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 many 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 [28]. Frameworks for program equivalence proofs have been developed in interactive theorem provers like Coq [18] where correlations and invariants are manually identified during proof codification. On the other hand, programming languages like Dafny [30] 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 [35, 45, 42, 44, 29, 47, 48, 39, 46, 31, 27, 32, 12, 41, 17, 24, 40, 34] across low level programming languages such as C and assembly[1]. 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 equivalence of a program pair, 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 and its optimized C implementations. We will demonstrate how S2C is capable of proving equivalence of multiple equivalent C implementations with vastly different (a) data layouts (e.g. array, linked list representations for a *list*) and (b) algorithmic strategies (e.g. alternate algorithms, optimizations) against a *single* functional specification.

[1] TODO:llvm ir also?

This opens the possibility of regression verification [43, 22], where S2C can be used to automate verification across software updates that change memory layouts for data structures.

1.1 A Motivating Example

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) [14] ensure several safety properties. We define a minimal functional language, called Spec, that enables the safe and succinct specification of programs manipulating and traversing recursive data structures. Spec is equipped with ADTs as well as boolean (`bool`) and fixed-size bitvector (`i<N>`) types.

We motivate our approach by considering example Spec and C programs. The major hurdles of our approach are listed alongside an informal discussion of our proposed solutions. We state our primary contributions in section 1.2 and finish with an outline of the rest of the thesis in section 1.3.

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

(a) Spec Program

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

(b) C Program with `malloc()`

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

Figures 1a and 1b show the construction of lists in Spec and C respectively. The `List` ADT in the Spec program is defined at line `A0` in fig. 1a. An empty `List` is represented by the *data constructor* `LNil`, whereas a non-empty list uses the `LCons` constructor to combine its first value (`val:i32`) and the remaining list (`tail:List`). The inputs to a Spec procedure are its well-typed arguments, which may include recursive data structure (i.e. ADT) values. The inputs to a C procedure are its explicit arguments and the implicit state of program memory at procedure entry. Similarly, the output of a C procedure consists of its explicit return value and the state of program memory at procedure exit.

The Spec procedure `mk_list` (defined at line `A6` in fig. 1a), takes a bitvector of size `32` (`n:i32`). It returns a `List` value representing the list $[(\texttt{n}-1), (\texttt{n}-2), \ldots, 1, 0]$. On the other hand, the C procedure `mk_list` (defined at line `B4` in Figure 1b) constructs a *pointer based* linked list representing the list identical to the Spec procedure. Unlike Spec, the construction of the linked list in C requires explicit allocation of memory through calls to `malloc` in addition to stores to the memory. We are interested in showing that the Spec and C `mk_list`

procedures are 'equivalent' i.e., given equal `n` inputs, they both construct lists that are 'equal'.

```
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:     mm := mm[addrof(p -mm->_lnode val)←i]_i32;
C6:     mm := mm[addrof(p -mm->_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.

For ease of comparison, we first convert both `mk_list` procedures to a common logical encoding, and call this the intermediate representation (IR for short). Figures 2a and 2b show the intermediate representations of the Spec and C `mk_list` procedures in figs. 1a and 1b respectively. For the Spec procedure, the tail-recursive function `mk_list_impl` is converted to a loop and inlined in the top-level function `mk_list` in the IR. For the C procedure in fig. 1b, the memory state is made explicit (represented by `mm`), and the size and memory layout of each type is concretized in the IR. For example, the `unsigned` and pointer types are encoded as the `i32` bitvector type.

Hence, we are interested in showing equivalence of the Spec and C IRs. Since the argument `n` to both procedures have identical types (i.e. `i32`), their equality is trivially expressible as: $\texttt{n}_S = \texttt{n}_C$ [2]. The Spec procedure uses the ADT `List` to represent a list. However, the C procedure represents its list using a collection of `lnode` objects linked through their `next` fields, and simply returns a value of type `i32` (`lnode*` in the original C program) pointing to the first `lnode` in the list (or the null value in case of an empty list). In order to express equality between these two values (of types `List` and `i32`) representing lists, we would like to 'adapt' one of the values so as to match their types. We choose to lift the C linked list (represented by the `i32` value and the C memory state) to a `List` value using an

[2]We use S and C subscripts to refer to variables in the Spec and C procedures respectively.

operator called a *lifting constructor*. Let us call this lifting constructor $\texttt{Clist}^{\texttt{lnode}}$ and the expression $\texttt{Clist}^{\texttt{lnode}}_{\texttt{mm}}(p\texttt{:i32})$ represents a `List` list constructed from a C pointer p (pointing to a `lnode` object) in the memory state $\texttt{mm}$. We will formally define $\texttt{Clist}^{\texttt{lnode}}$ in section 2.5. For now, this allows us to express equality between the outputs of the Spec and C procedures as $\texttt{ret}_S = \texttt{Clist}^{\texttt{lnode}}_{\texttt{mm}}(\texttt{ret}_C)$, where $\texttt{ret}_S$ and $\texttt{ret}_C$ represents the values returned by the respective Spec and C procedures in figs. 2a and 2b. To further emphasize the fact that we are comparing (a) a Spec ADT value with (b) an ADT value lifted from C values using a lifting constructor, we use '$\sim$' instead of '=' and call it a recursive relation: $\texttt{ret}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{mm}}(\texttt{ret}_C)$.

(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

Consequently, we are interested in proving that given $\texttt{n}_S = \texttt{n}_C$ at the procedure entries, $\texttt{ret}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{mm}}(\texttt{ret}_C)$ holds at the exits of both procedures. Before going into the proof method, we first introduce an alternate representation of IR, called the Control-Flow Graph (CFG for short). Figures 3a and 3b show the CFG representation of the Spec and C IRs in figs. 2a and 2b respectively. Unlike the linear IR, CFG gives a graphical view of the control flow structures. In essence, each node represents a PC location of its IR, and each edge represents (possibly conditional) transition between PCs through instruction execution. For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 3b, the edge C5→C3 represents the path C5→C6→C7→C8→C3.

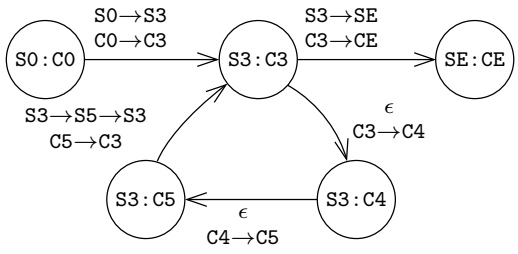


Figure 4: Product-CFG between the CFGs in figs. 3a and 3b

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

Due to the similarity of control flow (and loops) in the two procedures, we choose *bisimulation* as our proof method. Intuitively, a bisimulation relation encodes the execution of both procedures in lockstep which ensures equal output lists. Bisimulation can be represented as a *product program* [46] and its CFG representation is called a *product*-CFG. Figure 4 shows a product-CFG between the Spec and C procedures in figs. 3a and 3b respectively.

At each node of the product-CFG, *invariants* relate the states of the Spec and C procedures respectively. Table 1 lists invariants for the product-CFG in fig. 4. At the start node (S0:C0) of the product-CFG, the precondition (labeled (P)) ensures equality of input arguments $\mathtt{n}_S$ and $\mathtt{n}_C$ at the procedure entries. Inductive invariants (labeled (I)) need to be inferred at each intermediate product-CFG node (e.g., (S3:C3)) relating both programs' states. For example, at node (S3:C5), (I6) $\mathtt{i}_S = \mathtt{i}_C$ is an inductive invariant. The inductive invariant (I4) $\mathtt{l}_S \sim \mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$ is another example of a recursive relation and asserts equality between the intermediate Spec and C lists at the loop heads. Assuming that the precondition ((P)) holds at the entry node (S0:C0), a bisimulation check involves checking that the inductive invariants hold too, and consequently the postcondition ((E)) holds at the exit node (SE:CE). Checking correctness of a bisimulation relation involves checking whether an invariant holds (along with many other things). These checks result in proof queries which must be discharged by a theorem prover (also known as a solver).

1.2 Our Contributions

As previously summarized in section 1.1, an algorithm to find a bisimulation based proof of equivalence between a Spec and C procedure involves three major algorithms: (A1) An algorithm for construction of a product-CFG by correlating pro-

gram executions across the Spec and C programs respectively. (A2) An algorithm for identification of inductive invariants at intermediate correlated PCs. (A3) An algorithm for solving proof obligations generated by (A1) and (A2) algorithms. Next we list our major contributions.

- Proof Discharge Algorithm: Solving proof obligations ((A3)) involving recursive relations (generated by (A1) and (A2)) is quite 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 queries. These counterexamples are the backbone of counterexample-guided heuristics for (A1) and (A2) algorithms as we will see soon. As part of our proof discharge algorithm, we reformulate equality of ADT values (i.e. recursive relations) as equivalence programs and discharge these proof queries using a nested (albeit much simpler) equivalence check.

- Spec-to-C Automatic Equivalence Checker Tool: Our second contribution is S2C, a *sound* equivalence checker tool capable of proving equivalence between a Spec and a C program automatically. S2C either successfully finds a bisimulation relation implying equivalence or it provides a (sound but incomplete) unknown verdict. S2C is based on the Counter tool[24] and uses specialized versions of (a) counterexample-guided correlation algorithm for incremental construction of a product-CFG ((A1)) and (b) counterexample-guided invariant inference algorithm for inference of inductive invariants at correlated PCs in the (partially constructed) product-CFG ((A2)). S2C discharges required verification conditions (i.e. proof obligations) using our Proof Discharge Algorithm. The counterexamples generated by the proof discharge algorithm help steer the search algorithms (A1) and (A2). Figure 5 gives an overview of S2C and its interacting components.

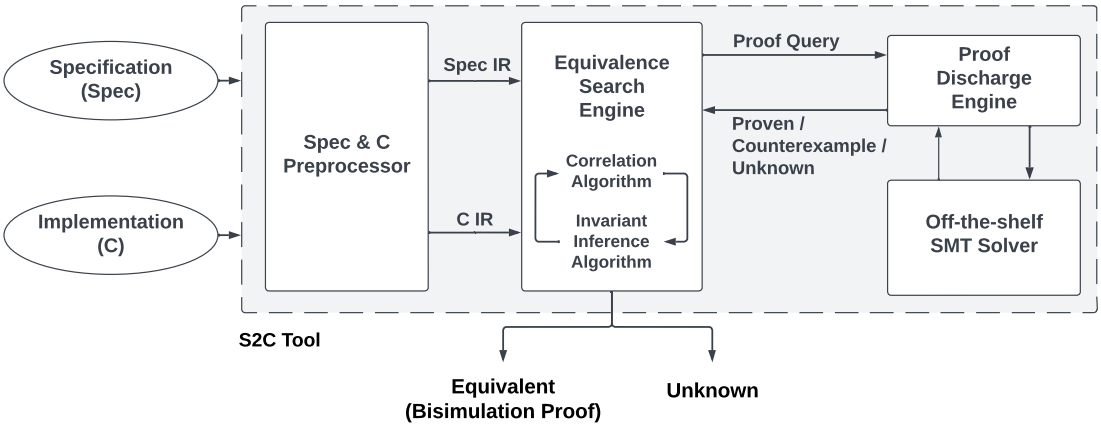


Figure 5: Diagram of Spec-to-C Automatic Equivalence Checker: S2C. The inputs to S2C are the Spec and C programs. S2C either successfully finds a bisimulation proof implying equivalence or soundly returns an unknown verdict.

1.3 Outline of the Thesis

TODO: needs updation after all content is fixed **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.

2 Languages and Equivalence

This section introduces the Spec language and give a detailed description of Spec along with the intermediate representations introduced in section 1.1. Next, we give a formal definition of equivalence in our context, followed by a discussion on bisimulation. We finish with an analysis of proof obligations generated during the search for a bisimulation relation.

2.1 The Spec Language

We start with an introduction to the Spec language. Spec supports recursive algebraic data types (ADT) [3] similar to the ones available in most functional languages (e.g. Haskell[4]). Spec does not support parametric types but does allow

[3] TODO:cite pls
[4] TODO:cite pls

ADTs which are mutually recursive. Additionally, Spec is equipped with the following *scalar* types: `unit`, `bool` (boolean) and `i<N>` (bitvector of size `N`). ADTs can be thought of as 'sum of product' types where each *data constructor* represents a variant and the arguments to each data constructor represents its *fields*. For example, the `List` type (defined at `A0` in fig. 1a) has two variants `LNil` and `LCons`. `LNil` has no fields while `LCons` has two fields `val` and `tail` of types `i32` and `List` respectively. Additionally, Spec follows *equirecursive* typing rules [5] i.e. a `List` value l and $\texttt{LCons}(1_{\texttt{i32}}, l)$ have *equal* types. Later in section 4.4.6, we give a more formal definition of ADTs with their graphical representation. The language also borrows its expression grammar heavily from functional languages. This includes the constructs: `let-in`, `if-then-else`, `match` and function application expressions. Pattern matching (i.e. deconstruction) of ADT values is archieved through `match`. 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 intrinsic scalar 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 unsigned or signed integers (e.g., $\leq_{\texttt{u,s}}$). The equality operator ($=$) is only supported for scalar types.

[5] TODO:cite pls

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

Figure 6: Simplified expression grammar of Spec language

Figure 6 shows the simplified expression grammar for Spec language. ⟨data-cons⟩ represents a ADT data constructor. The '⟨expr⟩ `is` ⟨data-cons⟩' construct returns a `bool` and is used to test whether the top-level constructor of the ADT value ⟨expr⟩ is ⟨data-cons⟩. ⟨scalar-op⟩ includes the logical, arithmatic and relational operators supported by Spec.

2.2 Intermediate Representations

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

We have already seen the the IRs (in figs. 2a and 2b) for the Spec and C programs that construct lists in figs. 1a and 1b. Figures 7a and 7b show Spec and C programs that traverse a list and return the sum of all the values in it. The corresponding IR programs are shown in figs. 8a and 8b.

During conversion of a Spec source to its IR, (a) `match` statements are lowered to explicit `if-else` conditionals where each branch is associated with a `match`

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

(a) Spec Program

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

(b) C Program

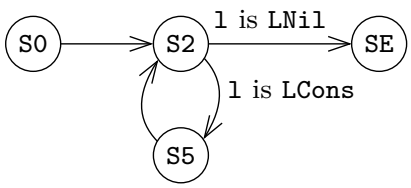
Figure 7: Spec and C Programs traversing a Linked List.

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

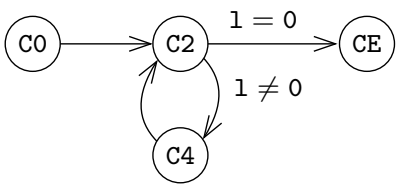
(a) (Abstracted) Spec IR

```
C0: i32 sum_list (i32 l) {
C1:   i32 sum ≔ 0i32;
C2:   while l ≠ 0i32:
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.

branch, (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. For example, during conversion of Spec program in fig. 7a, (a) the `match` statement in `A3` is converted to `if-else` (b) the tail recursive function `sum_list_impl` is converted to a loop, and (c) the helper function `sum_list_impl` is inlined, to obtain the IR in fig. 8a.

Similarly, the following is performed during conversion of a C source to its IR: (a) the sizes and memory layouts of both scalar (e.g., `int`) and compound (e.g., `struct`) types are concretized, (b) the program memory state along with loads and stores on the memory are made explicit and (c) we annotate `malloc` calls with their call-site i.e. IR PC. For example, during conversion of C program in fig. 1b to IR (in fig. 2b), (a) the size of pointer and `unsigned` types are fixed to 32-bits (i.e. `i32`), (b) `mm` is used to represent the program memory with explicit writes at `C5` and `C6`, and (c) $\texttt{malloc}_{\texttt{C4}}$ is annotated with its call-site `C4`.

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., $l.\texttt{val}$ and $l.\texttt{next}$ represents the first and second fields of the `LCons` constructor in fig. 8a. The IR also allows querying the top-level data constructor of an ADT value using the *sum-is* operator, e.g., l is `LNil` in fig. 8a. The `val` field is associated with the `LCons` data constructor and evidently, $l.\texttt{val}$ is only *well-formed* if l is `LCons`. Importantly, the construction of the Spec IR ensures the well-formedness of all expressions. Using *accessor* and *sum-is* operators, a `List` value l can be expanded as:

$$U_S : l = \underline{\texttt{if}}\ l\ \text{is}\ \texttt{LNil}\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(l.\texttt{val}, l.\texttt{next}) \tag{1}$$

In this expanded representation of l, the *sum-deconstruction* operator '$\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$' conditionally deconstructs the sum type into its variants `LNil` and `LCons`. The *underlined* $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ operator is a stricter version of `if-then-else`, and is only used for ADT values. An $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ expression e (for an ADT type T) must satisfy the following properties: (a) e has exactly one branch for each data construction of T (in the order they are defined), and (b) the branch associated with the data constructor V has the form $V(e_1, e_2, \dots)$ i.e. its top-

level operator is V. For example, an `if-then-else` expression for the `List` type must be of the form: '`if` e_1 `then LNil else LCons`(e_2, e_3)' for some expressions e_1, e_2, e_3. Equation (1) is called the *unrolling procedure* for the `List` variable l. We can similarly define the unrolling procedure for any ADT variable (based on the definition of the ADT).

The C memory is modeled as a byte-addressable array $\mathtt{m}$ in the IR and pointers are converted to bitvectors. "$\mathtt{m}[p]_{\mathtt{T}}$" represents a memory load operation and is equal to the bytes at addresses [p, p+`sizeof(T)`) in $\mathtt{m}$, interpreted as a value of type '`T`'. Similarly, "$\mathtt{m}[p \leftarrow v]_{\mathtt{T}}$" represents a memory store operation and is equal to $\mathtt{m}$ everywhere except at addresses [p, p+`sizeof(T)`) which contains the value v of type '`T`' (e.g., `C5` in fig. 2b). We use the following two C-like syntaxes to represent more complex memory loads succintly:

1. "$p \xrightarrow{\mathtt{m}}_{\mathtt{T}} \mathtt{f}$" is equivalent to "$\mathtt{m}[p + \mathtt{offsetof}(\mathtt{T}, \mathtt{f})]_{\mathtt{typeof(T.f)}}$" i.e., it returns the bytes in the memory array $\mathtt{m}$ starting at address '$p + \mathtt{offsetof}(\mathtt{T}, \mathtt{f})$' and interpreted as a value of type '`typeof(T.f)`'.

2. "$p[i]^{\mathtt{T}}_{\mathtt{m}}$" is equivalent to "$\mathtt{m}[p + i \times \mathtt{sizeof}(\mathtt{T})]_{\mathtt{T}}$" i.e., it returns the bytes in the memory array $\mathtt{m}$ starting at address '$p + i \times \mathtt{sizeof}(\mathtt{T})$' and interpreted as a value of type '`T`'. Interestingly, $\mathtt{m}[p]_{\mathtt{T}} = p[0]_{\mathtt{m}}$.

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 known constants. We use the '`addrof()`' operator to extract the address of a memory load expression: "`addrof(`$\mathtt{m}[p]_{\mathtt{T}}$`)`" is equivalent to p. For example, at PC `C5` in fig. 2b, $\mathtt{addrof}(p \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val}) \Leftrightarrow p + \mathtt{offsetof}(\mathtt{lnode}, \mathtt{val})$.

Figures 8c and 8d show the Control-Flow Graph (CFG) representation of the Spec and C IRs in figs. 8a and 8b respectively. Each CFG node represents a program point (i.e. IR PC) and edges represent transitions through execution of instructions. Each edge is associated with: (a) an *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. We omit these transfer functions in the CFG figures and only show the edge conditions (unless they are *true*). Henceforth, We refer to the IR programs as Spec and C directly unless a distinction is necessary.

2.3 Equivalence Definition

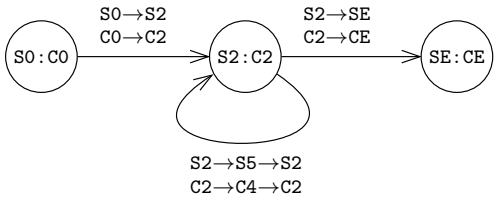
Given (1) a Spec function specification $\mathcal{S}$, (2) a C implementation $\mathcal{C}$, (3) a precondition Pre that relates the initial inputs $\texttt{Input}_S$ and $\texttt{Input}_C$ to $\mathcal{S}$ and $\mathcal{C}$ respectively, and (4) a postcondition $Post$ that relates the final outputs $\texttt{Output}_S$ and $\texttt{Output}_C$ of $\mathcal{S}$ and $\mathcal{C}$ respectively[6]: $\mathcal{S}$ and $\mathcal{C}$ are *equivalent* if for all possible inputs $\texttt{Input}_S$ and $\texttt{Input}_C$ such that $Pre(\texttt{Input}_S, \texttt{Input}_C)$ holds, $\mathcal{S}$'s execution is well-defined on $\texttt{Input}_S$, *and* $\mathcal{C}$'s memory allocation requests during its execution on $\texttt{Input}_C$ are successful, then both programs $\mathcal{S}$ and $\mathcal{C}$ produce outputs such that $Post(\texttt{Output}_S, \texttt{Output}_C)$ holds.

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

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

The returned values of $\mathcal{S}$ and $\mathcal{C}$ form their observable outputs. For $\mathcal{S}$, the returned values are explicit and may include ADT values. For $\mathcal{C}$, observables include the returned value alongside the implicit memory state at program exit. The postcondition $Post$ relates these outputs of the two programs. The pair $(Pre, Post)$ represents the input-output behaviour of $\mathcal{C}$ in terms of the specification $\mathcal{S}$, and is called the *input-output specification*. In general, Spec and C sources may contain multiple top-level procedures, with calls to each other. In this case, we

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



(a) Product-CFG

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

(b) Node Invariants of the Product-CFG

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

are interested in finding equivalence between each pair of $\mathcal{S}$ and $\mathcal{C}$ procedures with respect to their input-output specification.

Sometimes, the user may be interested in constraining the nature of inputs to $\mathcal{C}$ for the purpose of checking equivalence only for *well-defined* inputs. In those circumstances, we use a combination of Pre and ($\mathcal{S}$ `def`) to constrain the execution of $\mathcal{C}$ to inputs for which we are interested in proving equivalence. For example, the C library function `strlen(char* str`$_C$`)` is well-defined only if `str`$_C$ represents a valid null character terminated string. This includes the assumption that the pointer `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:

$$\mathtt{String} = \mathtt{SInvalid} \mid \mathtt{SNil} \mid \mathtt{SCons}(\mathtt{i8}, \mathtt{String})$$

($\mathcal{S}$ `def`) asserts $\neg$(`str`$_S$ is `SInvalid`) (using `assuming-do`) and the precondition Pre contains the relation (`str`$_S$ is `SInvalid`) $\Leftrightarrow$ (`str`$_C = 0$). Hence, ($\mathcal{S}$ `def`) and Pre ensure that we compute equivalence for those executions of $\mathcal{S}$ and $\mathcal{C}$ where the input strings are well-defined. A similar strategy is employed for other functions as explored later in section 5.2.

2.4 Bisimulation Relation

Recall that, we construct a *bisimulation relation* to identify equivalence between Spec and C procedures. A bisimulation relation correlates the transitions of $\mathcal{S}$ and $\mathcal{C}$ in lockstep, such that the lockstep execution ensures identical observable behaviour. A bisimulation relation between two programs can be represented

using a *product program* [46] 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 $\mathcal{S}$ and $\mathcal{C}$, e.g., (`S2:C2`) is formed by pairing `S2` and `C2`. If the lockstep execution of both programs is at node (`S2:C2`) in the product-CFG, then $\mathcal{S}$'s execution is at `S2` and $\mathcal{C}$'s execution is at `C2`. The start node (`S0:C0`) of the product-CFG correlates the start nodes of CFGs of $\mathcal{S}$ and $\mathcal{C}$. Similarly, 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 $\mathcal{S}$ with a path in $\mathcal{C}$. A product-CFG edge encodes the lockstep execution of its correlated paths. For example, the product-CFG edge (`S2:C2`)$\rightarrow$(`S2:C2`) is formed by pairing `S2`$\rightarrow$`S5`$\rightarrow$`S2` and `C2`$\rightarrow$`C4`$\rightarrow$`C2` in figs. 8c and 8d respectively, and represents that when $\mathcal{S}$ makes the transition `S2`$\rightarrow$`S5`$\rightarrow$`S2`, $\mathcal{C}$ makes the transition `C2`$\rightarrow$`C4`$\rightarrow$`C2` in lockstep. In general, a product-CFG edge e may correlate a finite path ρ_S in $\mathcal{S}$ with a finite path ρ_C in $\mathcal{C}$, written $e = (\rho_S, \rho_C)$. The empty path ϵ in $\mathcal{S}$ may be correlated with a finite path in $\mathcal{C}$. However, a product-CFG is only well-formed (i.e. represents a valid bisimulation relation) if no loop path in $\mathcal{C}$ is correlated with ϵ in $\mathcal{S}$. For example, fig. 4 shows the product-CFG between the programs in figs. 3a and 3b respectively. The edges (`S3:C3`)$\rightarrow$(`S3:C4`) and (`S3:C4`)$\rightarrow$(`S3:C5`) correlate the empty path ϵ with the non-empty paths `C3`$\rightarrow$`C4` and `C4`$\rightarrow$`C5` respectively. However, the only loop path `C3`$\rightarrow$`C4`$\rightarrow$`C5`$\rightarrow$`C3` in $\mathcal{C}$ is still correlated with the non-empty path `S3`$\rightarrow$`S5`$\rightarrow$`S3` in $\mathcal{S}$ and thus, the product-CFG in fig. 4 satisfies this well-formedness criterion.

At the start node (`S0:C0`) of the product-CFG in fig. 9a, the precondition *Pre* (labeled Ⓟ) ensures equality of input lists $\texttt{l}_S$ and $\texttt{l}_C$ at procedure entries. *Inductive invariants* (labeled Ⓘ) are inferred at each intermediate product-CFG node (e.g., (`S2:C2`)) that relate the values of $\mathcal{S}$ with values and memory state of $\mathcal{C}$. At the exit node (`SE:CE`) of the product-CFG, the postcondition *Post* (labeled Ⓟ) represents equality of observable outputs and forms our overall proof obligation. Assuming that the precondition *Pre* (Ⓟ) holds at the entry node (`S0:C0`), a bisimulation check involves checking that the inductive invariants (Ⓘ) hold too, and consequently the postcondition *Post* (Ⓔ) holds at the exit node (`SE:CE`).

The input-output specification (i.e. $(Pre, Post)$) is manually provided by the user while all inductive invariants are identified by an invariant inference algorithm described in section 4.3.

2.5 Recursive Relation

In section 1.1, we briefly introduced a lifting constructor ($\texttt{Clist}^{\texttt{lnode}}$) and recursive relations. In fig. 9b, the precondition ($\textcircled{\texttt{P}}$) is another instance of a recursive relation: "$\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$" where $\texttt{l}_S$ and $\texttt{l}_C$ represent the input arguments to the Spec and C procedures respectively, `lnode` is the C `struct` type that contains the `val` and `next` fields (defined at `B0` in fig. 7b), and $\texttt{m}$ is the byte-addressable array representing the current memory state of the C program. $l_1 \sim l_2$ is read *l_1 is recursively equal to l_2* and is semantically equivalent to $l_1 = l_2$. The '$\sim$' simply emphasizes that l_1 and l_2 are (possibly recursive) ADT values. The lifting constructor $\texttt{Clist}^{\texttt{lnode}}$ 'lifts' a C pointer value p (pointing to an object of type `struct lnode`) and memory state $\texttt{m}$ to a (possibly infinite in case of a circular list) `List` value, and is defined through its *unrolling procedure* as follows:

$$\begin{aligned} U_C : \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(p{:}\texttt{i32}) = \underline{\texttt{if}}\ p = 0\ \underline{\texttt{then}}\ \texttt{LNil} \\ \underline{\texttt{else}}\ \texttt{LCons}(p \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{val}, \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(p \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{next})) \end{aligned} \tag{2}$$

Note the recursive nature of the lifting constructor $\texttt{Clist}^{\texttt{lnode}}$: 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 $p \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{val}$ in memory $\texttt{m}$ and the list formed by recursively lifting $p \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{next}$ through $\texttt{Clist}^{\texttt{lnode}}$. $\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(p)$ allows us to adapt a C linked list (formed by chasing pointers in the memory $\texttt{m}$) to a `List` value and compare it with a Spec `List` value for equality.

2.6 Proof Obligations

As previously discussed, algorithms for (a) incremental construction of a Product-CFG and (b) inference of invariants at intermediate PCs in the (partially constructed) product-CFG, are based on prior work[24] and discussed subsequently

in sections 4.2 and 4.3. For now, we discuss the proof obligations that arise from a given product-CFG. Recall that a bisimulation check involves checking that all inductive invariants (and the postcondition *Post*) hold at their associated product-CFG nodes.

We use relational Hoare triples to express these proof obligations [13, 25]. If ϕ denotes a predicate relating the machine states of $\mathcal{S}$ and $\mathcal{C}$, then for a product-CFG edge $e = (\rho_S, \rho_C)$, $\{\phi_s\}(e)\{\phi_d\}$ denotes the condition: if any machine states σ_S and σ_C of programs $\mathcal{S}$ and $\mathcal{C}$ are related through precondition $\phi_s(\sigma_S, \sigma_C)$ and the finite paths ρ_S and ρ_C are executed in $\mathcal{S}$ and $\mathcal{C}$ respectively, then execution terminates normally in states $\sigma_S^{'}$ (for $\mathcal{S}$) and $\sigma_C^{'}$ (for $\mathcal{C}$) and postcondition $\phi_d(\sigma_S^{'}, \sigma_C^{'})$ holds.

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

$$(\phi_s \wedge \texttt{pathcond}_{\rho_S} \wedge \texttt{pathcond}_{\rho_C} \wedge \texttt{ubfree}_{\rho_S}) \Rightarrow \texttt{WP}_{\rho_S,\rho_C}(\phi_d) \tag{3}$$

Here, $\texttt{pathcond}_{\rho_X}$ represents the condition that path ρ is taken in program X and $\texttt{ubfree}_{\rho_S}$ represents the condition that execution of $\mathcal{S}$ along path ρ_S is free of undefined behaviour. $\texttt{WP}_{\rho_S,\rho_C}(\phi_d)$ represents the weakest-precondition of the predicate ϕ_d across the product-CFG edge $e = (\rho_S, \rho_C)$. From now on, we will use '`LHS`' and '`RHS`' to refer to the antecedent and consequent of the implication operator '$\Rightarrow$' in eq. (3).

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

$$\begin{aligned} \texttt{l}_S \sim \texttt{Clist}_{\texttt{m}}^{\texttt{lnode}}(\texttt{l}_C) \wedge \texttt{sum}_S = \texttt{sum}_C \wedge (\texttt{l}_S \text{ is } \texttt{LCons}) \wedge (\texttt{l}_C \neq 0) \\ \Rightarrow \texttt{l}_S.\texttt{next} \sim \texttt{Clist}_{\texttt{m}}^{\texttt{lnode}}(\texttt{l}_C \xrightarrow{\texttt{m}}_{\texttt{lnode}} \texttt{next}) \end{aligned} \tag{4}$$

Due to the presence of recursive relations, these proof queries (e.g., eq. (4)) 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 section demonstrates our proof discharge algorithm through examples. We consider proof obligations generated due to invariants shown in table 1 and fig. 9b for the product-CFGs in figs. 4 and 9a respectively. We start by describing the properties of the proof discharge algorthm. We also list the properties of the proof obligations generated by our equivalence checker; these properties are essential for the correctness of our proof discharge algorithm. Next, the proof discharge algorithm is explored using sample proof obligations, and we finish with a pseudo-code of the algorithm.

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 $\phi_s = (\phi^{\mathtt{1}}_{\mathtt{s}} \wedge \phi^{\mathtt{2}}_{\mathtt{s}} \wedge ... \wedge \phi^{\mathtt{k}}_{\mathtt{s}})$, at any node s of a product-CFG, if a recursive relation appears in ϕ_s, it must be one of ϕ^1_s, ϕ^2_s, ..., or ϕ^k_s. 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. (3)) to a first-order logic formula of the following form:

$$(\eta^l_1 \wedge \eta^l_2 \wedge ... \wedge \eta^l_m) \Rightarrow (\eta^r_1 \wedge \eta^r_2 \wedge ... \wedge \eta^r_n) \tag{5}$$

Thus, due to the conjunctive recursive relation property of ϕ_s and ϕ_d, any recursive relation in eq. (5) must appear as one of η^l_i or η^r_j. To simplify proof obligation discharge, we break a first-order logic proof obligation P of the form in eq. (5) into multiple smaller proof obligations of the form $P_j : (\mathtt{LHS} \Rightarrow \eta^{\mathtt{r}}_{\mathtt{j}})$, for $j = 1..n$. Each proof obligation P_j is then discharged separately. We call this conversion from a bigger query to multiple smaller queries, `RHS`-*breaking*.

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

3.2 Iterative Unification and Rewriting Procedure

We begin with some definitions. An expression e whose top-level constructor is a lifting constructor, e.g., $e = \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$, 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*. It is possible to convert any expression e into its canonical form $\hat{e}$. For example, the canonical form of $a + \mathtt{LCons}(b, l).\mathtt{tail.val}$ is given by $a + l.\mathtt{val}$, where $l.\mathtt{val}$ is a pseudo-variable.

Consider the expression tree of a canonical expression $\hat{e}$. The internal nodes of $\hat{e}$ represents ADT data constructors and the `if`-`then`-`else` sum-deconstruction operator. The leaves of $\hat{e}$ (also called *atoms* of $\hat{e}$) 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 $\hat{e}$'s tree is the path from the root of $\hat{e}$ to the node v. The *expression path condition* represents the conjunction of all the `if` conditions (if the `then` branch of taken along the path), or their negation (if the `else` branch is taken along the path) for each `if`-`then`-`else` along the path. 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 their tree structures created by data constructors and the `if`-`then`-`else` operator. The unification procedure either fails to unify, or it returns tuples $\langle p_1, a_1, p_2, e_2 \rangle$ where atom a_1 at expression path condition p_1 in one expression is correlated with expression e_2 at expression path condition p_2 in the other expression.

For two non-atomic expressions, e_1 and e_2 to unify successfully, it must be true that either the top-level operator in e_1 and e_2 is the same data constructor (in which case an unification is attempted for each of their children), *or* the top-level operator in atleast one of e_1 or e_2 is `if`-`then`-`else`.

If the top-level operator in *exactly one* of e_1 and e_2 (say e_2) is `if`-`then`-`else`, then e_1 must have a data constructor at its root. Given $e_2 =$ `if` c `then` $e_2^{\mathtt{th}}$ `else` $e_2^{\mathtt{el}}$, we first attempt to unify e_1 with the if branch $e_2^{\mathtt{th}}$ — if unification succeeds, we also unify c (then condition) with *true*. Otherwise, we unify e_1 with the else branch $e_2^{\mathtt{el}}$ and $\neg c$ (else condition) with *true*.

If the top-level operator in both e_1 and e_2 is `if`-`then`-`else`, we unify each child

(condition and branch expressions) of the corresponding `if`-`then`-`else` operators. Recall that the `if`-`then`-`else` operator (introduced in section 2.2) for an ADT T must have exactly one branch for each data constructor of T, and the branch associated with the data constructor V has V in its top-level. Whenever we descend down an `if`-`then`-`else` operator, we conjunct the `if` condition (if `then` branch is taken) or its negation (if `else` branch is taken) with its associated expression path condition. This allows us to keep track of the expression path conditions for both expressions during recursive descent to their children.

If one of e_1 and e_2 (say e_2) is atomic, unification always succeeds and returns $\langle p_2, e_2, p_1, e_1 \rangle$. With each atom of an ADT type, we associate an *unrolling procedure*. By definition, an ADT atom is either a pseudo-variable or a lifted expression. Each (pseudo-)variable is associated with its unrolling procedure governed by its type. For example, the unrolling procedure for a `List` variable l is given by U_S (eq. (1)). For lifted expressions, the unrolling procedure is given by its definition, e.g., U_C (eq. (2)) for the lifting constructor $\texttt{Clist}^{\texttt{lnode}}$.

Given two *canonical* expressions e_a and e_b at expression path conditions p_a and p_b respectively, an *iterative unification and rewriting procedure* $\Theta(p_a, e_a, p_b, e_b)$ is used to identify a set of correlation tuples 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 $\langle p_1, a_1, p_2, a_2 \rangle$ to the final output. Otherwise, if the unification correlates an atom a_1 at expression path condition p_1 to a non-atomic expression e_2 at expression path condition p_2, we *rewrite* a_1 using its unrolling procedure to obtain expression e_1. The unification algorithm then proceeds by unifying e_1 and e_2 through a recursive call to $\Theta(p_1, e_1, p_2, e_2)$. The maximum number of rewrites performed by $\Theta(p_a, e_a, p_b, e_b)$ (before termination) is bounded by the sum of number of ADT data constructors in e_a and e_b. The algorithms responsible for canonicalization, unification, and iterative unification and rewriting are further discussed in sections 4.4.1 to 4.4.3 respectively.

For a recursive relation $l_1 \sim l_2$, we unify (canonicalized) 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 $\langle p_1^i, a_1^i, p_2^i, a_2^i \rangle$ (for $i = 1 \ldots n$), then the *decomposition* of $l_1 \sim l_2$ is defined as:

$$l_1 \sim l_2 \Leftrightarrow \bigwedge_{i=1}^{n} (p_1^i \wedge p_2^i \rightarrow (a_1^i = a_2^i)) \tag{6}$$

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 $(\neg c_1, \neg c_2, l_1, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$. Consequently, 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 $(l_1\ \text{is}\ \texttt{LCons}) \wedge (l_1\ \text{is}\ \texttt{LCons} \rightarrow l_1.\texttt{val} = 42) \wedge (l_1\ \text{is}\ \texttt{LCons} \rightarrow l_1.\texttt{next} \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(l_2))$ [7]. 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 $(p_1^i \wedge p_2^i \rightarrow (a_1^i = a_2^i))$[8] in the decomposition is called a *decomposition clause*. A decomposition clause may relate only atomic values, i.e., it may relate (a) two scalars or (b) two ADT variable(s) and/or lifted expression(s). However, we restrict the shapes of recursive relation invariants such that each recursive relation in its decomposition *strictly* relates ADT values to lifted expressions. The invariant shapes along with the invariant inference procedure is discussed in section 4.3. We *decompose* a recursive relation by replacing it with its decomposition. We *decompose* a proof obligation by decomposing all recursive relations in it.

3.3 Categorization of Proof Obligations

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.

[7] $(l_1\ \text{is}\ \texttt{LCons})$ is equivalent to $\neg(l_1\ \text{is}\ \texttt{LNil})$. In general, for an ADT value v of type T (with data constructors $V_1, V_2, \ldots, V_k$), exactly one of $(v\ \text{is}\ V_i)$ is true.

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

For a decomposed proof obligation P_D : LHS $\Rightarrow$ 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 : LHS $\Rightarrow l \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(0)$, after elimination, yields P_E : LHS $\Rightarrow l$ is LNil. We categorize a proof obligation P : LHS $\Rightarrow$ 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 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 (S0:C0)$\rightarrow$(S3:C3) while checking if the (I4) invariant in table 1, $\texttt{l}_S \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(\texttt{l}_C)$ holds at (S3:C3): $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0}\rightarrow\texttt{S3}, \texttt{C0}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(\texttt{l}_C)\}$. 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 $\texttt{n}_S = \texttt{n}_C \Rightarrow \texttt{LNil} \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(0)$. Here, LNil is obtained for $\texttt{l}_S$ and 0 (null) is obtained for $\texttt{l}_C$. The one-unrolled form of this proof obligation yields $\texttt{n}_S = \texttt{n}_C \Rightarrow \mathit{true}$ which trivially resolves to true.

Consider the following example of a proof obligation: $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0}\rightarrow\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}, \texttt{C0}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(\texttt{l}_C)\}$. Notice, we

have changed the path in $\mathcal{S}$ (with CFG fig. 3a) to S0→S3→S5→S3 here. In this case, the corresponding first-order logic formula evaluates to: $(\mathtt{n}_S = \mathtt{n}_C) \wedge (0 <_u \mathtt{n}_S) \Rightarrow \mathtt{LCons}(0, \mathtt{LNil}) \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(0)$, where $(0 <_u \mathtt{n}_S)$ is the path condition for the path S0→S3→S5→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 $\{\mathtt{n}_S \mapsto 42, \mathtt{n}_C \mapsto 42\}$ evaluates this formula to false. These counterexamples assist in faster convergence of our correlation search and invariant inference procedures (as we will discuss later in sections 4.2 and 4.3).

Thus for type I queries, k-unrolling reduces all (if any) recursive relations in the original proof obligation into scalar equalities. The resulting query is further discharged using an SMT solver. Section 4.4 contains a deeper analysis of the following aspects of our proof discharge algorithm: (a) translation of formula to SMT logic (section 4.4.4), and (b) reconstruction of counterexamples from models returned by the SMT solver (section 4.4.5). 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

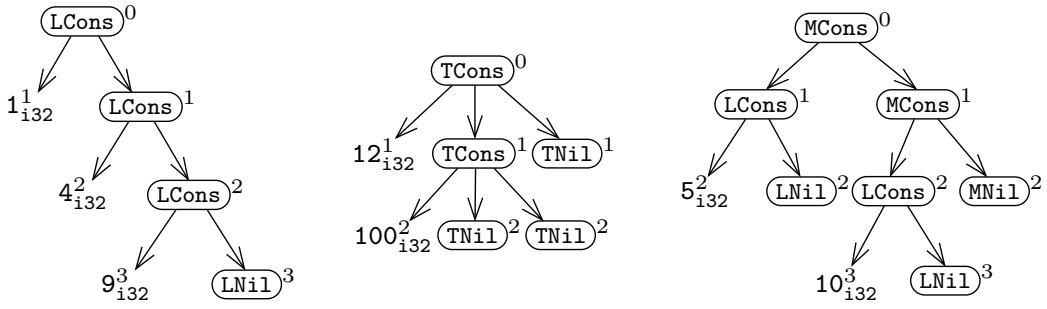
Consider the proof obligation for (I2) invariant $\mathtt{sum}_S = \mathtt{sum}_C$ across edge (S2:C2)→(S2:C2) in fig. 9a: $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2}\rightarrow\mathtt{S5}\rightarrow\mathtt{S2}, \mathtt{C2}\rightarrow\mathtt{C4}\rightarrow\mathtt{C2})\{\mathtt{sum}_S = \mathtt{sum}_C\}$, where the node invariant $\phi_{\mathtt{S2:C2}}$ contains the recursive relation $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$. The corresponding (simplified) first-order logic formula for this proof obligation is: $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C) \wedge (\mathtt{sum}_S = \mathtt{sum}_C) \wedge (\mathtt{l}_S \text{ is } \mathtt{LCons}) \wedge (\mathtt{l}_C \neq 0) \Rightarrow (\mathtt{sum}_S + \mathtt{l}_S.\mathtt{val}) = (\mathtt{sum}_C + \mathtt{l}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val})$. We fail to remove the recursive relation on the LHS even after k-unrolling for any finite unrolling parameter k because both sides of $\sim$ represent list values of arbitrary length. In such a scenario, we do not know of an efficient SMT encoding for the recursive relation $\mathtt{l}_S \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$. Ignoring this recursive relation will incorrectly (although soundly) evaluate the proof obligation to false; however, for a successful equivalence proof, we need the proof discharge algorithm to evaluate it to true. Let's call this requirement (R1).

Now, consider the proof obligation formed by correlating two iterations of the loop in program $\mathcal{S}$ (with CFG fig. 8c) with one iteration of the loop in program $\mathcal{C}$ (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: $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C) \wedge (\texttt{sum}_S = \texttt{sum}_C) \wedge (\texttt{l}_S \text{ is } \texttt{LCons}) \wedge (\texttt{l}_S\texttt{.tail} \text{ is } \texttt{LCons}) \Rightarrow (\texttt{sum}_S + \texttt{l}_S\texttt{.val} + \texttt{l}_S\texttt{.tail.val}) = (\texttt{sum}_C + \texttt{l}_C \overset{\texttt{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val})$. 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 depth of an ADT value v, we view the value as a tree $\mathcal{T}(v)$. The internal nodes of $\mathcal{T}(v)$ represent ADT data constructors and the leafs (also called *terminals*) represent scalar values (e.g. bitvector literals). The depth of a data 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. Figure 10 shows the tree representation and depths for multiple ADT values.



(a) `List` = `LNil` | `LCons(i32, List)` **(b)** `Tree` = `TNil` | `TCons(i32, Tree, Tree)` **(c)** `Matrix` = `MNil` | `MCons(List, Matrix)`

Figure 10: Tree representation of three values, each of type `List`, `Tree` and `Matrix` respectively. The depths are shown as superscripts for each node in the trees.

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:

$$l_1 \approx_a l_2 \Rightarrow l_1 \approx_b l_2 \Rightarrow l_1 \sim l_2 \Rightarrow l_1 \sim_b l_2 \Rightarrow l_1 \sim_a l_2 \tag{7}$$

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 identical till depth d *and* the corresponding terminal values in both d-depth identical structures are also equal. Note that these conditions only require scalar equalities. $l_1 \sim_d l_2$ can be identified through 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: $(true, true, l \text{ is } \texttt{LNil}, p = 0)$, $(l \text{ is } \texttt{LCons}, p \neq 0, l.\texttt{val}, p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val})$ and $(l \text{ is } \texttt{LCons}, p \neq 0, l.\texttt{tail}, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{next}))$. Keeping only those correlation tuples that relate scalar expressions, the above condition reduces to the SMT-encodable predicate:

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

- Recall that $l_1 \approx_d l_2 \Leftrightarrow \Gamma_d(l_1) \wedge \Gamma_d(l_2) \wedge l_1 \sim_d l_2$. $\Gamma_d(l)$ is equivalent to the condition that the tree nodes at depths $> d$ are unreachable. This is achieved through expanding 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 $(l \text{ is } \texttt{LNil}) \vee ((l \text{ is } \texttt{LCons}) \wedge (l.\texttt{tail} \text{ is } \texttt{LNil}))$. Similarly, $\Gamma_2(\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p))$ is equivalent to $(p = 0) \vee ((p \neq 0) \wedge (p \overset{\mathbb{m}}{\rightarrow}_{\texttt{lnode}} \texttt{next} = 0))$. Finally, $l \approx_2 \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p) \Leftrightarrow \Gamma_2(l) \wedge \Gamma_2(\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)) \wedge l \sim_2 \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$.

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

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

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

Similarly, the proof obligation $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}{\rightarrow}\texttt{S5}{\rightarrow}\texttt{S2}{\rightarrow}\texttt{S5}{\rightarrow}\texttt{S2}, \texttt{C2}{\rightarrow}\texttt{C4}{\rightarrow}\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$ successfully evaluates to false using a depth 2 underapproximation of the precondition $\phi_{\texttt{S2:C2}}$.

In the depth 2 underapproximate version, we try to prove that if the equal lists $\texttt{l}_S$ and $\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$ have exactly two nodes[9], 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.

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

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 m to an array that maps the regions starting at addresses 0x123 and 0x456 (regions of size 'sizeof(lnode)') to memory objects of type lnode (with the val and next fields shown for each object). All other addresses (except the ones for which an explicit mapping is available), m provides a default byte-value 77 (shown as () $\mapsto$ 77) in this counterexample.

This counterexample satisfies the preconditions $\texttt{l}_S \approx_2 \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$, $\texttt{sum}_S = \texttt{sum}_C$ and the path conditions. Further, when the paths S2→S5→S2→S5→S2 and C2→C4→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).

[9]The underapproximation restricts both lists to have at most two nodes; the path condition for S2→S5→S2→S5→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 $(\texttt{S3:C5}) \rightarrow (\texttt{S3:C3})$ while checking if the (I4) invariant, $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$, holds at $(\texttt{S3:C3})$: $\{\phi_{\texttt{S3:C5}}\}(\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}, \texttt{C5}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)\}$. Here, a recursive relation is present both in the precondition $\phi_{\texttt{S3:C5}}$ ((I8)) and in the postcondition ((I4)) and we are unable to remove them after k-unrolling. When lowered to first-order logic through $\texttt{WP}_{\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3},\texttt{C5}\rightarrow\texttt{C3}}$, this translates to (showing only relevant relations):

$$\begin{aligned}(\texttt{i}_S = \texttt{i}_C \wedge \texttt{p}_C = \texttt{malloc}() \wedge \texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)) \\ \Rightarrow (\texttt{LCons}(\texttt{i}_S, \texttt{l}_S) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}'}(\texttt{p}_C))\end{aligned} \tag{8}$$

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

$$\texttt{m}' \Leftrightarrow \texttt{m}[\texttt{p}_C{+}\texttt{offsetof}(\texttt{lnode}, \texttt{val}) \leftarrow \texttt{i}_C]_{\texttt{i32}}[\texttt{p}_C{+}\texttt{offsetof}(\texttt{lnode}, \texttt{next}) \leftarrow \texttt{l}_C]_{\texttt{i32}} \tag{9}$$

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

3.6.1 LHS-to-RHS Substitution and RHS Decomposition

We start by utilizing the $\sim$ relationships in the LHS (antecedent) of '$\Rightarrow$' to rewrite eq. (8) so that the ADT variables (e.g., $\texttt{l}_S$) in its RHS (consequent) are substituted with the lifted $\mathcal{C}$ values (e.g., $\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$). Thus, we rewrite eq. (8) to:

$$\begin{aligned}(\texttt{i}_S = \texttt{i}_C \wedge \texttt{p}_C = \texttt{malloc}() \wedge \texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)) \\ \Rightarrow (\texttt{LCons}(\texttt{i}_S, \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}'}(\texttt{p}_C))\end{aligned} \tag{10}$$

Next, we decompose the RHS by decomposing the recursive relation in the RHS followed by RHS-breaking. This process reduces eq. (10) into the following smaller proof obligations (LHS denotes the antecedent of the proof obligation in eq. (10)): (a) $\texttt{LHS} \Rightarrow (\texttt{p}_C \neq 0)$, (b) $\texttt{LHS} \wedge (\texttt{p}_C \neq 0) \Rightarrow (\texttt{i}_S = \texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{val})$, and (c) $\texttt{LHS} \wedge (\texttt{p}_C \neq 0) \Rightarrow \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{next})$.

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

1. The first proof obligation with postcondition $(\texttt{p}_C \neq 0)$ evaluates to *true* because the LHS ensures that $\texttt{p}_C$ is the return value of an allocation function (i.e. malloc) which must be non-zero due to the ($\mathcal{C}$ fits) assumption.

2. The second proof obligation with postcondition $(\texttt{i}_S = \texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{val})$ also evaluates to *true* because $\texttt{i}_C$ is written at address $\texttt{p}_C + \texttt{offsetof}(\texttt{lnode}, \texttt{val})$ in $\mathbb{m}'$ (eq. (9)) and the LHS ensures that $\texttt{i}_S = \texttt{i}_C$.

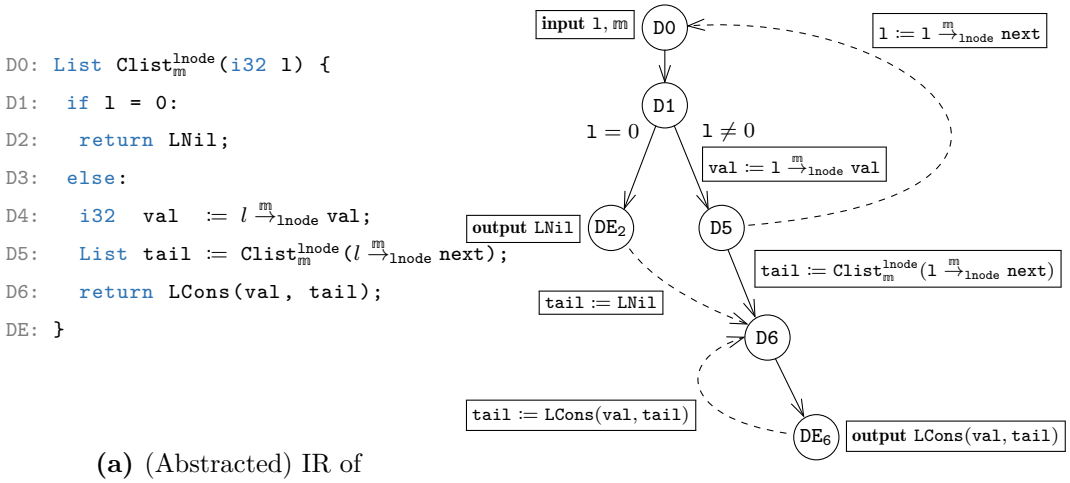
For ease of exposition, we simply the postcondition of the third proof obligation by rewriting $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{next})$ to $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C)$. This simplification is valid because $\texttt{l}_C$ is written to address $\texttt{p}_C + \texttt{offsetof}(\texttt{lnode}, \texttt{next})$ in $\mathbb{m}'$ (eq. (9)). Also, we have already shown that $(\texttt{p}_C \neq 0)$ holds due to the ($\mathcal{C}$ 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:

$$\texttt{LHS} \Rightarrow \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C) \tag{11}$$

Hence, we are interested in proving equality between two List values lifted from $\mathcal{C}$ 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 Deconstruction Programs for Lifted Values

Consider a program that recursively calls the definition (i.e. unrolling procedure) of $\texttt{Clist}^{\texttt{lnode}}$ (eq. (2)) to deconstruct $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l)$. For example, $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l)$ may yield a recursive call to $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{next})$ and so on, until the argument



(a) (Abstracted) IR of Deconstruction Program

(b) CFG of Deconstruction Program

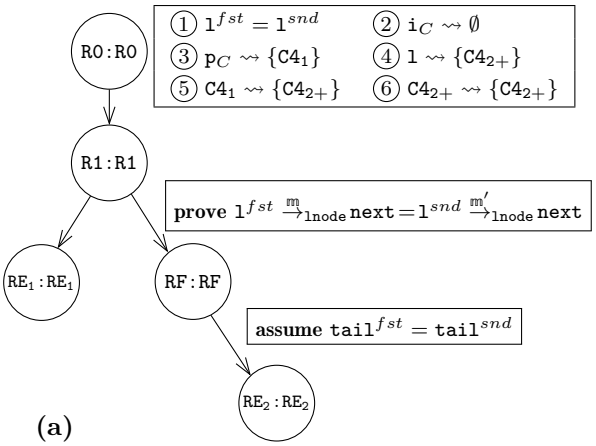
Figure 11: IR and CFG representation of deconstruction program based on the lifting constructor $\texttt{Clist}^{\texttt{lnode}}$ defined in eq. (2). In fig. 11a, `D6` contains a recursive function call. In fig. 11b, the square boxes show the transfer functions for the deconstruction program. The dashed edges represent the recursive function call in the CFG representation as shown in fig. 11b.

becomes zero. This program essentially deconstructs $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(l)$ into its terminal (scalar) values and reconstructs a `List` value equal to the value represented by $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(l)$. We call this program a *deconstruction program* based on the lifting constructor $\texttt{Clist}^{\texttt{lnode}}$. Figure 11 shows the IR and CFG representation of the deconstruction program for the lifting constructor $\texttt{Clist}^{\texttt{lnode}}$.

Theorem 1. *Under an antecedent* `LHS`, $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\mathtt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\mathtt{l}_C)$ *holds* if and only if *the two deconstruction programs* $\mathcal{D}_1$ *and* $\mathcal{D}_2$*, based on* $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)$ *and* $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\mathtt{l}_C)$*, are equivalent. The equivalence must ensure that the observables generated by both programs (i.e. output* `List` *values) are equal, given the that inputs* $(\mathtt{l}_C, \mathtt{m})$ *and* $(\mathtt{l}_C, \mathtt{m}')$ *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 $\mathcal{D}_1$ and $\mathcal{D}_2$ are their output `List` values. Also, the value represented by a lifted expression is equal to the output of its deconstruction program. Thus, a successful equivalence proof ensures equal values represented by the lifting constructors and vice versa. □

Thus, to check if $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\mathtt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\mathtt{l}_C)$ holds; we instead check if



(a) Decons-PCFG

Figure 12: 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 ??, D0 represents the unrolling procedure entry node, and the square boxes show the transfer functions of the unrolling procedure (eq. (2)). The dashed edges represent a recursive function call. In fig. 12a, the square box to the right of node D0:D0 contains the inferred invariants for this decons-PCFG.

a bisimulation relation exists between their respective deconstruction programs $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ (implying equivalence). Theorem 1 generalizes to arbitrary lifted expressions with potentially different $\mathcal{C}$ 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 between $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ — to distinguish this product-CFG from the top-level product-CFG that relates $\mathcal{S}$ and $\mathcal{C}$, we call this product-CFG that relates two deconstruction programs, a *deconstruction product-CFG* or *decons-PCFG* for short.

The decons-PCFG for the proof obligation in eq. (11) is shown in fig. 12. 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., $\mathtt{p}_C$). To check bisimulation between the programs that deconstruct $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(\mathtt{l}_C)$ and $\mathtt{Clist}_{\mathbb{m}'}^{\mathtt{lnode}}(\mathtt{l}_C)$ (i.e. $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ respectively), the decons-PCFG correlates one unrolling of the first program with one unrolling

of the second program, as defined by the unrolling procedure in eq. (2). Thus, the PC-transition correlations of $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ 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., `D0:D0` in fig. 12a). A recursive call becomes a back-edge in the decons-PCFG that terminates at the recursive-node. At the start of both deconstruction programs, $\mathtt{l}^{fst} = \mathtt{l}^{snd} = \mathtt{l}_C$ — the same $\mathtt{l}_C$ is passed to both $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$, only the memory states $\mathtt{m}^{fst} = \mathtt{m}$ and $\mathtt{m}^{snd} = \mathtt{m}'$ are different. The bisimulation check thus involves checking that if the invariant $\mathtt{l}^{fst} = \mathtt{l}^{snd}$ holds at the recursive-node, then during one iteration of the unrolling procedure in both programs:

1. The `if` condition $(\mathtt{l}^{fst} = 0)$ in $\mathcal{D}^{fst}$ is equal to the corresponding `if` condition $(\mathtt{l}^{snd} = 0)$ in $\mathcal{D}^{snd}$: $(\mathtt{l}^{fst} = 0) = (\mathtt{l}^{snd} = 0)$.

2. If the `if` condition evaluates to false in both $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$, then observable values (that are used in the construction of the list) are equal:
$((\mathtt{l}^{fst} \neq 0) \wedge (\mathtt{l}^{snd} \neq 0)) \Rightarrow (\mathtt{l}^{fst} \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val} = \mathtt{l}^{snd} \xrightarrow{\mathtt{m}'}_{\mathtt{lnode}} \mathtt{val})$.

3. If the `if` condition evaluates to false in both $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$, then the invariant holds at the beginning of the programs invoked through the recursive call. This involves checking equality of the arguments to the recursive call:
$((\mathtt{l}^{fst} \neq 0) \wedge (\mathtt{l}^{snd} \neq 0)) \Rightarrow (\mathtt{l}^{fst} \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next} = \mathtt{l}^{snd} \xrightarrow{\mathtt{m}'}_{\mathtt{lnode}} \mathtt{next})$.

The first check succeeds due to the invariant $\mathtt{l}^{fst} = \mathtt{l}^{snd}$. For the second and third checks, we additionally need to reason that the memory objects $\mathtt{l} \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{l} \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next}$ cannot alias with the writes (in $\mathtt{m}'$ in eq. (9)) to the newly allocated objects $\mathtt{p}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{p}_C \xrightarrow{\mathtt{m}}_{\mathtt{lnode}} \mathtt{next}$. We capture this aliasing information 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 *may-point-to* information for each program value using a flow-sensitive version of Andersen's algorithm [10]. The range of this computed may-point-to function is the set of *region labels*, where each region label identifies a set of memory objects. The sets of memory objects identified by two distinct region labels are necessarily disjoint. We write $p \rightsquigarrow \{\mathtt{R}_1, \mathtt{R}_2\}$ 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 $\mathcal{C}$ 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 A_1, identifies the set of memory objects that were allocated by the most recent execution of A, and (b) the second region label, called A_{2+}, identifies the set of memory objects that were allocated by older (not the most recent) executions of A. We also include a special heap region, $\mathcal{H}$ to represent the rest of the memory not covered by the allocation site regions.

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

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

3.6.5 Transferring Points-to Information to Decons-PCFG

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

Our points-to analysis on the $\mathcal{C}$ program (in fig. 2b) determines that at PC C5 (i.e. start of the product-CFG edge (S3:C5)→(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 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. 12a). To identify such points-to invariant, we run our points-to analysis on the deconstruction programs (i.e. $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$) before comparing them for equivalence. To model procedure calls, A *supergraph* is created with edges representing control flow to (and from) the entry (and exits) of the program respectively (e.g., dashes edges in fig. 11b). To see why $\texttt{l}^{snd} \rightsquigarrow \{\texttt{C4}_{2+}\}$ is an inductive invariant at the recursive-node:

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

The same analysis is run for both $\mathcal{C}$ and the deconstruction programs. For a deconstruction program $\mathcal{D}$, the boundary condition (at entry) for the points-to

analysis is based on the results of the points-to analysis on $\mathcal{C}$ at the PC where the proof obligation is being discharged. For example, the points-to information of $\mathcal{C}$ PC `C5` (in fig. 1b) is used during the points-to analysis on $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ in fig. 12.

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. 12a, and consequently, successfully discharge the proof obligation $\{\phi_{\texttt{S3:C5}}\}(\texttt{S3}{\rightarrow}\texttt{S5}{\rightarrow}\texttt{S3}, \texttt{C5}{\rightarrow}\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(\texttt{l}_C)\}$ in table 1. The points-to analysis is further discussed in section 4.1.

3.6.6 Summary of Type III Proof Discharge Algorithm

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

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

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

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

```
Function Prove(({φ_s})(e)({φ_d}), k, d_o, d_u)
    F ↤ LowerToFOL(({φ_s})(e)({φ_d}));
    foreach LHS ⇒ RHS_i in RHSBreak(F) do
        if Solve(LHS, RHS_i, k, d_o, d_u) = False(Γ) then
            return False(Γ);
        end
    end
    return True;
end
Function Solve(LHS, RHS, k, d_o, d_u)
    (LHS_k, RHS_k) ↤ DecomposeAndUnroll(LHS, RHS, k);
    switch Categorize(LHS_k, RHS_k) do
        case Type I do
            return SMTProve(LHS_k ⇒ RHS_k);
        case Type II do
            LHS_o ↤ Overapproximate(LHS, d_o);
            if SMTProve(LHS_o ⇒ RHS_k) = True then
                return True;
            end
            LHS_u ↤ Underapproximate(LHS, d_u);
            if SMTProve(LHS_u ⇒ RHS_k) = False(Γ) then
                return False(Γ);
            end
            return False(∅);
        case Type III do
            RHS′ ↤ RewriteRHSUsingLHS(LHS, RHS);
            foreach P_i ⇒ RHS_i in DecomposeAndRHSBreak(RHS′) do
                if RHS_i = l_1 ∼ l_2 then
                    (D_1, D_2) ↤ GetDeconstructionPrograms(l_1, l_2);
                    if CheckEquivalence(LHS ∧ P_i, D_1, D_2) = False then
                        return False(∅);
                    end
                else if Solve(LHS ∧ P_i, RHS_i, k, d_o, d_u) = False(Γ) then
                    return False(Γ);
                end
            end
            return True;
        end
    end
end
```

Algorithm 1: Summary of the Proof Discharge Algorithm

3.7 Overview of Proof Discharge Algorithm

Algorithm 1 gives a basic pseudo-code of our proof discharge algorithm. The top-level function responsible for discharging Hoare triple proof obligations is: $Prove()$. $Prove()$ accepts the proof obligation along with the categorization (k) and approximation (d_o and d_u) parameters. $Prove()$ either returns `True` representing a successful proof attempt, or it returns `False`(Γ), where Γ is a set of counterexamples. Recall that our proof discharge algortihm is *sound* and may return `False`($\emptyset$) to indicate a failed (proof and counterexample generation) attempt. As discussed in section 2.6, we lower the Hoare triple into a first-order logic formula (F) using weakest-precondition predicate transformer. This is followed by `RHS`-breaking (introduced in section 3.1), which results in multiple smaller proof obligations. $Prove$ attempts to prove each of these proof obligations individually through a call to $Solve()$. If any one of these queries fail, we immediately stop and return `False` with the counterexamples in Γ — a counterexample to one of the smaller queries is also a counterexample to the original query.

$Solve()$ is responsible for discharging these smaller queries. Inputs include `LHS`, `RHS` (representing the proof obligation P : `LHS` $\Rightarrow$ `RHS`); along with the parameters: k, d_o and d_u. $Solve()$ begins by finding the k-unrolled form of P and categorizes it into one of the three types. As discussed in section 3.4, we simply discharge a type I query using SMT solvers (through $SMTProve()$). $SMTProve()$ is responsible for (a) translating the input formula (absent of recursive relations) to SMT logic, and (b) reconstruction of counterexamples from the models returned by the SMT solvers. These two topics are further explored in sections 4.4.4 and 4.4.5 respectively. As summarized in section 3.5.4, for a type II query, we attempt to prove its overapproximate version first. In case of a failure, we attempt to disprove (and generate counterexamples) its underapproximate version. If both attempts fail, we *soundly* return `False` (without counterexamples). Lastly, a type III query P is discharged as detailed in section 3.6.6. In brief, we decompose and perform `RHS`-breaking on P. This results in smaller proof obligations; ones without a recursive relation in its `RHS`, are type II queries and discharged through a recursive call to $Solve()$. For those containing a recursive relation $l_1 \sim l_2$ in their `RHS`, we reformulate the query as an equivalence check between the deconstruction program of l_1 and l_2 respectively. If any one of these queries fail, we immediately return `False`

with the counterexamples (if available). Otherwise, we have successfully proven a type III query and return `True`.

4 Spec-to-C Equivalence Checker

In this section, we present our automatic equivalence checker algorithm S2C. S2C is able to search for a bisimulation based proof of equivalence between Spec and C programs. We start with a dataflow formulation of our points-to analysis. Recall that the points-to analysis is used to identify may-point-to invariants in the C program, as well as deconstruction programs. As described in section 1.2, S2C is based on three primary algorithms: (a) an algorithm to incrementally construct a product-CFG by correlating program executions across the Spec and C procedures respectively, (b) an algorithm to identify inductive invariants at intermediate PCs in the (partially constructed) product-CFG, and (c) an algorithm for solving proof obligations generated by the first two algorithms. The last section illustrates our proof discharge algorithm through sample proof obligations. We describe our counterexample-guided best-first search algorithm for construction of a product-CFG in section 4.2. This is followed by a dataflow formulation of our counterexample-guided invariant inference algorithm in section 4.3. We finish with a comprehensive analysis of our proof discharge algorithm and its related subprocedures.

Table 2: Dataflow Formulation of the Points-to Analysis

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

4.1 Points-to Analysis

Recall that in section 3.6.3, we needed to reason about aliasing to successfully discharge a type III proof obligation. These aliasing relationships are described in section 3.6.4 and used in section 3.6.5 to successfully discharge the proof obligation. A points-to analysis is used to identify these aliasing relationships in $\mathcal{C}$ as well as each deconstruction program $\mathcal{D}$. We present a dataflow formulation of our points-to analysis as shown in table 2. We start by identifying the set $\mathbb{R}$ of all region labels representing mutually non-overlapping regions of the C memory state $\mathtt{mm}$. For each call to `malloc()` at PC A, we add A_1 and A_{2+} to $\mathbb{R}$. Recall that A_1 represents the region of memory returned by the *most recent* execution of A. A_{2+} represents the region of memory returned by older (i.e. all but most recent) executions of A. $\mathbb{R} = \bigcup_A \{A_1, A_{2+}\} \cup \{\mathcal{H}\}$, where $\mathcal{H}$ is the region of memory $\mathtt{mm}$ not covered by the labels associated with allocation sites.[10]

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

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

Next, we discuss the transfer function f_e for our points-to analysis. For an IR instruction $\mathtt{x} \coloneqq \mathtt{c}$, for constant $\mathtt{c}$, the transfer function updates $\Delta(\mathtt{x}) \coloneqq \emptyset$. For instruction $\mathtt{x} \coloneqq \mathtt{y}\ \mathtt{op}\ \mathtt{z}$ (for some arithmetic or logical operator $\mathtt{op}$), we update

[10] TODO: per procedure or global?

$\Delta(\mathtt{x}) \coloneqq \Delta(\mathtt{y}) \cup \Delta(\mathtt{z})$. For a load instruction $\mathtt{x} \coloneqq \mathtt{mm}[y]_{\mathtt{T}}$, we update $\Delta(\mathtt{x}) \coloneqq \bigcup_{t \in \Delta(y)} \Delta(t)$. For a store instruction $\mathtt{mm} \coloneqq \mathtt{mm}[x \leftarrow y]_{\mathtt{T}}$, for all $t \in \Delta(\mathtt{x})$, we update $\Delta(t) \coloneqq \Delta(t) \cup \Delta(y)$. For a malloc instruction $\mathtt{x} \coloneqq \mathtt{malloc}_A()$ (where A represents the allocation site), we perform the following steps (in order):

1. Convert all existing occurrences of A_1 to A_{2+}, i.e., for all $t \in (\mathbb{S}^{\mathcal{C}} \cup \mathbb{R})$, if $A_1 \in \Delta(t)$, then update $\Delta(t) \coloneqq (\Delta(t) \setminus \{A_1\}) \cup \{A_{2+}\}$.
2. Update $\Delta(\mathtt{x}) \coloneqq \{A_1\}$.
3. Update $\Delta(A_{2+}) \coloneqq \Delta(A_{2+}) \cup \Delta(A_1)$.
4. Update $\Delta(A_1) \coloneqq \emptyset$.

For function calls, a *supergraph* is created by adding control flow edges from the call-site to the procedure head (copying actual arguments to the formal arguments) and from the procedure exit to the program point just after the call-site (copying returned value to the variable assigned at the callsite), e.g., in fig. 12, the dashed edges represent supergraph edges.

The allocation-site abstraction (with a bounded-depth call stack) is known to be effective at disambiguating memory regions belonging to different data structures [26, 15, 11]. In our work, we also need to reason about non-aliasing of the most-recently allocated object (through a `malloc` call) and the previously-allocated objects (as in the `List` construction example). The coarse-grained $\{1, 2+\}$ categorization of allocation recency is effective for such disambiguation.

4.2 Counterexample-guided Product-CFG Construction

S2C constructs a product-CFG incrementally to search for an observably-equivalent bisimulation relation between the CFGs of a Spec procedure $\mathcal{S}$ and a C procedire $\mathcal{C}$. Multiple candidate product-CFGs are partially constructed during this search; the search completes when one of these candidates yield an equivalence proof.

Anchor nodes are identified in the CFGs of $\mathcal{S}$ and $\mathcal{C}$, and represents the source and destination nodes (i.e. IR PCs) of paths chosen to be correlated between the

two programs. The algorithm ensures that every cycle in both $\mathcal{S}$ and $\mathcal{C}$ contains at least one anchor node. The start and exit nodes are always anchor nodes. Also, for every function call, the nodes just before and after its callsite are considered anchor nodes. For example, in fig. 3b, `C4` and `C5` are anchor nodes around the call to `malloc`. The selected anchor nodes for the CFGs in figs. 3a and 3b are: $\{\texttt{S0}, \texttt{S3}, \texttt{SE}\}$ and $\{\texttt{C0}, \texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{CE}\}$ respectively. For each anchor node in $\mathcal{C}$, our search algorithm searches for a correlated anchor node in $\mathcal{S}$ — if a (partially constructed) product-CFG π contains a product-CFG node $(n_S : n_C)$, then π correlates node n_C in $\mathcal{C}$ with node n_S in $\mathcal{S}$. The search procedure begins with a single partially-constructed product-CFG π_{init}. π_{init} contains exactly one node (`S0:C0`) that encodes the correlation of the entry nodes (i.e. `S0` and `C0`) of $\mathcal{S}$ and $\mathcal{C}$.

At each step of the incremental construction process, a node $(n_S : n_C)$ is chosen in a product-CFG π and a path ρ_C in $\mathcal{C}$ starting at n_C (and ending at an anchor node in $\mathcal{C}$) is selected. Then, we enumerate potentially correlated paths in $\mathcal{S}$ for the path ρ_C in $\mathcal{C}$. For example, during construction of the product-CFG shown in fig. 4, say we select the product-CFG node (`S3:C3`). We choose the $\mathcal{C}$ path `C3`→`C4` and enumerate its potential correlations (i.e. paths in $\mathcal{S}$ starting at `S3`): ϵ, `S3`→`S5`→`S3`, `S3`→`S5`→`S3`→`S5`→`S3`, …, `S3`→$(\texttt{S5}\rightarrow\texttt{S3})^{\mu}$. The *unroll factor* μ is a fixed parameter of the algorithm and represents the maximum number of iterations of a loop (in $\mathcal{S}$), that may be correlated with a path ρ_C in $\mathcal{C}$. Importantly, for paths ρ_S (in $\mathcal{S}$) and ρ_C (in $\mathcal{C}$) to be considered for correlation, they must begin and end at anchor nodes, i.e. the path `S3`→`S5` is skipped during enumeration. Moreover, the path ρ_C may not contain anchor nodes in the middle. Hence, the path `C3`→`C4`→`C5` is not considered for ρ_C, instead we attempt to correlate the subpaths `C3`→`C4` and `C4`→`C5` individually.

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`→`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 $\mathcal{S}$ and $\mathcal{C}$). 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 behaviour 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 [24]. A best-first search (BFS) procedure based on this ranking criterion is used to incrementally construct a product-CFG (starting from π_{init}). For each intermediate candidate product-CFG π generated during this search procedure, an automatic invariant inference procedure (discussed next in section 4.3) is used to identify invariants at all the nodes in π. The counterexamples obtained from the proof obligations generated by this invariant inference procedure are added to the respective nodes in π; these counterexamples help rank future correlations starting at those nodes.

If after invariant inference, we realize that an intermediate candidate product-CFG π_1 is not promising enough, we backtrack and choose another candidate product-CFG π_2 and explore the potential correlations that can be added to π_2. Thus, a product-CFG is constructed one edge at a time. If at any stage, a product-CFG π contains correlations for every path in $\mathcal{C}$ and invariants ensure equal observables (i.e. *Post* holds at correlated exit nodes), we have successfully shown equivalence. This counterexample-guided BFS procedure is similar to the one described in prior work on the Counter algorithm [24].

4.2.1 Correlation in the Presence of Procedure Calls

Recall that a procedure δ in $\mathcal{S}$ and $\mathcal{C}$ may make function calls (including self calls), e.g., allocation of memory in C, traversal of a tree data structure. Recall that the nodes just before and after a function call are always considered anchor nodes. Calls to memory allocation functions in $\mathcal{C}$ (i.e. `malloc`) are handled by correlating the function call edge with the empty path (ϵ) in $\mathcal{S}$. For example, in the product-CFG shown in fig. 4, the `malloc` edge `C4`→`C5` in $\mathcal{C}$ is correlated with ϵ in $\mathcal{S}$.

For all other calls, our correlation algorithm (in section 4.2) ensures that the

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

We utilize the user-supplied input-output specification for δ' (say $(Pre_{\delta'}, Post_{\delta'})$) to obtain the desired invariants at nodes $(n_S : n_C)$ and $(n'_S : n'_C)$ in the product-CFG. A successful proof must *ensure* that $Pre_{\delta'}(\texttt{args}_{n_S},\texttt{args}_{n_C},\texttt{mm}_{n_C})$ holds at $(n_S : n_C)$. Further, the proof can *assume* that $Post_{\delta'}(\texttt{ret}_{n'_S},\texttt{ret}_{n'_C},\texttt{mm}_{n'_C})$ holds at $(n'_S : n'_C)$. Here, $\texttt{mm}_{n_C}$ and $\texttt{mm}_{n'_C}$ represents the memory states in $\mathcal{C}$ at n_C and n'_C respectively. Thus, for function calls, we inductively prove the precondition (on the arguments) at $(n_S : n_C)$ and assume the postcondition (on the returned values) at $(n'_S : n'_C)$.

```
Function bestFirstSearch(S, C, μ)
    π_init ↤ createInitProductCFG(S, C);
    Q ↤ {π_init};
    while Q is not empty do
        π_cur ↤ extractMostPromising(Q);
        InferInvariantsAndCounterexamples(π_cur);
        if getCPathToCorrelate(C, π_cur) = Found(ρ_C) then
            foreach ρ_S in enumeratePathsInS(S, ρ_C, μ) do
                π_next ↤ extendProductCFG(π_cur, ρ_S, ρ_C);
                Q ↤ Q ∪ {π_next};
            end
        else if productCFGRepresentsBisim(π_cur) then
            return Found(π_cur);
        end
    end
    return NotFound;
end
```

Algorithm 2: Pseudo-code for Best-First-Search Procedure for construction of Product-CFG

Table 3: Dataflow Formulation of the Invariant Inference Algorithm

Domain	$\left\{ (\phi_n, \Gamma_n) \mid \phi_n \text{ is a conjunction of predicates drawn from } \mathbb{G},\ \Gamma_n \text{ is a set of counterexamples} \right\}$
Direction	Forward
Boundary Condition	(ϕ_n, Γ_n) for start node : $\phi_n \leftarrow \mathit{Pre}$, $\Gamma_n \leftarrow \emptyset$
Initialization to $\top$	(ϕ_n, Γ_n) for non-start nodes : $\phi_n \leftarrow \texttt{false}$, $\Gamma_n \leftarrow \emptyset$
Transfer function across edge $e = (s \to d)$	$(\phi_d, \Gamma_d) = f_e(\phi_s, \Gamma_s)$ (shown in fig. 13a)
Meet operator $\otimes$	$(\phi_n, \Gamma_n) \leftarrow (\phi_n^1, \Gamma_n^1) \otimes (\phi_n^2, \Gamma_n^2)$ $\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2$, $\phi_n \leftarrow \mathit{StrongestInvCover}(\Gamma_n)$

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

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

$$\begin{aligned} \texttt{Inv} \ &\to \ \textstyle\sum_i c^i v^i = c \mid v^1 \odot v^2 \\ &\mid \ \alpha_S \sim \texttt{Clift}^{\texttt{T}}_{\mathbb{m}}(v_C \ldots) \end{aligned}$$

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

Figure 13: Transfer function f_e and Predicate grammar $\mathbb{G}$ for invariant inference dataflow analysis in table 3. 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. $\mathit{StrongestInvCover}(\Gamma)$ computes the strongest invariant cover for counterexamples Γ. $\texttt{exec}_e(\Gamma)$ (concretely) executes counterexamples Γ over edge e. $\mathit{Prove}(P)$ (in algorithm 1) discharges the proof obligation P, and returns either $\texttt{True}$ or $\texttt{False}(\Gamma)$.

4.3 Invariant Inference and Counterexample Generation

We formulate our counterexample-guided invariant inference algorithm as a dataflow analysis as shown in table 3. The invariant inference procedure is responsible for inferring invariants ϕ_n at each intermediate node n of a (partially constructed) product-CFG, while also generating a set of counterexamples Γ_n that represents the potential concrete machine states at n.

Given the invariants and counterexamples at node s: (ϕ_s, Γ_s), the transfer function initializes the new candidate set of counterexamples at d (Γ_d^{can}) with the

current set of counterexamples at d (Γ_d) *union*-ed with the counterexamples obtained by executing Γ_s on edge e (through $\texttt{exec}_e$). 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 $\{\phi_s\}(e)\{\phi_d^{can}\}$ (through a call to $Prove()$). If the proof succeeds ($Prove()$ returns `True`), the candidate invariant ϕ_d^{can} is returned along with the counterexamples Γ_d^{can} learned so far. Otherwise, $Prove()$ returns $\texttt{False}(\gamma_s)$. The candidate invariant ϕ_d^{can} is weakened using the counterexamples obtained (i.e. γ_s) and the proof attempt is repeated.

The candidate invariants are drawn from the predicate grammar $\mathbb{G}$ shown in fig. 13b. In addition to affine and inequality relations between bitvectors in $\mathcal{S}$ and $\mathcal{C}$, $\mathbb{G}$ supports recursive relations between an ADT variable in $\mathcal{S}$ and a lifted expression in $\mathcal{C}$. The candidate lifting constructors (i.e. $\texttt{Clift}^{\texttt{T}}$) are derived from the lifting constructors present in the precondition Pre and the postcondition $Post$, as supplied by the user. More sophisticated strategies for inference of new lifting constructors is possible.

$StrongestInvCover()$ for affine relations involve identifying the basis vectors of the kernel of the matrix formed by the counterexamples in the bitvector domain [33, 17]. For inequality relations, $StrongestInvCover(\Gamma)$ returns *true* (i.e. the weakest invariant) iff any counterexample in Γ evaluates the relation to false — this effectively simulates the Houdini approach [23]. Similarly, in case of a recursive relation $l_1 \sim l_2$, $StrongestInvCover(\Gamma)$ returns *true* iff any counterexample in Γ evalutes its η-depth over-approximation $l_1 \sim_\eta l_2$ to false, where η is a fixed parameter of the algorithm.

4.4 Proof Discharge Algorithm

4.4.1 Summary of Canonicalization Procedure

Algorithm 3 shows the pseudo-code for the canonicalization procedure. $Canonicalize(e)$ is responsible for converting an expression e to its canonical form $\hat{e}$ (introduced in section 3.2). Recall that a pseudo-variable is an expression of the form $v.\texttt{a}_1.\texttt{a}_2...\texttt{a}_\texttt{n}$, where v is a variable. Also recall that, an expression e is canonical iff each *accessor* and *sum-is* expression operate on a pseudo-variable. An

ADT expression with a data constructor, a lifting constructor or the if-then-else-operator at its top-level, is called a *foldable* expression. $Canonicalize(e)$ iteratively folds each *accessor* and *sum-is* subexpressions of e that operate on a foldable argument. Thus, $Canonicalize(e)$ returns an expression where none of the *accessor* or *sum-is* subexpressions is foldable. This condition implies the requirements of the canonical form. For example, $a + \texttt{LCons}(b, l).\texttt{tail.val}$ and $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$ is `LNil` canonicalizes to $a + l.\texttt{val}$ and $(p = 0)$ respectively.

Function $Canonicalize(e)$
  $\hat{e} \leftarrow e$;
  while e **contains** $e' = e_1.\texttt{a}^{\texttt{i}}$ **where** e_1 **is** $foldable$ **do**
    if $e_1 = V_1^{(n)}(e_1^1, e_1^2, \ldots, e_1^n)$ **then**
      $\hat{e} \leftarrow \{e' \mapsto e_1^i\}\hat{e}$;
    else if $e_1 = \underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ V_1^{(n)}(e_1^1, e_1^2, \ldots, e_1^n)\ \underline{\texttt{else}}\ e_1^{\texttt{el}}$ **then**
      if $V_1^{(n)}$ **contains** $\texttt{a}^{\texttt{i}}$ **then** $\hat{e} \leftarrow \{e' \mapsto e_1^i\}\hat{e}$;
      else $\hat{e} \leftarrow \{e' \mapsto e_1^{\texttt{el}}.\texttt{a}^{\texttt{i}}\}\hat{e}$;
    else $e_1 = \texttt{Clift}^{\texttt{T}}_{\mathbb{m}}(e^1, e^2, \ldots, e^n)$
      $\hat{e} \leftarrow \{e' \mapsto rewrite(e_1).\texttt{a}^{\texttt{i}}\}\hat{e}$;
    end
  end
  while e **contains** $e' = e_1$ is $V_2^{(m)}$ **where** e_1 **is** $foldable$ **do**
    if $e_1 = V_1^{(n)}(e_1^1, e_1^2, \ldots, e_1^n)$ **then**
      if $V_1^{(n)} = V_2^{(m)}$ **then** $\hat{e} \leftarrow \{e' \mapsto true\}\hat{e}$;
      else $\hat{e} \leftarrow \{e' \mapsto false\}\hat{e}$;
    else if $e_1 = \underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ V_1^{(n)}(e_1^1, e_1^2, \ldots, e_1^n)\ \underline{\texttt{else}}\ e_1^{\texttt{el}}$ **then**
      if $V_1^{(n)} = V_2^{(m)}$ **then** $\hat{e} \leftarrow \{e' \mapsto c_1\}\hat{e}$;
      else $\hat{e} \leftarrow \{e' \mapsto \neg c_1 \wedge (e_1^{\texttt{el}}$ is $V_2^{(m)})\}\hat{e}$;
    else $e_1 = \texttt{Clift}^{\texttt{T}}_{\mathbb{m}}(e^1, e^2, \ldots, e^n)$
      $\hat{e} \leftarrow \{e' \mapsto rewrite(e_1).\texttt{a}^{\texttt{i}}\}\hat{e}$;
    end
  end
  return $\hat{e}$;
end

Algorithm 3: Pseudo-code for Canonicalization Procedure

4.4.2 Summary of Unification Procedure

Algorithm 4 shows the pseudo-code for the unification algorithm introduced in section 3.2. $\theta(p_1, e_1, p_2, e_2)$ is responsible for unifying expressions e_1 and e_2 under the expression path conditions p_1 and p_2 respectively. θ either fails to unify with the `Fail` output, or it successfully returns $\texttt{Succ}(S)$, where S is the set of correlation tuples that relate (a) either two atomic expressions, or (b) an atom

with an non-atomic expression. $\theta(p_1, e_1, p_2, e_2)$ terminates when one of e_1 and e_2 is an atomic expression. In case both e_1 and e_2 contains a data constructor at their top-level, θ attempts to recursively unify the data constructors and their corresponding children. If exactly one of e_1 and e_2 is a `if`-`then`-`else` expression, θ attempts to unify both branches of `if`-`then`-`else` (along with the path conditions) with the other expression and return whichever succeeds. If both e_1 and e_2 are `if`-`then`-`else` expressions, θ attempts to recursively unify their children. θ uses the $\sqcup$-operator to combine the results of successive self-calls. $A \sqcup B$ is equal to $\texttt{Succ}(S_1 \cup S_2)$ if $A = \texttt{Succ}(S_1)$ and $B = \texttt{Succ}(S_2)$; otherwise (if one of A and B is `Fail`), $A \sqcup B = \texttt{Fail}$.

Function $\theta(p_1, e_1, p_2, e_2)$
  if e_1 **is** *atomic* **then**
    return $\texttt{Succ}(\{\langle p_1, e_1, p_2, e_2\rangle\})$;
  else if e_2 **is** *atomic* **then**
    return $\texttt{Succ}(\{\langle p_2, e_2, p_1, e_1\rangle\})$;
  else if $e_1 = V_1^{(n)}(e_1^1, e_1^2, \ldots, e_1^n)$ **and** $e_2 = V_2^{(m)}(e_2^1, e_2^2, \ldots, e_2^m)$ **then**
    if $V_1^{(n)} \neq V_2^{(m)}$ **then**
      return `Fail`;
    end
    return $\bigsqcup_{i \in [1,n]} \theta(p_1, e_1^i, p_2, e_2^i)$;
  else if $e_1 = V_1^{(n)}(e_1^1, e_1^2, \ldots, e_1^n)$ **and** $e_2 = \underline{\texttt{if}}\ c_2\ \underline{\texttt{then}}\ e_2^{\texttt{th}}\ \underline{\texttt{else}}\ e_2^{\texttt{el}}$ **then**
    $R^{\texttt{th}} \leftarrowtail \theta(p_1, true, p_2, c_2) \sqcup \theta(p_1, e_1, p_2 \wedge c_2, e_2^{\texttt{th}})$;
    if $R^{\texttt{th}} = \texttt{Succ}(S)$ **then return** $\texttt{Succ}(S)$;
    $R^{\texttt{el}} \leftarrowtail \theta(p_1, true, p_2, \neg c_2) \sqcup \theta(p_1, e_1, p_2 \wedge \neg c_2, e_2^{\texttt{el}})$;
    if $R^{\texttt{el}} = \texttt{Succ}(S)$ **then return** $\texttt{Succ}(S)$;
    return `Fail`;
  else if $e_1 = \underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ e_1^{\texttt{th}}\ \underline{\texttt{else}}\ e_1^{\texttt{el}}$ **and** $e_1 = V_2^{(m)}(e_2^1, e_2^2, \ldots, e_2^m)$ **then**
    $R^{\texttt{th}} \leftarrowtail \theta(p_1, c_1, p_2, true) \sqcup \theta(p_1 \wedge c_1, e_1^{\texttt{th}}, p_2, e_2)$;
    if $R^{\texttt{th}} = \texttt{Succ}(S)$ **then return** $\texttt{Succ}(S)$;
    $R^{\texttt{el}} \leftarrowtail \theta(p_1, \neg c_1, p_2, true) \sqcup \theta(p_1 \wedge \neg c_2, e_1^{\texttt{el}}, p_2, e_2)$;
    if $R^{\texttt{el}} = \texttt{Succ}(S)$ **then return** $\texttt{Succ}(S)$;
    return `Fail`;
  else $e_1 = \underline{\texttt{if}}\ c_1\ \underline{\texttt{then}}\ e_1^{\texttt{th}}\ \underline{\texttt{else}}\ e_1^{\texttt{el}}$ **and** $e_2 = \underline{\texttt{if}}\ c_2\ \underline{\texttt{then}}\ e_2^{\texttt{th}}\ \underline{\texttt{else}}\ e_2^{\texttt{el}}$
    $R_1 \leftarrowtail \theta(p_1, c_1, p_2, c_2)$;
    $R_2 \leftarrowtail \theta(p_1 \wedge c_1, e_1^{\texttt{th}}, p_2 \wedge c_2, e_2^{\texttt{th}})$;
    $R_3 \leftarrowtail \theta(p_1 \wedge \neg c_1, e_1^{\texttt{el}}, p_2 \wedge \neg c_2, e_2^{\texttt{el}})$;
    return $R_1 \sqcup R_2 \sqcup R_3$;
  end
end

Algorithm 4: Pseudo-code for Unification Procedure

4.4.3 Summary of Iterative Unification and Rewriting Procedure

Algorithm 5 shows the pseudo-code for the iterative unification and rewriting procedure introduced in section 3.2. $\Theta(p_a, e_a, p_b, e_b)$ is responsible for unifying expressions e_a and e_b under the expression path conditions p_a and p_b respectively. Θ either fails to unify with the `Fail` output, or it successfully returns $\texttt{Succ}(S)$, where S is the set of correlation tuples that relate *only* atomic expressions. Θ attempts to iteratively (a) unify the expressions (through a call to the unification procedure θ in section 4.4), and (b) perform rewriting (of atom a_1 for those correlation tuples $\langle p_1, a_1, p_2, e_2\rangle$ where e_2 is non-atomic), followed by a recursive call to Θ. A recursive relation $l_1 \sim l_2$ is decomposed through the top-level invocation of $\Theta(true, l_1, true, l_2)$.

```
Function Θ(p_a, e_a, p_b, e_b)
    R ↤ ∅;
    S ↤ θ(p_a, e_a, p_b, e_b);
    if S = Fail then return Fail;
    foreach ⟨p_1, a_1, p_2, e_2⟩ in S do
        if e_2 is atomic then
            R ↤ R ∪ {⟨p_1, a_1, p_2, e_2⟩};
        else
            e_1 ↤ rewrite(a_1);
            R_1 ↤ Θ(p_1, e_1, p_2, e_2);
            if R_1 = Fail then return Fail;
            R ↤ R ∪ R_1;
        end
    end
    return Succ(R);
end
```

Algorithm 5: Pseudo-code for Iterative Unification and Rewriting Procedure

4.4.4 SMT Encoding of First Order Logic Formula

As summarized in algorithm 1, our proof discharge algorithm solves a proof obligation P : $\texttt{LHS} \Rightarrow \texttt{RHS}$, through a sequence of queries P_i : $\texttt{LHS}_i \Rightarrow \texttt{RHS}_i$ to off-the-shelf SMT solvers. Recall that P may contain recursive relations. However, our algorithm ensures that each P_i is free of recursive relations and only contain scalar equalities. We encode each query P_i in SMT logic with bitvector and array theories. In this section, we describe the process of encoding a proof obligation P_i into SMT logic. We begin by converting P_i : $\texttt{LHS}_i \Rightarrow \texttt{RHS}_i$ into its canonical form $\hat{P}_i$ (as described in section 4.4.1). Although $\hat{P}_i$ does not contain recursive relations,

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

1. For each top-level *accessor* expression $e = v.\mathtt{a_1}.\mathtt{a_2}...\mathtt{a_n}$, we replace it with a variable named $v \parallel \mathtt{a_1} \parallel \mathtt{a_2} \parallel \ldots \parallel \mathtt{a_n}$, where $\parallel$ concatenates two strings with a '_' character in between i.e. $"a" \parallel "b" = "a\_b"$.

2. For each ADT T with data constructors $V_1, V_2, \ldots, V_k$, we define an enumeration type $\mathcal{E}(T)$ in SMT logic with items $\mathcal{E}(V_1), \mathcal{E}(V_2), \ldots, \mathcal{E}(V_k)$ respectively. In the canonical form, each *sum-is* expression e must operate on a pseudo-variable. The last step guarantees that e must be of the form: $e = v$ is V. We replace e with the its SMT equivalent: $(v \parallel tag) = \mathcal{E}(V)$ [11].

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

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

[11] Spec does not allow naming a field of a data constructor `tag`. This prevents collision between variable names obtained due to flattening.

4.4.5 Reconciliation of Counterexamples

As detailed in section 4.4.4, each ADT variable v gets lowered into a set of scalar variables $\mathcal{F}(v)$ during SMT encoding. Evidently, the models returned by SMT solvers map these variables (in $\mathcal{F}(v)$ instead of v) to constant values. We are interested in recovering a counterexample for the original query from a model returned by the SMT solver. Recall that, these counterexamples help guide the correlation search (in section 4.2) and invariant inference (in section 4.3) procedures. The process of constructing a constant for v from the constant values returned for $\mathcal{F}(v)$ by an SMT solver is called *reconciliation*. Obviously, the reconciled counterexample must be a valid counterexample to the original proof obligation. $\mathit{Reconcile}(v : T, \gamma)$ is responsible for performing reconciliation for variable v (of type T) from the model γ (returned by a SMT solver). $\mathit{Rand}(T)$ returns an arbitrary constant of type T. For example, consider the rather contrived proof obligation P : `true` $\Rightarrow l$ is `LNil`. Clearly, any valuation of l where l is a non-empty list is a valid counterexample to P. However, a counterexample γ returned by an SMT solver must contain the mapping $\{l\_tag \mapsto \mathcal{E}(\texttt{LCons})\}$. During reconciliation, we find that l_tag is mapped to the data constructor `LCons` and recurse for each of its fields `val` and `tail`. Since γ do not contain a mapping for either of these fields, we soundly generate random constants for these instead. Note that *Reconcile* correctly constructs a non-empty but otherwise arbitrary list for l, which is indeed a counterexample to P.

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

Algorithm 6: Pseudo-code for Reconciliation Procedure

4.4.6 Value Tree Representation

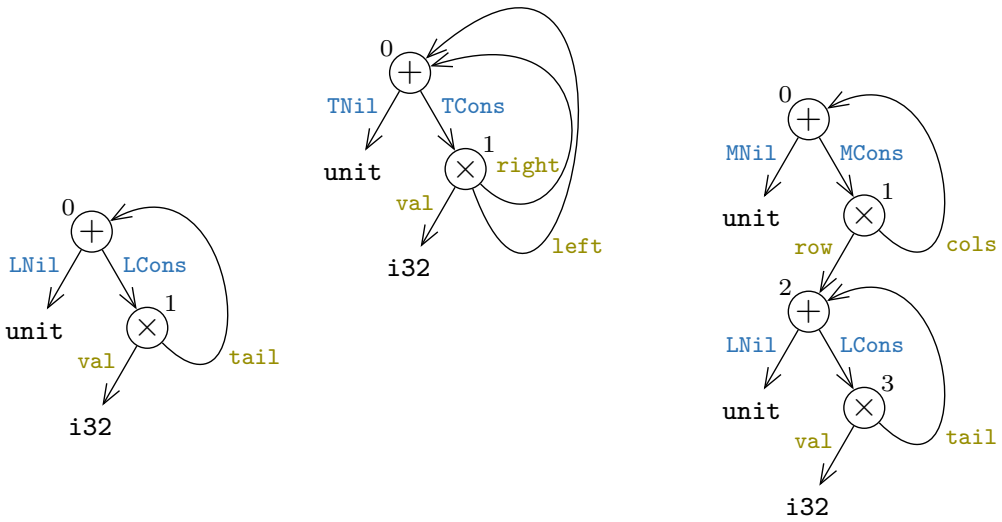
This section introduces a graphical representation of expressions which encodes both its canonical value and its deconstruction program at the same time. We call this the 'Value Tree' and we use $\mathcal{V}(e)$ to denote the value tree of e. The value properties allow us to consolidate the following procedures used during conversion of a proof obligation *with* recursive relations into its canonical form: (a) decomposition of recursive relations, (b) over- and under-approximation of recursive relations, and (c) canonicalization of a proof obligation without recursive relations (required for encoding in SMT logic). On the program side, instead of checking equivalence of the deconstruction programs for a recursive relation $l_1 \sim l_2$, we check 'equivalence' of the value graphs $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ instead.

We begin with a formal description of ADTs. This allows us to first introduce a graphical representation of types, which is analogous to the value tree representation, but simpler. Let's call this the 'Type Tree' representation. Recall that ADTs are simply 'sum of product' types where each data construction represents a variant (of the sum-type) and each data construction contains values for each of its fields (of the product-type). On top of ADTs, Spec also has built-in scalar types: `unit`, `bool` and `i<N>`. Types in Spec can be represented in *first order recursive*

types using the product ($\times$) and sum ($+$) type constructors; and the scalar types (i.e. nullary type constructors). The type system is characterized by the grammar $\mathbb{T}$ as follows:

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

Every type in Spec can be encoded as a closed term (i.e. term without free variables) in $\mathbb{T}$. For example, the `List` type can be written as $\mu\alpha.\texttt{unit} + (\texttt{i32} \times \alpha)$. Note the use of a type variable α which is bound using μ to represent recursion. Consider the mutually-recursive types `Tree` $=$ `TNil`|`TCons`(`i32`, `Forest`) and `Forest` $=$ `FNil`|`FCons`(`Tree`, `Forest`). `Tree` represents a generic `i32` tree datatype: either its empty or it contains a value along with a *list of trees* for its children (represented by `Forest`). `Tree` can be represented in $\mathbb{T}$ as: $\mu\alpha.\texttt{unit} + (\texttt{i32} \times (\mu\beta.\texttt{unit} + (\alpha \times \beta)))$, where α and β stand for the recursive types `Tree` and `Forest` respectively.

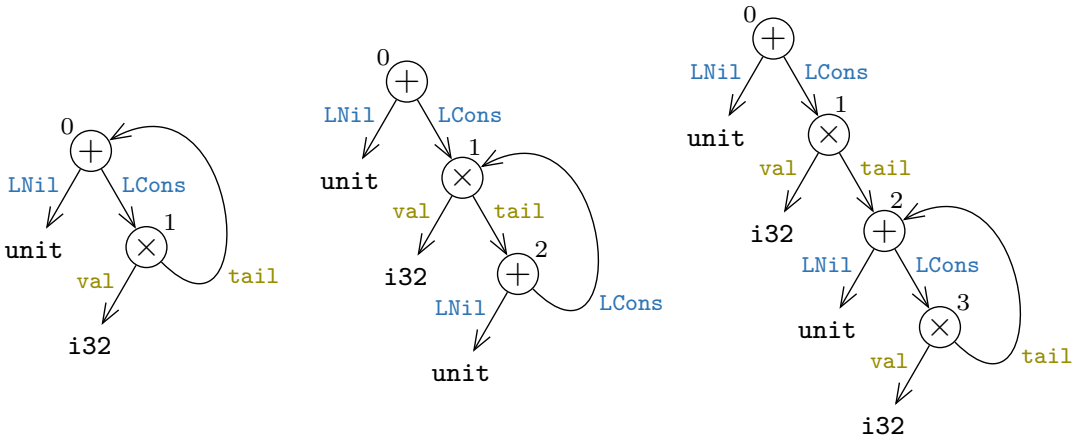


(a) `List` = `LNil` | `LCons(i32, List)` **(b)** `Tree` = `TNil` | `TCons(i32, Tree, Tree)` **(c)** `Matrix` = `MNil` | `MCons(List, Matrix)`

Figure 14: Type trees for the ADTs `List`, `Tree` and `Matrix` respectively.

Figure 14 shows the type trees based for three Spec ADTs encoded in $\mathbb{T}$. Each internal node represents either a product ($\otimes$) or a sum ($\oplus$) type constructor.

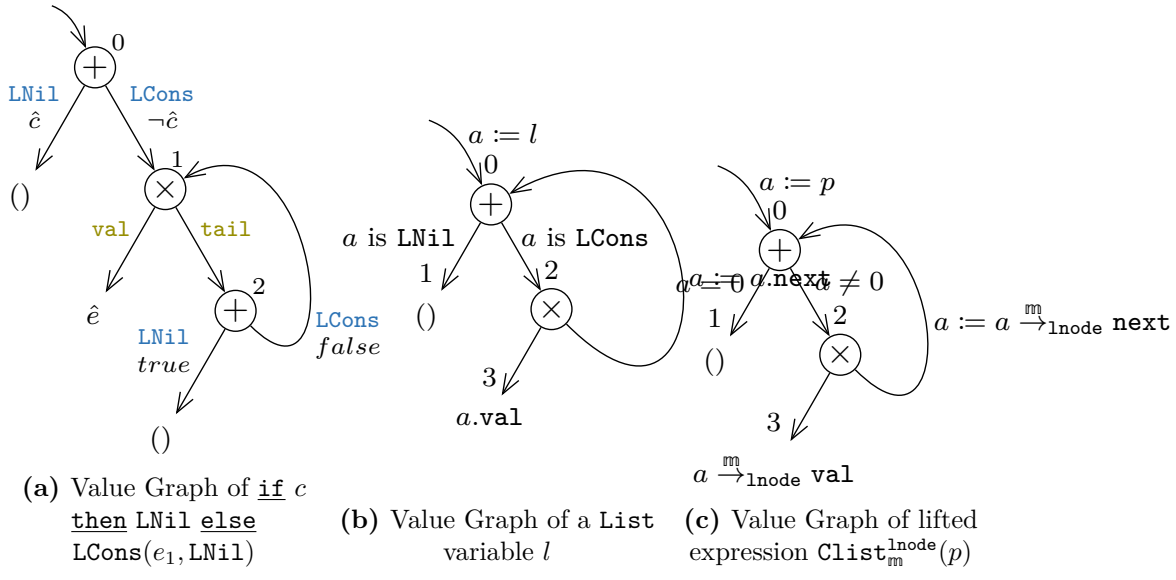
The leaf nodes are the scalar types. For each bound type variable, a backedge is created to the ancestor at which it is bounded.



(a) Canonical `List` ADT **(b)** Peeled `List` ADT **(c)** Unrolled `List` ADT

Figure 15: Three equivalent graphical representations for `List` ADT. Figure 15a shows the graphical representation of the canonical form of `List`. Figure 15b is obtained by peeling the *backedge* [2,1] in fig. 15a. Figure 15c is obtained by unrolling the backedge [2,1] in fig. 15a or by peeling the backedge [4,1] in fig. 15b respectively.

Recall that types in Spec follow equirecursive typing rules i.e. the two types $\mu\alpha.T$ and $T[\mu\alpha.T/\alpha]$ in $\mathbb{T}$ are *equal* types, where $T[\mu\alpha.T/\alpha]$ is defined as the *unfolding* of $\mu\alpha T$. In general, under equirecursive typing, two types are equal iff their infinite expansions (through unfolding) are equal. In the type tree representation, two types are equal iff their infinite expansions are equivalent. Expansion through unfolding is equivalent to *unrolling* one iteration of a cycle in its type tree. Figure 15 shows the three equivalent type trees for the `List` type. In practice, equality of two types in $\mathbb{T}$ can be checked by converting both to their *canonical* forms and checking syntactic equality. Figure 15a shows the canonical type tree for `List`. On the other hand, figs. 15b and 15c are obtained through *peeling* and *unrolling* the only cycle in fig. 15a. Peeling is a form of partial unrolling which only extracts the starting node in the cycle instead of one iteration. Note that the type trees in fig. 15 are technically not 'trees' since they contain cycles. However, all three of them are isomorphic and represents the same infinite tree obtained by unrolling each of them forever.



(a) Value Graph of `if` c `then LNil else` `LCons`$(e_1, \mathtt{LNil})$

(b) Value Graph of a `List` variable l

(c) Value Graph of lifted expression $\mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(p)$

Figure 16: Value Graphs of three `List` expressions.

With type trees out of the way, we are ready to define a value tree. A value tree for an expression of type T is isomorphic to a type tree for T with the following differences:

1. Each leaf node contains a canonical expression of its scalar type.

2. There is a special node called the *entry node* and a special edge from the entry node to the root of the tree, called the *entry edge*.

3. Similar to the CFG representation, each edge is associated with an edge condition and a transfer function as follows: (a) the entry edge has an edge condition *true*. (b) all outgoing edges for a $\otimes$ node has edge condition *true*. (c) all outgoing edges for a $\oplus$ node are mutually exclusive and exhaustive i.e. exactly one of them is *true*.

Figure 16 shows the value trees for three different expressions. Note that all three expressions have `List` type and they are all isomorphic to one of the `List` type trees in fig. 15.

4.4.7 Conversion Algorithm

We give an algorithm for converting an arbitrary expression e to its value tree. Next, we reformulate multiple procedures related to our proof discharge algorithm in terms of the value trees directly.

today: draw the figures for all conversions first then write a first draft

tomorrow: draw the figures for decomposition, the 2 deconstruction programs, their unified forms and points-to analysis on them

then go straight into the conversion algorithm quite easy for most some time on if-then-else some times on the variable + lifting constructor straight into recursive relations decomposition approximations then deconstruction program alternative unification using the example give the same example with points-to analysis on it write the final algorithm DONE (will need like 5-6 days to write all this out with figures) but for now, i can atleast try to do as much as possible so sir can start to review it

5 Evaluation

We have implemented S2C on top of the Counter tool [24]. 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` [20], `Yices2-45e38fc` [21], 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*. We use a value of *five* for η (used by *StrongestInvCover()* during weakening of recursive relation invariants).

S2C requires the user to provide a Spec program S (specification), a C implementation C, and a file that contains their input-output specifications. An equivalence check requires the identification of lifting constructors to relate C values to the ADT values in Spec through recursive relations. Such relations may be required at the entry of both programs (i.e. in the precondition Pre), in the middle of both programs (i.e., in the invariants at intermediate product-CFG nodes), and at the exit of both programs (i.e., in the postcondition $Post$). Pre and $Post$

are user-specified, whereas the inductive invariants are inferred automatically by our algorithm. During invariant inference, S2C derives the candidate lifting constructors from the user-specified *Pre* and *Post*. More sophisticated approaches to finding lifting constructors are left as future work.

5.1 Experiments

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

Table 4: String lifting constructors and their definitions.

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

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 $\texttt{Cstr}_{\mathbb{m}}^{\texttt{u8[]}}$. `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 $\texttt{Cstr}_{\mathbb{m}}^{\texttt{lnode(u8)}}$ and $\texttt{Cstr}_{\mathbb{m}}^{\texttt{clnode(u8)}}$ respectively. These lifting constructors are defined in table 4. $\texttt{Cstr}_{\mathbb{m}}^{\texttt{lnode(u8)}}$ requires a single argument p representing the pointer to the list node. On the other hand, $\texttt{Cstr}_{\mathbb{m}}^{\texttt{clnode(u8)}}$ requires two arguments p and i, where p represents the pointer to the chunked linked list node and i represents the position of the initial character in the chunk.

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

Lifting constructors are named based on the C data layout being lifted and the Spec ADT type of the lifted value. For example, $\texttt{Cstr}^{\texttt{u8[]}}$ 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., `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.

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

(a) Strlen Specification

```
size_t strlen(char* s);

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

(b) Strlen Implementation using Array

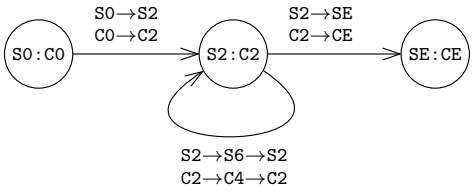
```
typedef struct clnode {
  char chunk[4]; struct clnode* next; } clnode;
size_t strlen(clnode* cl);

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

(c) Optimized Strlen Implementation using Chunked Linked List

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

Table 5: List lifting constructors and their definitions.

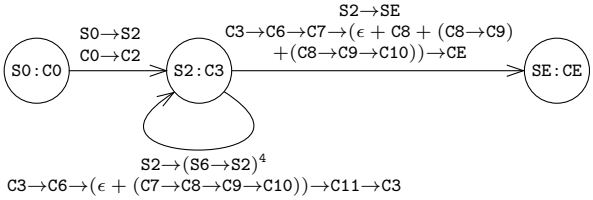
Lifting Constructor	Definition
(T2) `List = LNil | LCons(i32, List)`	
$\texttt{Clist}_m^{\texttt{u32[]}}(p\ i\ n{:}\texttt{i32})$	$\underline{\texttt{if}}\ i \geq_u n\ \ \underline{\texttt{then}}\ \texttt{LNil}$ $\underline{\texttt{else}}\ \texttt{LCons}(p[i]_m^{\texttt{i32}}, \texttt{Clist}_m^{\texttt{u32[]}}(p,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



(a) Product CFG for programs figs. 17a and 17b

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

(b) Invariants Table for fig. 18a



(c) Product CFG for programs figs. 17a and 17c

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

(d) Invariants Table fig. 18c

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

5.1.2 List

We wrote a Spec program specification that creates a list, a program that traverses a list to compute the sum of its elements and a program that computes the dot product of two lists. We use three different data layouts for a list in C: array ($\mathtt{Clist}_{\mathtt{m}}^{\mathtt{u32[]}}$), linked list ($\mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode(u32)}}$), and a chunked linked list ($\mathtt{Clist}_{\mathtt{m}}^{\mathtt{clnode(u32)}}$). The lifting constructors are shown in table 5. Although similar to the String lifting constructors, these lifting constructors differ widely in their data encoding. For example, $\mathtt{Clist}_{\mathtt{m}}^{\mathtt{u32[]}}(p, i, n)$ 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. ($\mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode(u32)}}$) and ($\mathtt{Clist}_{\mathtt{m}}^{\mathtt{clnode(u32)}}$), 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 6: Tree lifting constructors and their definitions.

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

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 ($\texttt{Ctree}_{\mathtt{m}}^{\mathtt{u32[]}}$), and (2) a linked tree node with two pointers for the left and right children ($\texttt{Ctree}_{\mathtt{m}}^{\mathtt{tnode(u32)}}$) (shown in table 6). Both Spec and C programs contain non-tail recursive procedure calls for left and right children. S2C is able to correlate these recursive calls using user-provided *Pre* and *Post*. At the entry of the recursive calls, S2C is required to prove that *Pre* holds for the arguments and at the exit of the recursive calls, S2C assumes *Post* on the returned states.

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

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

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 7). The C implementations for a `Matrix` object include (a) a two-dimensional array ($\texttt{Cmat}_{\mathtt{m}}^{\mathtt{u32[][]}}$), (b) a flattened row-major array ($\texttt{Cmat}_{\mathtt{m}}^{\mathtt{u32[r]}}$), (c) a flattened

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

Table 8: 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 8 lists the various C implementations and the time it took to compute equivalence with their specifications. For functions that take two or more data structures as arguments, we show results for different combinations of data layouts for each argument. We also show the minimum under-approximation (d_u) and over-approximation (d_o) depths at which the equivalence proof completed (keeping all other parameters to their default values).

During the verification of `strchr` and `strpbrk` implementations, we identified an interesting subtlety. Since `strchr` and `strpbrk` return null pointers to signify absence of the required character(s) in the input string, we additionally need to model the UB assumption that the zero address does not belong to the null character terminated array representing the string. We use an explicit constructor `SInvalid` to expose this well-formedness property in a Spec `String`. Furthermore, we relate `SInvalid` to the condition of C character pointer being null using the lifting constructors $\texttt{Cstr}^{\texttt{T}}_{\mathbb{m}}(p\,\texttt{:i32}, \dots)$ (as defined in table 5). These lifting constructors are used as part of Pre to equate S and C input strings. Finally in S, we model the absence of `SInvalid` in the input string as a UB assumption using the `assuming-do` statement introduced in section 2.1. Due to the ($\mathcal{S}$ `def`) assumption, this constraints the inputs to S as well as C to well-formed strings only. This is an example where ($\mathcal{S}$ `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)

5.3 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 [24] 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.

6 Conclusion

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

Prior work on push-button verification of specific systems [16, 38, 36, 37] involves a combination of careful system design and automatic verification tools like SMT solvers. Constrained Horn Clause (CHC) Solvers [19] 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.

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] **Balakrishnan, G.** and **T. Reps**, Recency-abstraction for heap-allocated storage. *In Proceedings of the 13th International Conference on Static Analysis*, SAS'06. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3540377565. URL https://doi.org/10.1007/11823230_15.

[12] **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.

[13] **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.

[14] **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.

[15] **Chase, D. R.**, **M. Wegman**, and **F. K. Zadeck**, Analysis of pointers and structures. *In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90. Association for Computing Machinery, New York, NY, USA, 1990. ISBN 0897913647. URL https://doi.org/10.1145/93542.93585.

[16] **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.

[17] **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.

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

[19] **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.

[20] **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.

[21] **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.

[22] **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.

[23] **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.

[24] **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.

[25] **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.

[26] **Jones, N. D.** and **S. S. Muchnick**, A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82. Association for Computing Machinery, New York, NY, USA, 1982. ISBN 0897910656. URL https://doi.org/10.1145/582153.582161.

[27] **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.

[28] **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.

[29] **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.

[30] **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.

[31] **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.

[32] **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`.

[33] **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.

[34] **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`.

[35] **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`.

[36] **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`.

[37] **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`.

[38] **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`.

[39] **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.

[40] **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.

[41] **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.

[42] **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.

[43] **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.

[44] **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/.

[45] **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.

[46] **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.

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

[48] **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`.