttt{LHS}_u \Rightarrow \texttt{RHS}_k) \equiv \texttt{F}(\Gamma)$ **then return** $\texttt{F}(\Gamma)$;
      else return $\texttt{F}(\emptyset)$;
    case Type III **do**
      foreach $\texttt{P}_i \Rightarrow \texttt{RHS}_i : \mathit{DecomposeAndRHSBreak}(\texttt{LHS}, \texttt{RHS})$ **do**
        if $\texttt{RHS}_i \equiv l_1 \sim l_2$ **then**
          $(D_1, D_2) \hookleftarrow \mathit{GetDeconstructionPrograms}(l_1, l_2)$;
          if $\mathit{CheckBisimilarity}(\texttt{LHS} \wedge \texttt{P}_i, D_1, D_2) \equiv \texttt{F}$ **then return** $\texttt{F}(\emptyset)$;
        else
          if $\mathit{Solve}(\texttt{LHS} \wedge \texttt{P}_i, \texttt{RHS}_i, k, d_o, d_u) \equiv \texttt{F}(\Gamma)$ **then return** $\texttt{F}(\Gamma)$;
        end
      end
      return T;
    end
  end
end

Figure 11: Summary of the Proof Discharge Algorithm

4 Spec-to-C Equivalence Checker

start with the correlation algo, then procedure calls, then invariant inference, then points-to analysis

4.1 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. 3b, `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 π contains a product-CFG node (n_S, n_C), then π 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 π and a path ρ_C in C's CFG starting at n_C (and ending at an anchor node in C) is selected. Then, the potential correlations ρ_C with paths in S's CFG are enumerated. For example, in fig. 4, at product-CFG node (`S3:C3`), we first select the C path `C3`→`C4`, and its potential correlation possibilities with paths ϵ, `S3` → `S5`, `S3` → `S5` → `S3`, `S3` → `S5` → `S3` → `S5`, ... in S are enumerated (up to an unroll factor μ).

For each enumerated correlation possibility (ρ_S, ρ_C), a separate product-CFG π' is created (by cloning π) and a new product-CFG edge $e = (\rho_S, \rho_C)$ is added to π'. 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 ρ_S and

ρ_C. For example, the node `(S3:C4)` is added to the product-CFG if it correlates paths ϵ and `C3` $\rightarrow$ `C4` starting at `(S3:C3)`. For each node s in a product-CFG π, 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 unsucessful proof obligation $\{\phi_s\}(s \rightarrow d)\{\phi_d\}$ (for some edge $s \rightarrow d$ and node d in π). Thus, by construction, these counterexamples represent concrete state pairs that may potentially occur at s during the lockstep execution encoded by π.

To evaluate the promise of a possible correlation (ρ_S, ρ_C) starting at node s in product-CFG π, 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 π generated during this search procedure, an automatic invariant inference procedure is used to identify invariants at all the nodes in π. The counterexamples obtained from the proof obligations created by this invariant inference procedure are added to the respective nodes in π; these counterexamples help rank future correlations starting at those nodes.

If after invariant inference, we realize that an intermediate candidate product-CFG π_1 is not promising enough, we backtrack and choose another candidate product-CFG π_2 and explore the potential correlations that can be added to π_2. Thus, a product-CFG is constructed one edge at a time. If at any stage, the invariants inferred for a product-CFG π_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 (**????** and sections 3.4 to 3.6). These proof obligations may be generated either at the intermediate (search) or the final (check) phases of the BFS procedure.

Table 2: Dataflow formulation for the Invariant Inference Algorithm.

Domain	$\left\{ \begin{array}{c} \phi_n \text{ is a conjunction of predicates drawn from} \\ \text{grammar in 12b, } \Gamma_n \text{ is a set of counterexamples} \end{array} \right\}$
Direction	Forward
Transfer function across edge $e = (s \to d)$	$(\phi_d, \Gamma_d) = f_e(\phi_s, \Gamma_s)$ (fig. 12a)
Meet operator $\otimes$ $(\phi_n, \Gamma_n) \leftarrow (\phi_n^1, \Gamma_n^1) \otimes (\phi_n^2, \Gamma_n^2)$	$\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2, \qquad \phi_n \leftarrow \mathit{StrongestInvCover}(\Gamma_n)$
Boundary condition	$\texttt{out[}n^{start}\texttt{]} = (Pre, \Gamma_{n^{start}})$
Initialization to $\top$	$\texttt{in[}n\texttt{]} = (\texttt{False}, \{\})$ for all non-start nodes

Function $f_e(\phi_s, \Gamma_s)$
  $\Gamma_d^{can} := \Gamma_d \cup \texttt{exec}_e(\Gamma_s)$;
  $\phi_d^{can} := \mathit{StrongestInvCover}(\Gamma_d^{can})$;
  while $\texttt{SAT}(\neg(\{\phi_s\}(e)\{\phi_d^{can}\}), \gamma_s)$ **do**
    $\gamma_d := \texttt{exec}_e(\gamma_s)$;
    $\Gamma_d^{can} := \Gamma_d^{can} \cup \gamma_d$;
    $\phi_d^{can} := \mathit{StrongestInvCover}(\Gamma_d^{can})$;
  end
  return $(\phi_d^{can}, \Gamma_d^{can})$;
end

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

$$\begin{array}{rcl} Inv & \to & \sum_i c_i v_i = c \mid v_1 \odot v_2 \\ & \mid & \alpha_S = \texttt{liftC}_m(v^C, \ldots) \end{array}$$

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

Figure 12: Transfer function f_e and Predicate grammar Inv for invariant inference dataflow analysis in table 2. Given invariants (ϕ_s) and counterexamples (Γ_s) at node s, f_e returns the updated invariants (ϕ_d) and counterexamples (Γ_d) at node d. *StrongestInvCover(*Γ*)* computes the strongest invariant cover for counterexamples Γ. $\texttt{exec}_e(\Gamma)$ (concretely) executes counterexamples Γ over edge e. $\texttt{SAT}(\phi, \gamma)$ determines the satisfiability of ϕ; if satisfiable, the models (counterexamples) are returned in output parameter γ.

4.2 Invariant Inference and Counterexample Generation

Table 2 presents our dataflow analysis for inferring invariants ϕ_n at each node n of a product-CFG, while also generating a set of counterexamples Γ_n at node n that represents the potential concrete machine states at n.

Given the invariants and counterexamples at node s (ϕ_s, Γ_s), the transfer function initializes the new candidate set of counterexamples at d (Γ_d^{can}) to the current set of counterexamples at d (Γ_d) union-ed with the counterexamples obtained by executing Γ_s on edge e ($\texttt{exec}_e$). The candidate invariant at d (ϕ_d^{can}) is computed as the strongest cover of Γ_d^{can} (*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 ϕ_d^{can} is returned alongwith the counterexamples Γ_d^{can} learned so far. Else the candidate invariant ϕ_d^{can} is weakened using the counterexamples obtained from the `SAT` query (γ) 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 (α_S) and a *lifted* ADT value from C ($\mathtt{liftC}_m(p_C)$). 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 [30, 15]. For inequality relations, *StrongestInvCover(*Γ*)* returns false iff any counterexample in Γ evaluates the relation to false — this effectively simulates the Houdini approach [21]. In case of recursive relations, *StrongestInvCover(*Γ*)* attempts to disprove the recursive relation $l_1 \sim l_2$ by evaluating its depth-η under-approximation $l_1 \sim_\eta l_2$ for each counterexample in Γ and returns false if any one of them successfully evaluates to false. η is a constant parameter of the algorithm.

4.3 Modeling Procedure Calls

A top-level procedure δ in S or C may make non-tail recursive calls, e.g., for traversing a tree data structure. Our correlation algorithm (section 4.1) 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 $A_S^{\mathtt{b}}$ and $A_S^{\mathtt{a}}$ respectively. Let $\mathtt{args}_{A_S}$ represent the values of the actual arguments of this procedure call. Let $\mathtt{ret}_{A_S}$ represent the values returned by this procedure call. Similarly, for a procedure call at PC A_C in C, let $A_C^{\mathtt{b}}$, $A_C^{\mathtt{a}}$, $\mathtt{args}_{A_C}$ and $\mathtt{ret}_{A_C}$ 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 π for these S and C program points are $A_\pi^{\mathtt{b}}$=($A_S^{\mathtt{b}}$,$A_C^{\mathtt{b}}$) and $A_\pi^{\mathtt{a}}$=($A_S^{\mathtt{a}}$,$A_C^{\mathtt{a}}$).

Recall that the recursive call at A_S (or A_C) must be a call to the top-level procedure δ. We utilize the user-supplied *Pre* and *Post* conditions for δ to obtain the desired invariants at nodes $A_\pi^{\mathtt{b}}$ and $A_\pi^{\mathtt{a}}$ in the product-CFG. We require a successful proof to *ensure* that $Pre(A_S^{\mathtt{argss}}, A_C^{\mathtt{argsc}}, m_{\mathtt{b}})$ holds at $A_\pi^{\mathtt{b}}$. Further, the proof can *assume* that $Post(A_S^{\mathtt{rets}}, A_C^{\mathtt{retc}}, m_{\mathtt{a}})$ holds at $A_\pi^{\mathtt{a}}$. Here, $m_{\mathtt{b}}$ and $m_{\mathtt{a}}$ represent the memory states in C at $A_C^{\mathtt{b}}$ and $A_C^{\mathtt{a}}$ respectively. Thus, for such recursive calls to the top-level function, we inductively prove the precondition (on the arguments of the procedure call) at $A_\pi^{\mathtt{b}}$ and assume the postcondition (on the return values of the procedure call) at $A_\pi^{\mathtt{a}}$.

4.4 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. $R_C = \bigcup_A \{A_1, A_{2+}\} \cup \{\mathtt{heap}\}$, 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 `x := c`, for constant c, the transfer function updates $\Delta(\mathtt{x}) := \emptyset$. For instruction `x := y op z` (for some arithmetic or logical operand `op`), we update $\Delta(\mathtt{x}) := \Delta(\mathtt{y}) \cup \Delta(\mathtt{z})$. For a load instruction `x := *y`, we update $\Delta(\mathtt{x})$ to $\bigcup_{R_C \in \Delta(y)} \Delta(R_C)$. For a store instruction `*x := y`, for all $R_C \in \Delta(\mathtt{x})$, 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. 10, the dashed edges represent supergraph edges. For a malloc instruction $\mathtt{x := malloc}_A\mathtt{()}$ (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 $\Delta(\mathtt{x}) := \{A_1\}$

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 Δ_P^{pc} represents the may-point-to function for program P at PC pc.

In case of a reconstruction program R, the domain of Δ 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:

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

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 3.6.5.

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 3: String lifting constructors and their definitions.

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

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 $\texttt{Cstr}^{\texttt{u8[]}}_{\mathbb{m}}$. `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 $\texttt{Cstr}^{\texttt{lnode(u8)}}_{\mathbb{m}}$ and $\texttt{Cstr}^{\texttt{clnode(u8)}}_{\mathbb{m}}$ respectively. These lifting constructors are defined in table 3. $\texttt{Cstr}^{\texttt{lnode(u8)}}_{\mathbb{m}}$ requires a single argument p representing the pointer to the list node. On the other hand, $\texttt{Cstr}^{\texttt{clnode(u8)}}_{\mathbb{m}}$ 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 13 shows the `strlen` specification and two vastly different C implementations. Figure 13b 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. 13c differs from fig. 13b 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 14 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, $\texttt{Cstr}^{\texttt{u8[]}}$ 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.,

```
i32 strlen (Str s) {
  i32 len := 0_i32;
  while ¬(s is SNil):
    assume ¬(s is SInvalid);
    // (s is SCons)
    s   := s.tail;
    len := len + 1_i32;
  return len;
}
```

(a) Strlen Specification

```
size_t strlen(char* s);
i32 strlen (i32 s) {
  i32 i := 0_i32;
  while s[0_i32]^i8_m ≠ 0_i8:
    s := s + 1_i32;
    i := i + 1_i32;
  return i;
}
```

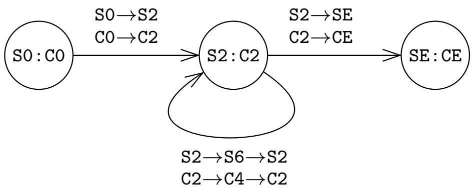
(b) Strlen Implementation using Array

```
typedef struct clnode {
  char chunk[4]; struct clnode* next; } clnode;
size_t strlen(clnode* cl);
i32 strlen (i32 cl) {
  i32 hi := 0x80808080_i32; i32 lo := 0x01010101_i32;
  i32 i  := 0_i32;
  while true:
    i32 dword_ptr := &cl →^m_clnode chunk;
    i32 dword     := dword_ptr[0_i32]^i32_m;
    if ((dword − lo) & (∼ dword) & hi) ≠ 0_i32:
      if dword_ptr[0_i32]^i8_m = 0_i8: return i;
      if dword_ptr[1_i32]^i8_m = 0_i8: return i + 1_i32;
      if dword_ptr[2_i32]^i8_m = 0_i8: return i + 2_i32;
      if dword_ptr[3_i32]^i8_m = 0_i8: return i + 3_i32;
    cl := cl →^m_clnode next; i  := i + 4_i32;
}
```

(c) Optimized Strlen Implementation using Chunked Linked List

Figure 13: Specification of Strlen along with two possible C implementations. Figure 13b is a generic implementation using a null-terminated array for `String`. Figure 13c is an optimized implementation using a chunked linked list for `String`.

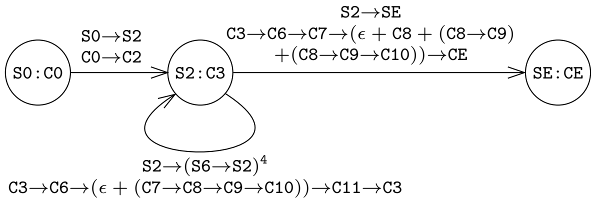
`u8[]`). `lnode(T)` represents a linked list node type containing a value of type `T`. Similarly, `clnode(T)` and `tnode(T)` represent a chunked linked list and a tree node with values of type `T` respectively.



(a) Product CFG for programs figs. 13a and 13b

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

(b) Invariants Table for fig. 14a



(c) Product CFG for programs figs. 13a and 13c

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

(d) Invariants Table fig. 14c

Figure 14: Product CFGs and Invariants Tables showing bisimulation between Strlen specification in fig. 13a and two C implementations in figs. 13b and 13c

Table 4: List lifting constructors and their definitions.

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

5.1.2 List

We wrote a Spec program specification that creates a list, a program that traverses a list to compute the sum of its elements and a program that computes the dot product of two lists. We use three different data layouts for a list in C: array ($\texttt{Clist}_{\mathbb{m}}^{\texttt{u32}[]}$), linked list ($\texttt{Clist}_{\mathbb{m}}^{\texttt{lnode(u32)}}$), and a chunked linked list ($\texttt{Clist}_{\mathbb{m}}^{\texttt{clnode(u32)}}$). The lifting constructors are shown in table 4. Although similar to the String lifting constructors, these lifting constructors differ widely in their data encoding. For example, $\texttt{Clist}_{\mathbb{m}}^{\texttt{u32}[]}(p, i, n)$ 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. ($\texttt{Clist}_{\mathbb{m}}^{\texttt{lnode(u32)}}$) and ($\texttt{Clist}_{\mathbb{m}}^{\texttt{clnode(u32)}}$), 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 5: Tree lifting constructors and their definitions.

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

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 ($\texttt{Ctree}_{\mathbb{m}}^{\texttt{u32[]}}$), and (2) a linked tree node with two pointers for the left and right children ($\texttt{Ctree}_{\mathbb{m}}^{\texttt{tnode(u32)}}$) (shown in table 5). 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 6: Matrix and auxiliary List lifting constructors and their definitions.

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

5.1.4 Matrix

We wrote a Spec program to count the frequency of a value appearing in a 2D matrix. A matrix is represented as an ADT that resembles a `List` of `Lists` ((T4) in table 6). The C implementations for a `Matrix` object include (a) a two-dimensional array ($\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{u32[][]}}$), (b) a flattened row-major array ($\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{u32[r]}}$), (c) a flattened column-major array ($\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{u32[c]}}$), (d) a linked list of 1D arrays ($\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{lnode(u32[])}}$), (e) a 1D array of linked lists ($\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{lnode(u32)[]}}$) and (f) a 1D array of chunked linked list ($\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{clnode(u32)[]}}$) 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, $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{u32[r]}}$ and $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{u32[c]}}$ for lifting each row of matrices lifted using the corresponding $\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{u32[r]}}$ and $\mathtt{Cmat}_{\mathbb{m}}^{\mathtt{u32[c]}}$ `Matrix` lifting constructors. These constructors are listed in table 6.

Table 7: 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)				

5.2 Results

Table 7 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 $\mathtt{Cstr}^{\mathtt{T}}_{\mathbb{m}}(p : \mathtt{i32}, \dots)$ (as defined in table 4). 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 2.1. 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, 35, 33, 34] 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. 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.

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/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`.

[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] **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`.

[30] **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.

[31] **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`.

[32] **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.

[33] **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.

[34] **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.

[35] **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.

[36] **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.

[37] **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.

[38] **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.

[39] **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`.

[40] **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`.

[41] **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/`.

[42] **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`.

[43] **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`.

[44] **Zuck, L.**, **A. Pnueli**, **Y. Fang**, and **B. Goldberg** (2003). Voc: A methodology for the translation validation of optimizing compilers. **9**(3), 223–247.

[45] **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`.