Synopsis of Thesis on

COUNTEREXAMPLE GUIDED EQUIVALENCE CHECKING BETWEEN PROGRAM SPECIFICATION USING SAFE ALGEBRAIC DATA TYPES AND ITS C IMPLEMENTATIONS

by

Indrajit Banerjee
(2020CSY7569)

Under the guidance of

Prof. Sorav Bansal
(Computer Science and Engineering)

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

Master of Science (Research)

to the

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

June 2023

Abstract

We describe an algorithm capable of checking equivalence of two programs that manipulate recursive data structures such as linked lists, strings, trees and matrices. The first program, called specification, is written in a succint 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 bisimulation relation between the two programs. Our primary contribution is an algorithm to discharge proof obligations containing relations between recursive data structure values across the two diverse syntaxes. Our proof discharge algorithm is capable of generating falsifying counterexamples in case of a proof failure. As part of our proof discharge algorithm, we formulate an elegent program representation of ADT values. This allows us to reformulate proof obligations due to the top-level equivalence check into smaller nested equivalence checks. We evaluate our equivalence checker on implementations of common string library functions taken from popular C library implementations, as well as, implementations of common list, tree and matrix operations. These implementations differ in data layout of recursive data structures as well as algorithmic strategies. We demonstrate that our equivalence checker, based on this algorithm, is able to compute equivalence between an abstract specification program and all of its diverse C implementations.

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

Contents

1 Introduction

Recursive data structures like lists, strings, and trees are the building blocks of many algorithms and software systems. In languages like C, pointer and array based implementations of these data-structures are prone to safety and liveness bugs. Similar recursive data structures are also available in safer functional languages like Haskell, where algebraic data types (ADTs) [13] ensure several safety properties.

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

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

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

(a) Spec Program

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

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

(b) C Program with `malloc()`

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

The programs in figs. 1a and 1b construct lists in a functional language and in C respectively. In prior work on formally-verified systems (e.g., seL4 [25]), researchers have employed interactive proof assistants to prove that a C implementation is observably equivalent to a higher-level functional implementation. Unfortunately, this method of manually codifying equivalence proofs through an interactive theorem prover requires expertise and is laborious.

We present S2C, an algorithm to automatically search for a proof of equivalence between a functional specification of a recursive data-structure program and its optimized C implementation. To support this, we define a minimal functional language, called Spec, that enables the safe and succinct specification of programs manipulating and traversing recursive data structures. Our proof-search algorithm automatically (push-button) searches for a bisimulation relation between data-structure manipulation programs written in Spec and C. The large semantic gap between the two syntaxes make such automatic proofs particularly interesting: for the same Spec specification, there exist multiple C implementations that may differ in their memory layout and iteration logic; yet, S2C can compare equivalence for all such program pairs automatically.

```
S0: List mk_list (i32 n) {
S1:   List l := LNil;
S2:   i32  i := 0_i32;
S3:   while ¬(i ≥_u n):
S4:     l := LCons(i, l);
S5:     i := i + 1_i32;
S6:   return l;
SE: }
```

(a) (Abstracted) Spec IR

```
C0: i32 mk_list (i32 n) {
C1:   i32 l := 0_i32;
C2:   i32 i := 0_i32;
C3:   while i <_u n:
C4:     i32 p := malloc_C4(sizeof lnode);
C5:     m := m[&p ->^m_lnode val ← i]_i32;
C6:     m := m[&p ->^m_lnode next ← l]_i32;
C7:     l := p;
C8:     i := i + 1_i32;
C9:   return l;
CE: }
```

(b) (Abstracted) C IR

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

Such equivalence proofs require the inference of relations between data-structure values at correlated intermediate program points of both programs. For example, if we correlate PC `S3` of the Spec IR program in fig. 2a with PC `C3` of the C IR program in fig. 2b, we need to infer that the contents of the entire linked list starting at variable $\mathtt{l}_C$ in the C program are equal to the contents of the `List` value $\mathtt{l}_S$ in the Spec program. (Throughout the paper, we use subscripts S and C to represent values of the Spec and C programs respectively). We call such relations that relate recursive data structure values, *recursive relations*. The automatic inference of invariants with recursive relations relies on the discharge of proof obligations that involve equality of arbitrarily deep data structures. *Our primary contribution* *is a proof discharge algorithm that uses an off-the-shelf SMT solver to tackle proof obligations involving recursive relations in the context of an equivalence check.*

Our algorithm leverages prior work on automatic counterexample-guided search for a bisimulation relation [22]. At every step of this counterexample-guided search for a bisimulation relation, inductive invariants and correlations are proposed which need to be checked using off-the-shelf SMT solvers. Thus, a proof obligation, that may potentially involve a recursive relation, needs to be converted to a form that is amenable for reasoning through an SMT solver. Further, if an SMT proof query is determined to be *not provable*, we expect a counterexample; this counterexample represents a potential concrete machine state that may occur in the program (based on our invariant reasoning). These counterexamples help in faster convergence of the invariant inference and correlation algorithms during the automatic construction of a bisimulation relation. This requires the reconstruction of a machine state which may include recursive ADT values from counterexamples returned by the SMT solver. These procedures to convert a proof obligation involving a recursive relation to a sequence of SMT solver queries and the conversion back from the SMT-generated counterexamples to a machine state possibly containing ADT values, are part of our proof discharge algorithm.

We have manually developed a small number of succinct specifications of data-structure programs in Spec involving ADT-based lists, strings, trees, and two-dimensional matrices. Using these, we automatically verify equivalent programs in popular C libraries with strings and common functions operating on lists, trees, matrices. A diverse set of data layouts are considered for each such data structure. For example, a list may be implemented using a flat array, a linked list or even a chunked linked list (each node contains a constant-sized chunk). On the other hand, a matrix might be laid out in a 2-dimensional array, a row major array, a column major array or an array of linked lists etc. For one specification program in Spec, multiple different C implementations are verified.

2 Problem Setting and Equivalence Definition

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

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

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

Similarly, the C implementation is also lowered to a 3AC IR that resembles LLVM IR [29]. The primary differences between a C source and its IR are: (a) the sizes and memory-layouts of both scalar (e.g., `int`) and compound (e.g., `struct`) types are concretized in the IR, and (b) we annotate any `malloc` calls with the IR PC at which that call appears (e.g., $\texttt{malloc}_{\texttt{C4}}$ in fig. 2b).

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

2.1 Equivalence Definition

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

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

The $(S\ \ \texttt{def})$ antecedant states that we are only interested in proving equivalence for well-defined executions of S, i.e., executions that are free of undefined behaviour (UB). For example, division-by-zero is UB in S. Sometimes the user may be interested in constraining the nature of inputs to the C program, e.g., the $\texttt{strlen(char* s}_C\texttt{)}$ function is well-defined only if $\texttt{s}_C$ is not null. Thus, for `strlen`, we are only interested in computing equivalence for non-null input pointers. Spec has no notion of pointers and so this condition cannot be encoded in S alone. In these cases, we use a combination of Pre and $(S\ \ \texttt{def})$ to constrain the executions of C for which we are interested in proving equivalence. In the `strlen` example, $(S\ \ \texttt{def})$ is encoded as an abstract condition that the input string $\texttt{s}_S$ is "not invalid", written $\neg(\texttt{s}_S\ \ \textit{\texttt{is}}\ \texttt{SInvalid})$, where `SInvalid` is a constructor for the Spec `String` type. The precondition Pre then contains the relation $(\texttt{s}_S\ \ \textit{\texttt{is}}\ \texttt{SInvalid}) \Leftrightarrow (\texttt{s}_C\texttt{=0})$. This ensures that we compute equivalence only for those executions of C

[1] Input_C and Output_C include the initial and final memory state of C respectively.

where the input pointer $\mathtt{s}_C$ is non-null. The use of an explicit constructor for expressing ill-formedness of Spec input values along with a cleverly chosen *Pre* allows us to constrain the executions to *Spec* and C to well-formed inputs only during equivalence check. Please refer to `Chapter XXX` of thesis for a more detailed explaination of this strategy for the function `strchr`.

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

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

3 Algorithm through Linked List Examples

A `List` ADT in the Spec program is defined at line `A0` in fig. 1a. An empty list is represented by the constant `LNil()`[2]; a non-empty list uses the `LCons` constructor to combine its first value (`val:i32`) and the remaining list (`tail:List`). Spec supports `i<N>` (bitvectors of length `N`), `bool`, and `unit` types, also called *scalar types.* Spec's type system prevents the creation of cycles in ADT values. If `l` is an object of type `List`, then to access its constituent values, we may expand (or unroll) `l` to

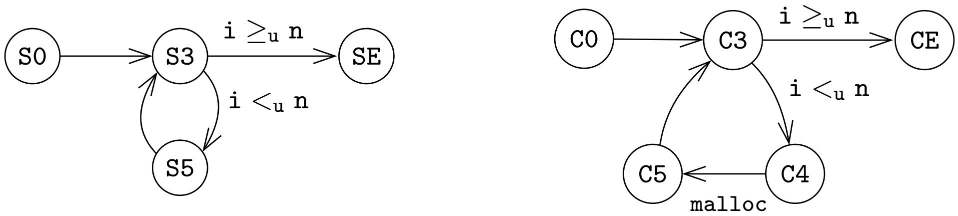
$$U_S : \mathtt{l} \ = \ \underline{\mathtt{if}}\ \mathtt{l}\ \mathit{is}\ \mathtt{LNil}\ \underline{\mathtt{then}}\ \mathtt{LNil}\ \underline{\mathtt{else}}\ \mathtt{LCons}(\mathtt{l.val}, \mathtt{l.tail}) \qquad (1)$$

In this expanded representation of `l`, the *sum-deconstruction* operator[3] '<u>if</u>-<u>then</u>-<u>else</u>' deconstructs a sum type where the <u>if</u> condition '`l` *is* `Constructor`'

[2] `LNil()` represents the application of the nullary constructor `LNil` on the unit value `()`. For brevity, we will simply write `LNil` for `LNil()` henceforth.

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

checks whether the top level constructor of `l` is '`Constructor`'. If `l` is a non-empty list constructed through `LCons`, then `l.val` and `l.tail` are used to access `l`'s first value and `l`'s tail respectively. The right-hand side of eq. (1) can also be viewed as an executable program that unrolls the input `List` object `l` once and outputs a `List` object constructed from `l`'s constituents — we call eq. (1) the *unrolling procedure* U_S of the `List` ADT. We can similarly define the unrolling procedure for any ADT variable.



(a) CFG of Spec Program

(b) CFG of C Program

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

Figures 3a and 3b show the Control-Flow Graph (CFG) representations of the Spec and C programs in figs. 2a and 2b respectively. The CFG nodes represent PC locations of the program, and edges represent transitions through instruction execution. For brevity, we sometimes represent multiple program instructions with a single edge, e.g., in fig. 3b, the edge `C5`→`C3` represents the path `C5`→`C6`→`C7`→`C8`→`C3`. A control-flow edge is associated with an *edge condition* (the condition under which that edge is taken), a *transfer function* (how the program state is mutated if that edge is taken), and a *UB assumption* (what condition should be true for the program execution to be well-defined across this edge). For example, the UB assumption associated with a division instruction in S will encode that the divisor must be non-zero.

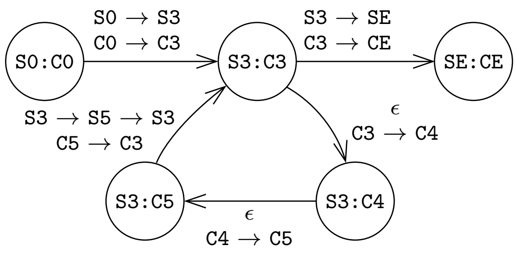


Figure 4: Product-CFG between CFGs in figs. 3a and 3b

3.1 Product CFG

We construct a *bisimulation relation* to identify equivalence between two programs. A bisimulation relation correlates the transitions of S and C in lockstep, such that this lockstep execution ensures identical observable behavior. An equivalence proof through bisimulation construction can be represented using a *product program* [44] and the CFG of a product program is called a *product-CFG*. Figure 4 shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in fig. 3a and fig. 3b.

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

An edge in the product-CFG is formed by pairing a path (a sequence of edges) in S with a path in C. A product-CFG edge encodes the lockstep execution of correlated transitions (or paths). For example, the product-CFG edge `(S3:C5)`$\rightarrow$`(S3:C3)` is formed by pairing the `(S3`$\rightarrow$`S5`$\rightarrow$`S3)` and `C5`$\rightarrow$`C3` in figs. 3a and 3b, and represents that when S makes a transition `(S3`$\rightarrow$`S5`$\rightarrow$`S3)`, then C makes the transition `C5`$\rightarrow$`C3` in lockstep. The edge `(S3:C3)`$\rightarrow$`(S3:C4)` correlates the ϵ path (no transition) in S with `C3`$\rightarrow$`C4` in C. In general, a product-CFG edge e may correlate a finite path ρ_S in S with a finite path ρ_C in C, written $e = (\rho_S, \rho_C)$.

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) $\mathtt{i_S} \leq_{\mathtt{u}} \mathtt{n_S}$	(I4) $\mathtt{l_S} \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l_C})$
(S3 : C4) (S3 : C5)	(I5) $\mathtt{n_S} = \mathtt{n_C}$	(I6) $\mathtt{i_S} = \mathtt{i_C}$	(I7) $\mathtt{i_S} <_{\mathtt{u}} \mathtt{n_S}$	(I8) $\mathtt{l_S} \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l_C})$
(SE : CE)	(E) $\mathtt{ret_S} \sim \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{ret_C})$			

Figure 5: Node Invariants for Product-CFG in fig. 4

At the start node `(S0:C0)` of the product-CFG, the precondition *Pre* (labeled (P)) ensures the equality of input arguments $\mathtt{n_S}$ and $\mathtt{n_C}$ at programs' entry. Inductive invariants are inferred at each product-CFG node that relate the variables of S with variables and memory locations of C. The inductive invariants are identified by running an invariant inference algorithm on the product-CFG, which is further discussed in section 4.3. The inductive invariants for our example are shown in fig. 5. For example, at node `(S3:C5)` in fig. 4, $\mathtt{i}_S = \mathtt{i}_C$ is an inductive invariant. If the inferred invariants ensure that the postcondition *Post* holds at the exit node `(SE:CE)` (labeled E), we have shown equivalence of both programs.

3.2 Recursive relations

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

In fig. 5, the relation between programs' variables at product-CFG nodes `S3:C3`, `S3:C4` and `S3:C5` is encoded as a recursive relation: "$\mathtt{l}_S \sim \mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{l}_C)$" where $\mathtt{l}_S$ and $\mathtt{l}_C$ represent the `l` variables in the Spec and C programs respectively, `lnode` represents the C `struct` type that contains the `val` and `next` fields, and m represents a byte-addressable array representing the current memory state of the C program. $l_1 \sim l_2$ is read l_1 *is recursively equal to* l_2, i.e., l_1 and l_2 are isomorphic and have equal values[4]. The *lifting constructor* $\mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{p})$ is a constructor that *lifts* the C pointer value `p` (pointing to an object of `struct lnode`) and the C memory state m to a Spec `List` value. $\mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{p})$ is defined through its unrolling procedure as:

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

By construction, this unrolling procedure U_C is isomorphic to `List`'s unrolling procedure U_S in eq. (1). "$\mathtt{p} \xrightarrow{m}_s \mathtt{f}$" represents the field '`f`' of the 'struct `s`' object pointed-to by pointer '`p`' in memory state 'm'. When represented in C-like syntax, '$\mathtt{p} \xrightarrow{m}_s \mathtt{f}$' is equivalent to "`*((typeof s.f)*)(&m[p+offsetof(s,f)])`", i.e., the expression '$\mathtt{p} \xrightarrow{m}_s \mathtt{f}$' returns the bytes in the memory array 'm' starting at address '`p+offsetof(s,f)`' and interpreted as an object of type '`typeof s.f`'.

[4] $l_1 \sim l_2$ and $l_1 = l_2$ are equivalent — the former emphasizes the recursive nature of the values being compared.

Note the recursive nature of the lifting value constructor `Clist`: if the pointer `p` of type `i32`[5] is zero (i.e. `p` is a null pointer), then this represents the empty list (`LNil`); otherwise it represents the list formed by `LCons`-ing the value stored at `p->val` in memory m and the list formed by recursively lifting `p->next` using `Clist` in memory m. The recursive lifting constructor `Clist` allows us to compare C values and Spec values for equality. In general, an equality relation between two (possibly recursive) ADT values is called a *recursive relation*. However in the context of bisimulation, we will only consider recursive relations between Spec values (such as variables) and lifted C values (lifted using a lifting constructor such as `Clist`).

We later discuss in section 4.2 how a product-CFG can be constructed automatically through a counterexample-guided search. Before that, we discuss the proof obligations that arise from a given product-CFG. Consider the product-CFG in fig. 4. Assuming that the precondition Ⓟ holds at the entry node `S0:C0` of this product-CFG, a bisimulation check involves checking that the invariants at the other product-CFG nodes hold too, and consequently the postcondition Ⓔ holds at the exit node `SE:CE`. Recall that the precondition Ⓟ and the postcondition Ⓔ are provided by the user, but all the other invariants are inferred automatically.

3.3 Proof Obligations

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

For every product-CFG edge $e = (s \rightarrow d) = (\rho_S, \rho_C)$ in fig. 4, we thus need to prove $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$, where ϕ_s and ϕ_d are the node invariants (shown

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

in fig. 5) at nodes s and d of the product-CFG respectively. The weakest-precondition transformer is used to translate a Hoare triple $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$ to the following first-order logic formula:

$$(\phi_s \wedge pathcond_{\rho_S} \wedge pathcond_{\rho_C} \wedge ubfree_{\rho_S}) \Rightarrow \mathtt{WP}_{\rho_S,\rho_C}(\phi_d) \tag{3}$$

Here, $pathcond_{\rho_X}$ represents the condition that path ρ_X is taken in program X. $ubfree_{\rho_S}$ represents the condition that the execution of program S along path ρ_S is free of undefined behavior. $\mathtt{WP}_{\rho_S,\rho_C}(\phi_d)$ represents the weakest-precondition of the predicate ϕ_d across the product-CFG edge $e = (\rho_S, \rho_C)$. We will use "`LHS`" and "`RHS`" to refer to the left and right hand sides of the implication operator "$\Rightarrow$" in eq. (3).

3.4 Proof Discharge Algorithm and Its Soundness

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

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

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

Our equivalence checking algorithm ensures that, for an invariant $\phi_s = (\phi_{\mathtt{s}}^{1} \wedge \phi_{\mathtt{s}}^{2} \wedge ... \wedge \phi_{\mathtt{s}}^{\mathtt{k}})$, at any node s of a product-CFG, if a recursive relation appears in ϕ_s, it must be one of ϕ_s^1, ϕ_s^2, ..., or ϕ_s^k. We call this the *conjunctive recursive relation* property of an invariant ϕ_s.

A proof obligation $\{\phi_s\}(e)\{\phi_d\}$, where $e = (\rho_S, \rho_C)$, gets lowered using $\mathtt{WP}_e(\phi_d)$ (as shown in eq. (3)) to a first-order logic formula of the following form:

$$(\eta_{1}^{\mathtt{l}} \wedge \eta_{2}^{\mathtt{l}} \wedge ... \wedge \eta_{\mathtt{m}}^{\mathtt{l}}) \Rightarrow (\eta_{1}^{\mathtt{r}} \wedge \eta_{2}^{\mathtt{r}} \wedge ... \wedge \eta_{\mathtt{n}}^{\mathtt{r}}) \tag{4}$$

In this formula, the LHS and RHS are written as conjunctions of η_i^l and η_j^r respectively (for $1 \leq i \leq m$, $1 \leq j \leq n$). Each η_j^r relation is obtained from $\mathtt{WP}_e(\phi_d^j)$, where $\phi_d = (\phi_{\mathtt{d}}^{1} \wedge \phi_{\mathtt{d}}^{2} \wedge ... \wedge \phi_{\mathtt{d}}^{\mathtt{n}})$. Thus, due to the conjunctive recursive relation property of ϕ_s and ϕ_d, any recursive relation in eq. (4) must appear as one of η_i^l or η_j^r.

To simplify proof obligation discharge, we break a first-order logic proof obligation P of the form in eq. (4) into multiple smaller proof obligations of the form $P_j : (\mathtt{LHS} \Rightarrow \eta_{\mathtt{j}}^{\mathtt{r}})$, for $j = 1..n$. Each proof obligation P_j is then discharged separately. We call this conversion from a bigger query to multiple smaller queries, *RHS-breaking*.

3.5 Iterative Unification and Unrolling

We begin with some definitions. An expression e whose top-level constructor is a lifting constructor, e.g., $e = \mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(\mathtt{l_C})$, is called a *lifted expression*. An expression e of the form $\mathtt{v.a}_1\mathtt{.a}_2\mathtt{...a}_n$ i.e. a variable with *zero* or more *product deconstruction* operators applied on it, is called a *pseudo-variable*. By definition, variables are pseudo-variables. An expression e in which (1) all product deconstructors (e.g. '`.tail`') appear as part of a *pseudo-variable* and (2) each *sum-is* operator (e.g. '*is* `LCons`') operate on a *pseudo-variable*, is called a *canonical expression*.

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

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

When we attempt to unify two expressions, we unify the structures created by the value constructors and the 'if-then-else' operator of their canonical forms. The unification procedure either fails to unify, or it returns tuples (p_1, p_2, a_1, e_2) where atom a_1 at expression path condition p_1 in one expression is correlated with expression e_2 at expression path condition p_2 in the other expression.

For two non-atomic expressions e_1 and e_2 to unify successfully, it must be true that either the top-level node in both e_1 and e_2 have the same value constructor (in which case a unification is attempted for each of the children of the top-level constructor), *or* the top-level node in one of e_1 or e_2 is if-then-else. If the top-level node of e_1 is "if (c) then $e_1^{\texttt{then}}$ else $e_1^{\texttt{else}}$, we attempt to unify both $e_1^{\texttt{then}}$ and $e_1^{\texttt{else}}$ with e_2 and return success iff any of these attempts succeed (similarly for e_2). Whenever we descend down an if-then-else operator, we conjunct the corresponding if condition (for $e_1^{\texttt{then}}$) or its negation (for $e_1^{\texttt{else}}$) to the respective expression path condition. If one of e_1 and e_2 (say e_2) is atomic, unification always succeeds and returns (p_2, p_1, e_2, e_1).

With each atom of an ADT type, we associate an unrolling procedure. By definition, an ADT atom is either a pseudo-variable or a lifted expression. Every (pseudo-)variable is associated with its unrolling procedure as governed by its ADT. For example, the unrolling procedure for a Spec variable `l` of `List` type is U_S (eq. (1)). For lifted expressions, the unrolling procedure is given by the definition of the lifting constructor such as U_C (eq. (2)) for the lifting constructor `Clist`.

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

If a proof obligation involves a recursive relation $e_a \sim e_b$, we unify e_a and e_b through a call to $\Theta(e_a, e_b, \texttt{true}, \texttt{true})$. For example, the unification of "<u>`if`</u> `(`$\texttt{c}_1$`)` <u>`then`</u> `LNil` <u>`else`</u> `LCons(0, `$\texttt{l}_S$`)`" and "<u>`if`</u> `(`$\texttt{c}_2$`)` <u>`then`</u> `LNil` <u>`else`</u> `LCons(0, `$\texttt{Clist}_m^{\texttt{lnode}}(\texttt{l}_C)$`)`" yields the correlation tuples: $(\texttt{c}_1, (), \texttt{c}_2, ())$, $(\neg\texttt{c}_1, 0, \neg\texttt{c}_2, 0)$ and $(\neg\texttt{c}_1, \texttt{l}_S, \neg\texttt{c}_2, Clist_m^{\texttt{lnode}}(\texttt{l}_C))$.

If the set of n tuples obtained after a successful unification of $e_a \sim e_b$ are $(p_1^i, a_1^i, p_2^i, a_2^i)$ (for $i = 1 \ldots n$), then $e_a \sim e_b \Leftrightarrow \bigwedge_{i=1}^{n}((p_1^i = p_2^i) \wedge ((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i)))$[6]. We call $\bigwedge_{i=1}^{n}((p_1^i = p_2^i) \wedge ((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i)))$ the *decomposition* of $e_a \sim e_b$. Each conjunctive clause of one of the forms $(p_1^i = p_2^i)$ and $((p_1^i \wedge p_2^i) \rightarrow (a_1^i = a_2^i))$ in this decomposition is called a *decomposition clause*. A decomposition clause may relate only atomic values, i.e., in the decomposed form, all recursive relations relate only ADT variables and/or lifted expressions. The decomposition for a failed unification is defined to be `false`. We *decompose* a recursive relation by replacing it with its decom-

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

position. We *decompose* a proof obligation P by decomposing all recursive relations in P.

3.6 k-unrolling with respect to an unrolling procedure

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

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

3.7 Category 1: The k-unrolling of P does not contain recursive relations

In fig. 4, consider a proof obligation generated across the product-CFG edge $(\texttt{S0} : \texttt{C0}) \rightarrow (\texttt{S3} : \texttt{C3})$ while checking if the (I4) invariant, $l_S \sim Clist_m^{\texttt{lnode}}(l_C)$, holds at `(S3:C3)`: $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0} \rightarrow \texttt{S3}, \texttt{C0} \rightarrow \texttt{C3})\{l_S \sim Clist_m^{\texttt{lnode}}(l_C)\}$. The precondition $\phi_{\texttt{S0:C0}} = (n_S = n_C)$ 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 "$(n_S = n_C) \Rightarrow (\texttt{LNil} \sim \texttt{Clist}_\texttt{m}^\texttt{lnode}(\texttt{0}))$". 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 $(n_S = n_C) \Rightarrow \texttt{LNil} \sim \texttt{LNil}$ which trivially resolves to true.

Consider another example of a proof obligation, $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0} \rightarrow \texttt{S3} \rightarrow \texttt{S5} \rightarrow \texttt{S3}, \texttt{C0} \rightarrow \texttt{C3})\{l_S \sim Clist_m^{\texttt{lnode}}(l_C)\}$. Notice, we have changed the path in S to

$\texttt{S0} \rightarrow \texttt{S3} \rightarrow \texttt{S5} \rightarrow \texttt{S3}$ here. In this case, the corresponding first-order logic condition evaluates to: "$(n_S = n_C) \Rightarrow (\texttt{LCons}(0, \texttt{LNil}) \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(0))$". One-unrolling of this proof obligation converts $\texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(0)$ to LNil, and decomposes RHS into false. The proof obligation is further discharged using an SMT solver which provides a counterexample (model) that evaluates the formula to false. For example, the counterexample $\{$ $\texttt{n}_S \mapsto \texttt{42}$, $\texttt{n}_C \mapsto \texttt{42}$ $\}$ evaluates this formula to false. These counterexamples assist in faster convergence of our invariant inference and correlation search procedures (as we will discuss later in sections 4.2 and 4.3).

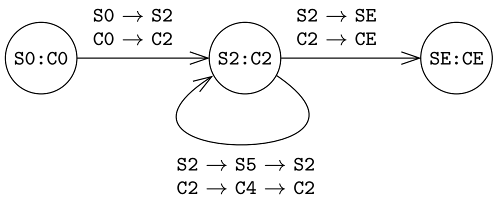
Thus, we unify the structure and values of the Spec objects on both sides of the $\sim$ operator (after k unrollings), and discharge the resulting proof obligations (that relate bitvector and array values) using an SMT solver. Please refer to Chapter XXX of the thesis for the intricacies of (a) translation of the formlua to SMT logic and (b) reconstruction of counterexamples from the models returned by the SMT solver. Assuming a capable enough SMT solver, all proof obligations in Category 1 can be discharged precisely, i.e., we can always decide whether P evaluates to true or false. If it evaluates to false, we also obtain a counterexample.

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

(a) (Abstracted) Spec IR

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

(b) (Abstracted) C IR



(c) Product-CFG

PC-Pair	Invariants
(S0 : C0)	(P) $\mathtt{l_S} \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l_C})$
(S2 : C2)	(I1) $\mathtt{l_S} \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l_C})$ (I2) $\mathtt{sum_S} = \mathtt{sum_C}$
(SE : CE)	(E) $\mathtt{ret_S} = \mathtt{ret_C}$

(d) Node Invariants of the Product-CFG

Figure 6: Spec and C programs for traversing a linked list. Figure 6c shows the Product-CFG between the IRs in figs. 6a and 6b. The inductive invariants of the Product-CFG are given in fig. 6d.

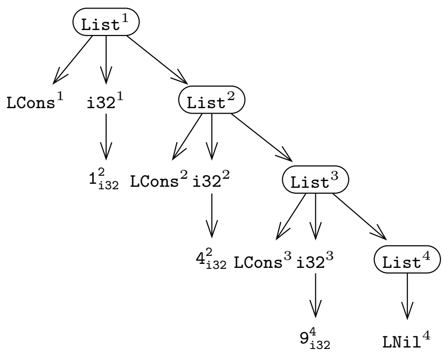
3.8 Category 2: The k-unrolling of P contains a recursive relation in the LHS only

Consider the pair of programs in figs. 6a and 6b that traverse a list to compute the sum of all elements. The corresponding product-CFG and its node invariants that ensure observable equivalence are shown in figs. 6c and 6d.

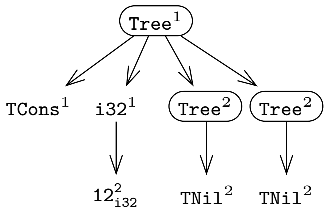
Consider the proof obligation $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2} \to \mathtt{S5} \to \mathtt{S2}, \mathtt{C2} \to \mathtt{C4} \to \mathtt{C2})\{sum_S = sum_C\}$, where the node invariant $\phi_{\mathtt{S2:C2}}$ contains the recursive relation $l_S \sim Clist^{\mathtt{lnode}}_m(l_C)$. The corresponding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

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

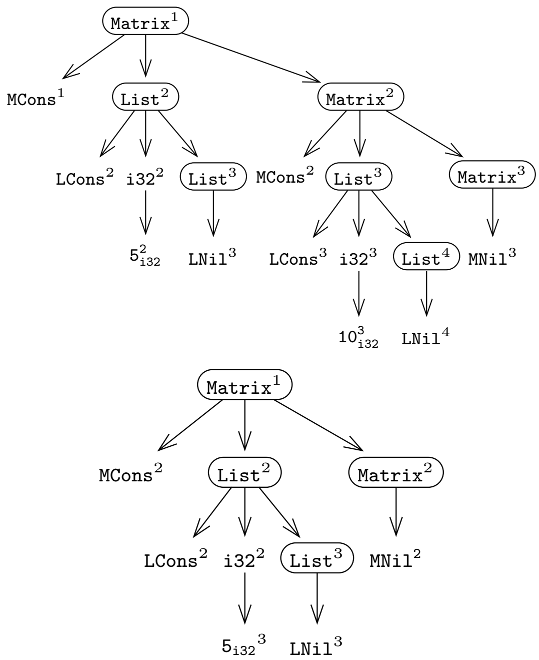
To tackle requirements R1 and R2, our proof discharge algorithm converts the original proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$ into two approximated proof obligations: $(P_{\texttt{pre-o}} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\})$ and $(P_{\texttt{pre-u}} : \{\phi_s^{u_{d_2}}\}(e)\{\phi_d\})$. Here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



(a) `List = LNil | LCons(i32, List)`



(b) `Tree = TNil | TCons(i32, Tree, Tree)`



(c) `Matrix = MNil | MCons(List, Matrix)`

Figure 7: Parsed Trees with depths (shown as superscript) for three values, each of ADT type `List`, `Tree`, and `Matrix` respectively.

3.8.1 Depth of an ADT value

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

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

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

In this context-free grammar interpretation of an ADT, a value of this ADT type can be viewed as a *parse tree* (also called a derivation tree) of the grammar. The *depth* of a node in this parse tree is the number of ADT identifiers (but not scalar type identifiers) in the path from the root node to the node representing that terminal value (both inclusive). Figure 7 shows examples of values of the `List`, `Tree`, and `Matrix` ADTs, and the depth values of the parse tree nodes. The *depth of an ADT value* is the maximum depth of any node in the parse tree of that value.

3.8.2 Overapproximation and Underapproximation

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

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

The d-depth underapproximated version of a recursive relation $l_1 \sim l_2$ is written as $l_1 \approx_d l_2$, where $\approx_d$ represents the condition that the two ADT values l_1 and l_2 are *recursively equal and bounded to depth* d, i.e., l_1, l_2 have a maximum depth $\leq d$ *and* they are recursively equal up to depth d. Thus, $l_1 \approx_d l_2$ is equivalent to $(l_1 \sim_d l_2) \wedge \Gamma_d(l_1) \wedge \Gamma_d(l_2)$, where $\Gamma_d(l)$ represents the condition that the maximum depth of l is d. $l_1 \approx_d l_2$ (for finite d) is a stronger condition than $l_1 \sim l_2$ (underapproximation) as it ensures both equality and max-depth of both values. For arbitrary depths a and b ($a \leq b$), the approximate versions of recursive relation are related as follows:

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

3.8.3 SMT encoding of overapproximate and underapproximate proof obligations

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

- $l_1 \sim_d l_2$ is equivalent to the condition that the parse tree structures of the two values l_1 and l_2 (after d-unrolling) are isomorphic till depth d *and* the corresponding values in both (d-depth) isomorphic structures are also equal. $l_1 \sim_d l_2$ can be identified through d-unrolling followed by removal of all conjunctive clauses containing recursive relations.

  For example, the condition $l \sim_1 \mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{p})$ can be one-unrolled to:

  $$\underline{\text{if}}\ l\ \mathit{is}\ \mathtt{LNil}\ \underline{\text{then}}\ \mathtt{LNil}\ \underline{\text{else}}\ \mathtt{LCons}(l.\mathtt{val}, l.\mathtt{tail})$$
  $$\sim_1$$
  $$\underline{\mathtt{if}}\ (\mathtt{p} == 0)\ \underline{\mathtt{then}}\ \mathtt{LNil}\ \underline{\mathtt{else}}\ \mathtt{LCons}(\mathtt{p} \xrightarrow{m}_{\mathtt{lnode}} \mathtt{val},\ \mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{p} \xrightarrow{m}_{\mathtt{lnode}} \mathtt{next})).$$

  During decomposition, keeping only correlation tuples that equate scalar expressions, the condition above reduces to the SMT-encodable predicate:

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

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

3.8.4 Proof discharge algorithm for Category 2 proof obligations

Thus, for a *Category 2* proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$, we first submit the proof obligation ($P_{\texttt{pre-o}} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\}$) to the SMT solver. Recall that the precondition $\phi_s^{o_{d_1}}$ is the overapproximated version of ϕ_s. If the SMT solver evaluates $P_{\texttt{pre-o}}$ 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 $P_{\texttt{pre-o}}$ to false, then we submit the proof obligation ($P_{\texttt{pre-u}} : \{\phi_s^{u_{d_2}}\}(e)\{\phi_d\}$) to the SMT solver. Recall that the precondition $\phi_s^{u_{d_2}}$ is the underapproximated version of ϕ_s. If the SMT solver evaluates $P_{\texttt{pre-u}}$ 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 $P_{\texttt{pre-u}}$ would also falsify P, and is thus usable in invariant inference and correlation procedures.

Finally, if the SMT solver evaluates $P_{\texttt{pre-u}}$ to true, then we have neither proven nor disproven P. In this case, we imprecisely (but soundly) return false for the original proof obligation P (without a counterexample). Revisiting our examples, the proof obligation $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2} \to \texttt{S5} \to \texttt{S2}, \texttt{C2} \to \texttt{C4} \to \texttt{C2})$ $\{sum_S = sum_C\}$ 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 $\texttt{l}_C$ are equal, and that is sufficient to prove that the new values of sum_S and sum_C are also equal (given that the old values are equal, as encoded in $\phi_{\texttt{S2:C2}}$).

Similarly, the proof obligation $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2} \to \texttt{S5} \to \texttt{S2} \to \texttt{S5} \to \texttt{S2}, \texttt{C2} \to \texttt{C4} \to \texttt{C2})$ $\{sum_S = sum_C\}$ evaluates to false (with a counterexample) 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 l_S and $\texttt{Clist}_m^{\texttt{lnode}}(l_C)$ have exactly two nodes[7], then the sum of the values in the two nodes of l_S is equal to the value stored in the first node in $\texttt{l}_C$. This proof obligation

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

will return a counterexample that maps program variables to their concrete values. We show a possible counterexample to this proof obligation below.

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

This counterexample maps variables to values (e.g., $\mathtt{sum}_C$ maps to an `i32` value `3` and $\mathtt{l}_S$ 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 `value` and `next` fields shown for each object). For all other addresses (except the ones for which an explicit mapping is available), m maps them to the default byte-value `77` (shown as $() \mapsto 77$) in this counterexample.

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

3.9 Category 3: The k-unrolling of P contains a recursive relation in RHS and optionally in LHS

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

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

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

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

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

3.9.1 Replacing Spec recursive values in the RHS with lifted C values

S2C utilizes the $\sim$ relationships in the LHS (antecedant) of "⇒" to rewrite eq. (5) so that the recursive List values in its RHS (conclusion) are replaced with the lifted C values (lifted using the Clist constructor). Thus, we rewrite eq. (5) to:

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

After decomposition and RHS-breaking, eq. (5) reduces to the following smaller proof obligations (showing only the RHS, the LHS is the same as in eq. (5)): (1) $\neg(\texttt{p}_C = 0)$, (2) $\neg(\texttt{p}_C = 0) \rightarrow \texttt{i}_S \texttt{ = } (\texttt{p}_C \xrightarrow{m'}_{\texttt{lnode}} \texttt{value})$, and (3) $\neg(\texttt{p}_C = 0) \rightarrow \texttt{Clist}^{\texttt{lnode}}_{m}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{m'}(\texttt{p}_C \xrightarrow{m'}_{\texttt{lnode}} \texttt{next})$. The first two proof obligations fall in *Category 2* and are discharged through over- and under-approximation schemes (as discussed in section 3.8):

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

2. The second proof obligation with postcondition $(\texttt{i}_S\texttt{=}(\texttt{p}_C \xrightarrow{m'}_{\texttt{lnode}} \texttt{value}))$ also evaluates to true because $\texttt{i}_C$ is written to address $\texttt{\&(p}_C \xrightarrow{m'}_{\texttt{lnode}} \texttt{value)}$ in m' (eq. (6)) and the LHS ensures that $\texttt{i}_S\texttt{=}\texttt{i}_C$.

For ease of exposition, we simplify the postcondition of the third proof obligation from $\neg(\mathtt{p}_C = 0) \rightarrow (\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)\sim \mathtt{Clist}^{\mathtt{lnode}}_{m'}(\mathtt{p}_C \xrightarrow{m'}_{\mathtt{lnode}}\mathtt{next}))$ to $(\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)\sim \mathtt{Clist}^{\mathtt{lnode}}_{m'}(\mathtt{l}_C))$. This simplification is valid because $\mathtt{l}_C$ is written to address $\&(\mathtt{p}_C \xrightarrow{m'}_{\mathtt{lnode}}\mathtt{next})$ in m' (eq. (6)). Also, we have already shown that $\neg(\mathtt{p}_C = 0)$ holds. Thus, the third proof obligation can be rewritten as a recursive relation between two lifted expressions[8]:

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

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

3.9.2 Conversion of recursive equality between lifted expressions to a bisimulation

Consider a program that recursively calls the unrolling procedure in eq. (2) to deconstruct $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$. For example, $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$ may yield a recursive call to the unrolling procedure $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_\mathtt{C} \xrightarrow{m}_{\mathtt{lnode}} \mathtt{next})$ and so on, until the argument to the unrolling procedure becomes zero. This program essentially deconstructs $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$ into its terminal (scalar) values and reconstructs a `List` value equal to the value represented by $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$. We call this program a *reconstruction program* based on the unrolling procedure of $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$.

Theorem 1. *Under the antecedant* $(\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C))$*:*

$(\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C) \sim \mathtt{Clist}^{\mathtt{lnode}}_{m'}(\mathtt{l}_C))$ *holds iff a bisimulation relation exists between the reconstruction programs based on* $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$ *and* $\mathtt{Clist}^{\mathtt{lnode}}_{m'}(\mathtt{l}_C)$.

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

The bisimulation relation must ensure that the observables generated by both procedures are identical.

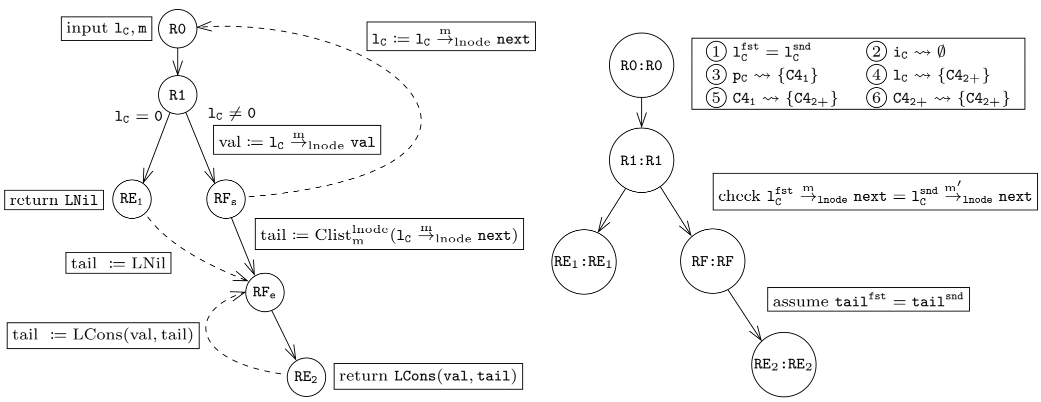
Proof. The "if" case of this "iff" relation follows from noting that the observables of a reconstruction program are the generated `List` values. Thus, a successful bisimulation check ensures equal `List` values upon termination. Termination follows from the antecedant because Spec values (such as l_S) must be finite.

The "only if" case follows from the unification of the unrolling procedure (in eq. (2)) for $\mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{l}_C)$ and $\mathtt{Clist}_{m'}^{\mathtt{lnode}}(\mathtt{l}_C)$.

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

3.9.3 Checking bisimulation between reconstruction programs

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



(a) Reconstruction Program **(b)** Recons-PCFG

Figure 8: The reconstruction program for $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$ and recons-PCFG between reconstruction programs of $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$ and $\mathtt{Clist}^{\mathtt{lnode}}_{m'}(\mathtt{l}_C)$. In fig. 8a, `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. 8b, the square box to the right of node `D0:D0` contains the inferred invariants for this recons-PCFG.

The reconstruction program and the recons-PCFG for our `Clist` example are shown in fig. 8. To check bisimulation between the programs that deconstruct $\mathtt{Clist}^{\mathtt{lnode}}_{m}(\mathtt{l}_C)$ and $\mathtt{Clist}^{\mathtt{lnode}}_{m'}(\mathtt{l}_C)$, the recons-PCFG correlates one unrolling of the first program with one unrolling of the second program. An unrolling of each reconstruction program is based on the unrolling procedure in eq. (2). Thus, the PC-transition correlations of both programs are trivially obtained by unifying the unrolling procedure with itself. A node is created in the recons-PCFG that encodes the correlation of the entries of the unrolling procedure in both programs, we call this node the *recursive-node* in the recons-PCFG, e.g., the recursive node in fig. 8b is `R0:R0`. A recursive call becomes a back-edge in the recons-PCFG that terminates at the recursive-node. A candidate invariant at the recursive-node is obtained by equating the pair of corresponding $\mathtt{l}_C$ variables across the first and second programs, i.e., $\mathtt{l}_C^{fst} = \mathtt{l}_C^{snd}$. At the start of both reconstruction programs, $\mathtt{l}_C^{fst} = \mathtt{l}_C^{snd} = \mathtt{l}_C^{start}$ — the same $\mathtt{l}_C^{start}$ is passed to both reconstruction programs, only the memory states m and m' are different. The bisimulation check thus involves checking that if the invariant $\mathtt{l}_C^{fst} = \mathtt{l}_C^{snd}$ holds at the recursive-node, then during one iteration of the unrolling procedure in both programs:

1. The <u>if</u> condition ($\texttt{l}_C^{fst} = 0$) in the first program is equal to the corresponding <u>if</u> condition ($\texttt{l}_C^{snd} = 0$) in the second program.

2. If the <u>if</u> condition evaluates to false in both programs, then the observable values (that are used in the construction of the list) are equal, i.e., $((\texttt{l}_C^{fst} \neq 0) \wedge (\texttt{l}_C^{snd} \neq 0)) \Rightarrow (\texttt{l}_\texttt{C}^\texttt{fst} \xrightarrow{m}_{\texttt{lnode}} \texttt{val} = \texttt{l}_\texttt{C}^\texttt{snd} \xrightarrow{m'}_{\texttt{lnode}} \texttt{val})$.

3. If the <u>if</u> condition evaluates to false in both programs, then the invariant holds at the beginning of the unrolling procedure invoked through the recursive call. This involves checking equality of the arguments to the recursive call, i.e., $((\texttt{l}_C^{fst} \neq 0) \wedge (\texttt{l}_C^{snd} \neq 0)) \Rightarrow \texttt{l}_\texttt{C}^\texttt{fst} \xrightarrow{m}_{\texttt{lnode}} \texttt{next} = \texttt{l}_\texttt{C}^\texttt{snd} \xrightarrow{m'}_{\texttt{lnode}} \texttt{next}$.

The first check succeeds due to the invariant $\texttt{l}_C^{fst} = \texttt{l}_C^{snd}$. For the second and third checks, we additionally need to reason that the memory objects $\texttt{l}_\texttt{C}^\texttt{fst} \xrightarrow{m}_{\texttt{lnode}} \texttt{val}$ and $\texttt{l}_\texttt{C}^\texttt{fst} \xrightarrow{m}_{\texttt{lnode}} \texttt{next}$ cannot alias with the writes (in m' in eq. (6)) to the newly allocated objects $\texttt{p}_C \xrightarrow{m}_{\texttt{lnode}} \texttt{val}$ and $\texttt{p}_C \xrightarrow{m}_{\texttt{lnode}} \texttt{next}$. This aliasing information is captured using a points-to analysis, described next in section 3.9.4.

Notice that a bisimulation check between the reconstruction programs is significantly easier than the top-level bisimulation check between Spec and C programs: here, the correlation of PC transitions is trivially identified by unifying the unrolling procedure with itself, and the candidate invariants are obtained by equating each corresponding pair of variables across the two programs.

3.9.4 Points-to Analysis

To reason about aliasing (as required during the bisimulation check in section 3.9.3), we conservatively compute the *may-point-to* information for each program value using Andersen's algorithm [10]. The range of this computed may-point-to function are *sets of region labels*, where each region label identifies a set of memory objects. The sets of memory objects identified by two distinct region labels are necessarily disjoint. We write $p \rightsquigarrow \{R_1, R_2\}$ to represent the condition that value p may point to an object belonging to one of the region labels R_1 or R_2 (but may not point to any object outside of R_1 and R_2).

We populate the set of all region labels using the *allocation sites* of the program, i.e., PCs where a call to `malloc` exists, e.g., `C4` in fig. 2b is an allocation site. For each allocation site A, we create two region labels: (1) the first region label, called A_1, identifies the set of memory objects that were allocated by the most recent execution of A. (2) The second region label, called A_{2+}, identifies the set of memory objects that were allocated by older (not the most recent) executions of A.

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

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

3.9.5 Transferring points-to information to the recons-PCFG

Recall that in section 3.9.2, we reduce a validity check of the condition $\mathtt{Clist}_m^{\mathtt{lnode}}(\mathtt{l}_C) \sim \mathtt{Clist}_{m'}^{\mathtt{lnode}}(\mathtt{l}_C)$ to a bisimulation check. Also, recall that we discharge the bisimulation check through the construction of a recons-PCFG that compares the unrolling procedure with itself (executing on memory states m and m'). During this bisimulation check, we need to prove that for each execution of the unrolling procedure, $\mathtt{l_C} \xrightarrow{m}_{\mathtt{lnode}} \{\mathtt{val}, \mathtt{next}\}$

and $\mathtt{l_C} \xrightarrow{m'}_{\mathtt{lnode}} \{\mathtt{val}, \mathtt{next}\}$[9] are equal. To successfully discharge these proof obligations, it suffices to show $\mathtt{l}_C$ cannot alias with the memory writes that distinguish m from m'.

Our points-to analysis on the C program determines that at PC `C5` (the start of the product-CFG edge `(S3:C5)`→`(S3:C3)` across which the proof condition is being evaluated), the pointer to the *head* of the list, i.e., $\mathtt{l}_C^{start}$ points to $\mathtt{C4}_{2+}$. It also determines that the distinguishing writes modify memory regions belonging to $\mathtt{C4}_1$. Further, we get $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}\}$ at PC `C5`. However, notice that these determinations only rule out aliasing of the list-head with the distinguishing writes. We also need to confirm non-aliasing of the internal nodes of the linked list with the distinguishing writes. For this, we need to identify a points-to invariant, $\mathtt{l}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$, at the recursive-node of the recons-PCFG (shown in fig. 8b). To see why $\mathtt{l}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$ is an inductive invariant at the recursive-node:

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

To identify this points-to invariant, we run our points-to analysis (the same analysis that is run on the C program) on the reconstruction programs (fig. 8a) before comparing them for equivalence. The boundary condition for the points-to analysis at the entry node of the reconstruction program (e.g., `R0` in fig. 8) is based on the results of the points-to analysis on C at the PC where the proof obligation is being discharged (e.g., `C5` in our fig. 1b). The points-to invariants at a node $(\mathtt{R}_i^{fst}, \mathtt{R}_j^{snd})$ of a recons-PCFG are derived from the results of the points-to analysis on the individual reconstruction programs at nodes $\mathtt{R}_i^{fst}$ and $\mathtt{R}_j^{snd}$ respectively.

During proof obligation discharge (e.g., during the bisimulation check on recons-PCFG), the points-to invariants are encoded as SMT constraints.

[9]Here, we use the symbol $\mathtt{l}_C$ to refer to equal values $\mathtt{l}_C^{fst}$ and $\mathtt{l}_C^{snd}$.

This allows us to successfully complete the bisimulation proof on the recons-PCFG, and consequently successfully discharge the proof obligation $\{\phi_{\mathtt{S3:C5}}\}$ $(\mathtt{S3} \rightarrow \mathtt{S5} \rightarrow \mathtt{S3}, \mathtt{C5} \rightarrow \mathtt{C3})\{l_S \sim Clist_m^{\mathtt{lnode}}(l_C)\}$ in fig. 5. The points-to analysis is described more formally in section 4.5.

3.9.6 Proof discharge algorithm for Category 3 obligations

Before the start of an equivalence check, a points-to analysis is run on the C IR once. During the equivalence check, to discharge a Category 3 proof obligation $P : \mathtt{LHS} \Rightarrow \mathtt{RHS}$ (expressed in first-order logic), we first replace the recursive values of program S in the `RHS` with lifted C values, based on the equalities present in the `LHS`, to obtain P_2. This is followed by decomposition and RHS-breaking of P_2.

Upon successful decomposition, we obtain several smaller proof obligations. To prove P, we require all these smaller proof obligations to be provable. If any of these smaller proof obligations is not provable, we are unable to prove P. If we obtain a counterexample to any of these smaller proof obligations, then that counterexample also falsifies P. Let P_3 represent any such smaller proof obligation. `RHS` of P_3, being a decomposition clause, must relate atomic expressions on the `RHS`. If P_3 relates two scalar values in the `RHS`, then it is a Category 2 proof obligation and can be discharged using the algorithm in section 3.8.4.

If P_3 relates two lifted expressions in the `RHS`, we check if the reconstruction programs of the two lifted ADT values being compared can be proven to be bisimilar (assuming that `LHS` of P_3 holds at the correlated entry nodes in the recons-PCFG). To improve the bisimulation check's precision, we transfer the points-to information of the C program (at the PC where the proof obligation is being discharged) to the entry of the reconstruction programs. The same points-to analysis is ran on the reconstruction programs to populate the points-to function at all PCs.

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

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

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

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

4 Formalism

4.1 The Spec Language

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

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

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

The language also borrows its expression grammar heavily from functional languages. This includes the usual constructs like `let-in`, `if-then-else`, function application and the `match` statement for pattern-matching (i.e. deconstructing) sum and product values. Unlike functional languages, Spec only supports first order functions. Also, Spec does not support partial function application. Hence, we constrain our attention to C programs containing only first order functions. Spec is equipped with a special `assuming-do` construct for explicitly providing UB conditions. These assumptions become part of (S `def`) as discussed in section 2.1. Spec also provides the typical boolean and bitvector operators for expressing computation in C succintly yet explicitly. This includes logical operators (e.g., `and`), bitvector arithmatic operators (e.g., `bvadd(+)`) and relational operators for comparing bitvectors interpreted as signed or unsigned integers (e.g., $\leq_{u,s}$).

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

Figure 9: Simplified expression grammar of Spec language

4.2 Counterexample-Guided Best-First Search Algorithm for a Product-CFG

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

Anchor nodes in the CFG of the C program are identified to ensure that every cycle in the CFG contains at least one anchor node. Also, for every procedure call in the CFG, anchor nodes are created just before and just after the callsite, e.g., in fig. 3b, `C4` and `C5` are anchor nodes around the call to `malloc()`. Our algorithm ensures that for each anchor node in C, we identify a correlated node in S — if a product-CFG π contains a product-CFG node (n_S, n_C), then π correlates node n_C in C with node n_S in S. The first partially-constructed product-CFG contains a single entry node that encodes the correlation of the entry nodes (`S0:C0`) of the two input CFGs.

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

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

To evaluate the promise of a possible correlation (ρ_S, ρ_C) starting at node s in product-CFG π, we examine the execution behavior of the counterexamples at s on the product-CFG edge $e = (s \to d) = (\rho_S, \rho_C)$. If the counterexamples ensure that the machine states remain related at d, then that candidate correlation is ranked higher. This ranking criterion is based on prior work [22]. A best-first search (BFS) procedure based on this ranking criterion is used to incrementally construct a product-CFG that proves bisimulation. For each intermediate candidate product-CFG π generated during this search procedure, an automatic invariant inference procedure is used to identify invariants at all the nodes in π. The counterexamples obtained from the proof obligations created by this invariant inference procedure are added to the respective nodes in π; these counterexamples help rank future correlations starting at those nodes.

If after invariant inference, we realize that an intermediate candidate product-CFG π_1 is not promising enough, we backtrack and choose another candidate product-CFG π_2 and explore the potential correlations that can be added to π_2. Thus, a product-CFG is constructed one edge at a time. If at any stage, the invariants inferred for a product-CFG π_i ensure equal observables, we have successfully demonstrated equivalence.

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

Table 1: Dataflow formulation for the Invariant Inference Algorithm.

Domain	$\left\{ \begin{array}{c} \phi_n \text{ is a conjunction of predicates drawn from} \\ \text{grammar in 10b, } \Gamma_n \text{ is a set of counterexamples} \end{array} \right\}$
Direction	Forward
Transfer function across edge $e = (s \to d)$	$(\phi_d, \Gamma_d) = f_e(\phi_s, \Gamma_s)$ (fig. 10a)
Meet operator $\otimes$ $(\phi_n, \Gamma_n) \leftarrow (\phi_n^1, \Gamma_n^1) \otimes (\phi_n^2, \Gamma_n^2)$	$\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2, \qquad \phi_n \leftarrow \mathit{StrongestInvCover}(\Gamma_n)$
Boundary condition	$\texttt{out}[n^{start}] = (Pre, \Gamma_{n^{start}})$
Initialization to $\top$	$\texttt{in}[n]$ = $(\texttt{False}, \{\})$ for all non-start nodes

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

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

$$\begin{aligned} Inv \;\rightarrow\; & \textstyle\sum_i c_i v_i = c \mid v_1 \odot v_2 \\ \mid\; & \alpha_S = \texttt{liftC}_m(v^C \ldots) \end{aligned}$$

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

Figure 10: Transfer function f_e and Predicate grammar Inv for invariant inference dataflow analysis in table 1. Given invariants (ϕ_s) and counterexamples (Γ_s) at node s, f_e returns the updated invariants (ϕ_d) and counterexamples (Γ_d) at node d. *StrongestInvCover*(Γ) computes the strongest invariant cover for counterexamples Γ. $\texttt{exec}_e(\Gamma)$ (concretely) executes counterexamples Γ over edge e. $\texttt{SAT}(\phi,\ \gamma)$ determines the satisfiability of ϕ; if satisfiable, the models (counterexamples) are returned in output parameter γ.

4.3 Invariant Inference and Counterexample Generation

Table 1 presents our dataflow analysis for inferring invariants ϕ_n at each node n of a product-CFG, while also generating a set of counterexamples Γ_n at node n that represents the potential concrete machine states at n.

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

The predicate grammar allows the automatic inference of affine and inequality relations between bitvector values of both programs, as well as, recursive relations between an ADT value in Spec (α_S) and a *lifted* ADT value from C ($\texttt{liftC}_m(p_C)$). We enumerate these recursive relation guesses for all bitvector variables v^C in C and candidate `liftC` lifting constructor. In our implementation, the candidate `liftC` constructors are derived from the constructors present in the precondition *Pre* and the postcondition *Post*. More sophisticated strategies for automatic guessing of these lifting constructors are possible.

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

4.4 Modeling Procedure Calls

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

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

4.5 Points-to Analysis

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

Let S_C be the set of all scalar pseudo-registers in C's IR. We use a forward dataflow analysis to identify a may-point-to function $\Delta : (S_C \cup R_C) \mapsto 2^{R_C}$ at each program point. For an IR instruction `x := c`, for constant c, the transfer function updates $\Delta(\mathtt{x}) := \emptyset$. For instruction `x := y op z` (for some arithmetic or logical operand `op`), we update $\Delta(\mathtt{x}) := \Delta(\mathtt{y}) \cup \Delta(\mathtt{z})$. For a load instruction `x := *y`, we update $\Delta(\mathtt{x})$ to $\bigcup_{R_C \in \Delta(y)} \Delta(R_C)$. For a store instruction `*x := y`, for all $R_C \in \Delta(\mathtt{x})$, we update $\Delta(R_C) := \Delta(R_C) \cup \Delta(y)$. For recursive procedure calls, a *supergraph* is created by adding control flow edges from the call-site to the procedure head (copying actual arguments to the formal arguments) and from the procedure return to the returning point of the call-site (copying returned value to the variable assigned at the callsite), e.g., in fig. 8, the dashed edges represent supergraph edges. For a malloc instruction $\mathtt{x} := \mathtt{malloc}_A\mathtt{()}$ (where A represents the allocation site), we perform the following steps (in order):

1. Convert all existing occurrences of A_1 to A_{2+}, i.e., for all $r \in S_C \cup R_C$,

if $A_1 \in \Delta(r)$, then update $\Delta(r) := (\Delta(r) \setminus \{A_1\}) \cup \{A_{2+}\}$.

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

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

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

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

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

5 Evaluation

We have implemented S2C on top of the Counter tool [22]. We use four SMT solvers running in parallel for discharging SMT proof obligations discharged by our proof discharge algorithm: `z3-4.8.7`, `z3-4.8.14` [18], `Yices2-45e38fc` [19], and `cvc4-1.7` [1]. An unroll factor of four is used to handle loop unrolling in the C implementation. We use a default value of eight for over- and under-approximation depths (d_o and d_u). The default value of our unrolling parameter k (used for categorization of proof obligations) is five.

The user provides a Spec program (specification), a C implementation, and a file that contains the precondition and postcondition. We use only four distinct ADTs in our specification programs written in Spec, one each for a string (T1), list (T2), tree (T3), and a two-dimensional matrix (T4). For example, the `String` ADT is a sum-type formed by constructors `Invalid` (to encode the well-formedness condition of a string, e.g., it should be non-null), `Nil` (representing an empty string) and `Cons` (representing a recursive construction formed through a product of a character byte and a string).

For each C implementation, an equivalence check requires the identification of lifting constructors that relate C values to the ADT values in Spec. The relations between a Spec value and a lifted value derived from C values (and current memory state) 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 inferred 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, but the inductive invariants are inferred automatically by our algorithm. Our invariant inference algorithm derives the shape of the lifting constructors from the user-specified *Pre* and *Post*, and uses these shapes in enumerating relational guesses at product-CFG nodes. The same Spec program is used for checking equivalence with multiple C implementations that may differ in data layouts and algorithmic strategies. The optimized algorithmic strategies involve loop optimizations through unrolling and manual vectorization.

5.1 Experiments

We consider implementations with diverse data layouts for each Spec ADT (e.g., `List, Matrix`). This includes array, linked list for `List` and row and column major layouts for `Matrix` to name a few. Next we consider each ADT in more detail. For each, we discuss (a) its functions considered, (b) list of C data layouts along with their lifting constructors and (c) different algorithmic strategies encountered during evaluation.

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

Table 2: String lifting constructors and their definitions.

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 (`small`) and another that is optimized for runtime (`fast`). All these library implementations use a *null character* terminated array to represent a string, and the corresponding lifting constructor is $\mathtt{Cstr}_m^{\mathtt{u8}[]}$. `u<N>` represents the N-bit unsigned integer in C. For example, `u8` represents `unsigned char` type.

Further, we implemented custom C implementations for some of these functions that used linked-list and *chunked linked list* data structures to represent a string. In a chunked linked list, a single list node (linked through a `next` pointer) contains a small array (chunk) of values. We use a default chunk size of four for our benchmarks. The corresponding lifting constructors are $\mathtt{Cstr}_m^{\mathtt{lnode(u8)}}$ and $\mathtt{Cstr}_m^{\mathtt{clnode(u8)}}$ respectively. All three `Cstr` lifting constructors are defined in table 2. $\mathtt{Cstr}_m^{\mathtt{lnode(u8)}}$ requires a single argument p representing the C pointer to the list node. On the other hand, $\mathtt{Cstr}_m^{\mathtt{clnode(u8)}}$ requires two arguments p and i where p represents the pointer to the chunked linked list node and i represents the position of the initial character in the chunk.

Usually, we label a lifting constructor based on its underlying data layout. For example, $\texttt{Cstr}_m^{\texttt{clnode(u8)}}$ represents a lifting constructor with a chunked linked list of characters as its underlying data layout. In general, we use the following naming convention for different data layouts. `T[]` represents an array of type T (e.g., `u8[]`). `lnode(T)` represents a linked list node type which encapsulates a value of type `T`. Similarly, `clnode(T)` and `tnode(T)` represent a chunked linked list and a tree node containing values of type `T` respectively.

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

(a) Strlen Specification

```
size_t strlen(char* s);
i32 strlen (i32 s) {
  i32 i ≔ 0_{i32};
  while s[0_{i32}]^{m}_{i8} ≠ 0_{i8}:
    s ≔ s + 1_{i32};
    i ≔ i + 1_{i32};
  return i;
}
```

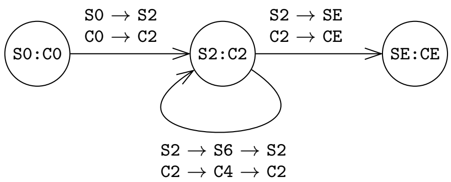
(b) Strlen Implementation using Array

```
typedef struct clnode {
char chunk[4]; struct clnode* next;
} clnode;
size_t strlen(clnode* cl);
i32 strlen (i32 cl) {
  i32 hi ≔ 0x80808080_{i32}; i32 lo ≔ 0x01010101_{i32};
  i32 i  ≔ 0_{i32};
  while true:
    i32 dword_ptr ≔ &cl →^{m}_{clnode} chunk;
    i32 dword     ≔ dword_ptr[0_{i32}]^{m}_{i32};
    if ((dword − lo) & (∼ dword) & hi) ≠ 0_{i32}:
      if dword_ptr[0_{i32}]^{m}_{i8} == 0_{i8}: return i;
      if dword_ptr[1_{i32}]^{m}_{i8} == 0_{i8}: return i + 1_{i32};
      if dword_ptr[2_{i32}]^{m}_{i8} == 0_{i8}: return i + 2_{i32};
      if dword_ptr[3_{i32}]^{m}_{i8} == 0_{i8}: return i + 3_{i32};
    cl ≔ cl →^{m}_{clnode} next; i ≔ i + 4_{i32};
}
```

(c) Optimized Strlen Implementation using Chunked Linked List

Figure 11: Abstracted IRs of Strlen Specification program in fig. 11a along with two different C implementations. Figure 11b is a generic implementation using a null character terminated array to represent a String. Figure 11c is an optimized implementation using chunked linked list representation of String.

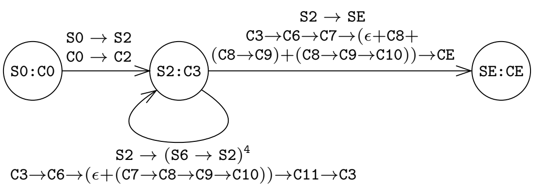
Figure 11 shows the `strlen` specification and two vastly different C implementations. fig. 11b is a generic implementation using a null character terminated array to represent a string similar to a C-style string. The second implementation in fig. 11c differs from fig. 11b in the following: (a) it uses a chunked linked list data layout for string values and (b) it uses an optimized bit manipulation algorithm to identify whether one of the four character bytes is zero (i.e. null) in a chunk at a time. S2C is able to automatically search for and find a bisimulation relation showing equivalence for both implementations against the unaltered specification. Figure 12 shows the Product CFGs and Invariants for these two implementations.



(a) Product CFG for programs figs. 11a and 11b

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. 12a



(c) Product CFG for programs figs. 11a and 11c

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. 12c

Figure 12: Product CFGs and Invariants Tables showing bisimulation between Strlen specification in fig. 11a and two C implementations in figs. 11b and 11c

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

Table 3: List lifting constructors and their definitions.

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: character array ($\texttt{Clist}_m^{\texttt{u32[]}}$), linked list ($\texttt{Clist}_m^{\texttt{lnode(u32)}}$), and a chunked linked list ($\texttt{Clist}_m^{\texttt{clnode(u32)}}$). The lifting constructors are shown in table 3. Although similar to the String lifting constructors, these lifting constructors differ widely in their data encodings. For example, $\texttt{Clist}_m^{\texttt{u32[]}}$ represents a list value constructed from a C array of size n pointed to by p. The list becomes empty when we are at the end of the array. $\texttt{Clist}_m^{\texttt{lnode(u32)}}$ and $\texttt{Clist}_m^{\texttt{clnode(u32)}}$, on the other hand, encodes empty lists using *null pointers*. These encodings are in contrast to the String lifting constructors, all of which uses *null character* to encode the empty string.

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

Table 4: Tree lifting constructors and their definitions.

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 ($\texttt{Ctree}_m^{\texttt{u32[]}}$), and (2) a linked tree node with two pointers for the left and right children ($\texttt{Ctree}_m^{\texttt{tnode(u32)}}$). Both Spec and C programs contain non-tail recursive procedure calls for left and right children. S2C is able to handle such procedure calls as described in section 4.4.

Lifting Constructor	Definition
Ⓣ4 $\texttt{Matrix = MNil} \mid \texttt{MCons(List, Matrix)}$	
$\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{u32[\,][\,]}}(\mathrm{p\ i\ u\ v : i32})$	$\underline{\mathrm{if}}\ (\mathrm{i} \geq_{\mathrm{u}} \mathrm{u})\ \underline{\mathrm{then}}\ \mathrm{MNil}$ $\underline{\mathrm{else}}\ \mathrm{MCons}(\mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[\,]}}(\mathrm{p[i]}_{\mathrm{i32}}^{\mathrm{m}}, 0_{\mathrm{i32}}, \mathrm{v}), \mathrm{Cmat}_{\mathrm{m}}^{\mathrm{u32[\,][\,]}}(\mathrm{p}, \mathrm{i}+1_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}))$
$\mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[r]}}(\mathrm{p\ i\ j\ u\ v : i32})$	$\underline{\mathrm{if}}\ (\mathrm{j} \geq_{\mathrm{u}} \mathrm{v})\ \underline{\mathrm{then}}\ \mathrm{LNil}$ $\underline{\mathrm{else}}\ \mathrm{LCons}(\mathrm{p[i*v+j]}_{\mathrm{i32}}^{\mathrm{m}}, \mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[r]}}(\mathrm{p}, \mathrm{i}, \mathrm{j}+1_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}))$
$\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{u32[r]}}(\mathrm{p\ i\ u\ v : i32})$	$\underline{\mathrm{if}}\ (\mathrm{i} \geq_{\mathrm{u}} \mathrm{u})\ \underline{\mathrm{then}}\ \mathrm{MNil}$ $\underline{\mathrm{else}}\ \mathrm{MCons}(\mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[r]}}(\mathrm{p}, \mathrm{i}, 0_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}), \mathrm{Cmat}_{\mathrm{m}}^{\mathrm{u32[r]}}(\mathrm{p}, \mathrm{i}+1_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}))$
$\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{u32[c]}}(\mathrm{p\ i\ u\ v : i32})$	$\underline{\mathrm{if}}\ (\mathrm{i} \geq_{\mathrm{u}} \mathrm{u})\ \underline{\mathrm{then}}\ \mathrm{MNil}$ $\underline{\mathrm{else}}\ \mathrm{MCons}(\mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[c]}}(\mathrm{p}, \mathrm{i}, 0_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}), \mathrm{Cmat}_{\mathrm{m}}^{\mathrm{u32[c]}}(\mathrm{p}, \mathrm{i}+1_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}))$
$\mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[c]}}(\mathrm{p\ i\ j\ u\ v : i32})$	$\underline{\mathrm{if}}\ (\mathrm{j} \geq_{\mathrm{u}} \mathrm{v})\ \underline{\mathrm{then}}\ \mathrm{LNil}$ $\underline{\mathrm{else}}\ \mathrm{LCons}(\mathrm{p[i+j*u]}_{\mathrm{i32}}^{\mathrm{m}}, \mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[c]}}(\mathrm{p}, \mathrm{i}, \mathrm{j}+1_{\mathrm{i32}}, \mathrm{u}, \mathrm{v}))$
$\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{lnode(u32[\,])}}(\mathrm{p\ v : i32})$	$\underline{\mathrm{if}}\ (\mathrm{p} == 0_{\mathrm{i32}})\ \underline{\mathrm{then}}\ \mathrm{MNil}$ $\underline{\mathrm{else}}\ \mathrm{MCons}(\mathrm{Clist}_{\mathrm{m}}^{\mathrm{u32[\,]}}(\texttt{p} \xrightarrow{\mathrm{m}}_{\texttt{lnode}} \texttt{val}, 0_{\mathrm{i32}}, \mathrm{v}),$ $\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{lnode(u32[\,])}}(\texttt{p} \xrightarrow{\mathrm{m}}_{\texttt{lnode}} \texttt{next}, \mathrm{v}))$
$\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{lnode(u32)[\,]}}(\mathrm{p\ i\ u : i32})$	$\underline{\mathrm{if}}\ (\mathrm{i} \geq \mathrm{u})\ \underline{\mathrm{then}}\ \mathrm{MNil}$ $\underline{\mathrm{else}}\ \mathrm{MCons}(\mathrm{Clist}_{\mathrm{m}}^{\mathrm{lnode(u32)}}(\mathrm{p[i]}_{\mathrm{i32}}^{\mathrm{m}}),$ $\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{lnode(u32)[\,]}}(\mathrm{p}, \mathrm{i}+1_{\mathrm{i32}}, \mathrm{u}))$
$\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{clnode(u32)[\,]}}(\mathrm{p\ i\ u : i32})$	$\underline{\mathrm{if}}\ (\mathrm{i} \geq \mathrm{u})\ \underline{\mathrm{then}}\ \mathrm{MNil}$ $\underline{\mathrm{else}}\ \mathrm{MCons}(\mathrm{Clist}_{\mathrm{m}}^{\mathrm{clnode(u32)}}(\mathrm{p[i]}_{\mathrm{i32}}^{\mathrm{m}}, 0_{\mathrm{i2}}),$ $\mathrm{Cmat}_{\mathrm{m}}^{\mathrm{clnode(u32)[\,]}}(\mathrm{p}, \mathrm{i}+1_{\mathrm{i32}}, \mathrm{u}))$

Table 5: Matrix and auxiliary lifting constructors and their definitions.

5.1.4 Matrix

We wrote a Spec program to count the frequency of a value appearing in a 2D matrix. A matrix is represented as an ADT that resembles a `List` of `Lists` (T4 in table 5). The *C* implementations for a `Matrix` object include a two-dimensional array ($\texttt{Cmat}_m^{\texttt{u32[][]}}$), a flattened row-major array ($\texttt{Cmat}_m^{\texttt{u32[r]}}$), a flattened column-major array ($\texttt{Cmat}_m^{\texttt{u32[c]}}$), a linked list of 1D arrays ($\texttt{Cmat}_m^{\texttt{lnode(u32[])}}$), a 1D array of linked lists ($\texttt{Cmat}_m^{\texttt{lnode(u32)[]}}$) and a 1D array of chunked linked list ($\texttt{Cmat}_m^{\texttt{clnode(u32)[]}}$) data layouts. Note that both `T[r]` and `T[c]` represent an array of type `T`. The *r* and *c* simply emphasizes that these flat arrays are used to represent 2-dimensional matrices in row-major and column-major encodings respectively. We also introduce two auxiliary lifting constructors, $\texttt{Clist}_m^{\texttt{u32[r]}}$ and $\texttt{Clist}_m^{\texttt{u32[r]}}$ for lifting each row of matrices lifted using the corresponding $\texttt{Cmat}_m^{\texttt{u32[r]}}$ and $\texttt{Cmat}_m^{\texttt{u32[r]}}$ Matrix lifting constructors. These constructors are listed in table 5.

TODO: finish the matfreq listings and refer them here, add the entry precondition also for clarity.

```
A0: u32 matfreq(u32 mat[],
A0:             u32 u,
A0:             u32 v,
A0:             u32 x) {
A0:   u32 count = 0;
A0:   for (u32 i=0; i<u; ++i) {
A0:     for (u32 j=0; j<v; ++j) {
A0:       if (mat[i*v+j] == x)
A0:         count++;
A0:     }
A0:   }
A0:   return count;
A0: }

A0: typedef struct lnode {
A0:   u32* val;
A0:   struct lnode* next;
A0: } lnode;
A0:
A0: u32 matfreq(lnode* mat,
A0:             u32 v,
A0:             u32 x) {
A0:   u32 count = 0;
A0:   while (mat) {
A0:     for (u32 j=0; j<v; ++j) {
A0:       if (mat->val[j] == x)
A0:         count++;
A0:     }
A0:     mat = mat->next;
A0:   }
A0:   return count;
A0: }
```

(a) Spec Program

```
A0: typedef struct clnode {
A0:   u32 chunk[4];
A0:   struct clnode* next;
A0: } clnode;
A0:
A0: u32 matfreq(clnode* mat[],
A0:             u32 u,
A0:             u32 x) {
A0:   u32 count = 0;
A0:   for (u32 i=0; i<u; ++i) {
A0:     clnode* cl = mat[i];
A0:     while (cl) {
A0:       if (cl->chunk[0] == x)
A0:         count++;
A0:       if (cl->chunk[1] == x)
A0:         count++;
A0:       if (cl->chunk[2] == x)
A0:         count++;
A0:       if (cl->chunk[3] == x)
A0:         count++;
A0:       cl = cl->next;
A0:     }
A0:     mat = mat->next;
A0:   }
A0:   return count;
A0: }
```

(b) C Program with `malloc()`

Figure 13: Spec and C Programs constructing a Linked List.

Data Layout	**Variant**	**Time(s)**	$\mathbf{d_u, d_o}$	**Data Layout**	**Variant**	**Time(s)**	$\mathbf{d_u, d_o}$
	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$_\mathrm{s}$	9	(1,2)		opt	2.3k	(4,6)
	dietlibc$_\mathrm{f}$	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$_\mathrm{s}$	16	(1,1)		**strcspn**		
	dietlibc$_\mathrm{f}$	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$_\mathrm{s}$	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$_\mathrm{s}$	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$_\mathrm{s}$	42	(1,1)	u8[],clnode(u8)	opt	535	(4,2)
	newlib$_\mathrm{f}$	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)				

Figure 14: Equivalence checking times and minimum under- and over-approximation depth values at which equivalence checks succeeded.

5.2 Results

Figure 14 lists all the various C implementations and the time it took to compute equivalence with their Spec counterparts. 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-terminated array representing the string. This is modeled as a UB assumption $\neg$`(`s_S `is SInvalid)` in the loop body that traverses the string using the `assuming-do` statement discussed in section 4.1. This constraints the inputs to only valid string values in S (that do not contain the `SInvalid` constructor) during equivalence check. The string lifting constructors $\texttt{Cstr}^{\texttt{T}}_{m}(\mathrm{p} : \mathrm{i32}, ...)$ relate the `SInvalid` constructor to the condition `(p == 0`$_{\texttt{i32}}$`)` (as defined in table 3). This ensures that the zero address cannot belong to the null-terminated array. Furthermore, these lifting constructors are used to assert equality of lists in the S and C programs as part of the precondition Pre. This constrains the inputs of C also to valid strings only (that do not contain a character at the null address) during equivalence check due to the (S `def`) assumption. This is an example where (S `def`) and Pre are used in combination to constrain the inputs of S and C to only well-formed values.

TODO: add the sensitivity graphs i.e. keep one depth at max and increase the other incrementally

6 Related Work and Conclusion

The verification of a C implementation against its functional specification through manually-coded refinement proofs has been performed extensively in the seL4 microkernel [25]. While the size of programs considered in our work is much smaller, we hope the ideas in S2C will help automate the proofs for such systems to some degree.

There exists significant prior work on automatic equivalence checking in the context of translation validation [33, 43, 40, 42, 26, 45, 46, 37, 44, 28, 24, 30, 11, 39, 15, 22, 38, 32]. S2C is perhaps most applicable in the context of regression verification [41, 20], where the specification to verify the absence of regressions may be written in a higher-level functional syntax. Using a higher-level functional syntax for the specification allows automatic regression verification across software updates that change data layouts and algorithms.

Frameworks for program equivalence proofs have been developed in interactive theorem provers like Coq [16] where correlations and invariants are identified manually during proof codification. Programming languages like Dafny [27] offer automated program reasoning facilities for imperative languages with abstract data types such as sets and arrays. Such languages perform automatic compile-time checks for manually-specified safety and liveness predicates. Prior work on push-button verification of specific systems [14, 34, 35, 36] involves a combination of careful system design and automatic verification tools like SMT solvers. Constrained Horn Clause (CHC) Solvers [17] encode verification conditions of programs containing loops and recursion, and raise the level of abstraction for automatic proofs. Compared to prior work, 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. Our equivalence checking tool based on our proof discharge algorithm requires the user to only specify the precondition and postcondition — all correlations and invariants involving recursive relations are identified automatically.

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.

TODO: talk about future scope? improvement to the invariant inference algorithm, equivalence in the presence of local allocation and more scalable encodings for data types such as tree where the number of clauses in the d-depth over-approximation is exponential in d.

7 Outline of the Thesis

Chapter 1 of the thesis contains a general introduction to the research problem of automatically verifying C functions against a functional specification. We take a C program and its analogue in a safe functional language, and contrast their differences. This helps us motivate the need to solve this problem. We finish by stating our contribution: a proof discharge algorithm.

In **Chapter 2**, we constrain the programs being considered by formulating the problem statement. This helps us clearly define the subproblem being solved. Next we define the necessary context and terminology (e.g., equivalence) for the rest of the thesis.

Chapter 3 starts with some basic concepts related to equivalence such as bisimulation and product program. The rest of the chapter gradually introduces the proof discharge algorithm and related subprocedures while going through two example program pairs for demonstration.

Next, we formalize previously discussed topics in **Chapter 4**. We begin with the description of our custom language 'Spec'. This is followed by algorithms required in tandem with our proof discharge algorithm for an automatic equivalence checker such as a best-first search algorithm for finding a bisimulation relation and an automatic invariant inference procedure. We finish this chapter with a dataflow analysis formulation of our pointer analysis.

In **Chapter 5**, we evaluate our automatic equivalence checker based on the proof discharge algorithm on a large variety of C programs involving lists, strings, trees and matrices. This includes C programs taken from C library implementations as well as manually written programs. We show that our equivalence checker is able to prove equivalence of a single specification with multiple of C implementations, each varying in its data layouts and algorithmic strategies.

Finally, **Chapter 6** concludes the research and discusses some related works. We note our major ideas and contributions finishing with some potential future improvements to our algorithm.

8 Publications Based on Research Work

References

[1] Cvc4 theorem prover webpage. `https://cvc4.github.io/`, 2023.

[2] diet libc webpage. `https://www.fefe.de/dietlibc/`, 2023.

[3] Gnu libc sources. `https://sourceware.org/git/glibc.git`, 2023.

[4] klibc libc sources. `https://git.kernel.org/pub/scm/libs/klibc/klibc.git`, 2023.

[5] musl libc sources. `https://git.musl-libc.org/cgit/musl`, 2023.

[6] Netbsd libc sources. `http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/`, 2023.

[7] Newlib libc sources. `https://www.sourceware.org/git/?p=newlib-cygwin.git`, 2023.

[8] Openbsd libc sources. `https://github.com/openbsd/src/tree/master/lib/libc`, 2023.

[9] uclibc libc sources. `https://git.uclibc.org/uClibc/`, 2023.

[10] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, 1994.

[11] C. Barrett, 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, editors, *Computer Aided Verification*, pages 291–295, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31686-2.

[12] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 14–25, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964003. URL `https://doi.org/10.1145/964001.964003`.

[13] R. M. Burstall, 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, page 136–143, New York, NY, USA, 1980. Association for Computing Machinery. ISBN 9781450373968. doi: 10.1145/800087.802799. URL `https://doi.org/10.1145/800087.802799`.

[14] H. Chen, 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, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815402. URL `https://doi.org/10.1145/2815400.2815402`.

[15] B. Churchill, 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, pages 1027–1040, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314596. URL `http://doi.acm.org/10.1145/3314221.3314596`.

[16] Coq:Equiv. Program Equivalence in Coq. `https://softwarefoundations.cis.upenn.edu/plf-current/Equiv.html`, apr 2023.

[17] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational verification through horn clause transformation. In X. Rival, editor, *Static Analysis*, pages 147–169, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53413-7.

[18] L. De Moura 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[19] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[20] D. Felsing, 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, pages 349–360, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642987. URL `http://doi.acm.org/10.1145/2642937.2642987`.

[21] C. Flanagan 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, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3540417915.

[22] S. Gupta, A. Rose, and S. Bansal. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4 (OOPSLA), Nov. 2020. doi: 10.1145/3428289. URL `https://doi.org/10.1145/3428289`.

[23] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL `https://doi.org/10.1145/363235.363259`.

[24] A. Kanade, A. Sanyal, and U. P. Khedker. Validation of gcc optimizers through trace generation. *Softw. Pract. Exper.*, 39(6):611–639, Apr. 2009. ISSN 0038-0644. doi: 10.1002/spe.v39:6. URL `http://dx.doi.org/10.1002/spe.v39:6`.

[25] G. Klein, 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, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL `https://doi.org/10.1145/1629575.1629596`.

[26] S. Kundu, 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, pages 327–337, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542513. URL `http://doi.acm.org/10.1145/1542476.1542513`.

[27] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

[28] A. Leung, D. Bounov, and S. Lerner. C-to-verilog translation validation. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 42–47, Sept 2015. doi: 10.1109/MEMCOD.2015.7340466.

[29] LLVM LangRef. LLVM language reference manual. http://llvm.org/docs/LangRef.html, apr 2023.

[30] N. P. Lopes and J. Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *Int. J. Softw. Tools Technol. Transf.*, 18(4):359–374, Aug. 2016. ISSN 1433-2779. doi: 10.1007/s10009-015-0366-1. URL http://dx.doi.org/10.1007/s10009-015-0366-1.

[31] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In M. Sagiv, editor, *Programming Languages and Systems*, pages 46–60, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31987-0.

[32] K. Namjoshi and L. Zuck. Witnessing program transformations. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 304–323. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38855-2. doi: 10.1007/978-3-642-38856-9_17. URL http://dx.doi.org/10.1007/978-3-642-38856-9_17.

[33] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349314. URL http://doi.acm.org/10.1145/349299.349314.

[34] L. Nelson, 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, pages 252–269, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132748. URL http://doi.acm.org/10.1145/3132747.3132748.

[35] L. Nelson, 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, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019. doi: 10.1145/3341301.3359641. URL https://doi.org/10.1145/3341301.3359641.

[36] L. Nelson, 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, Novem-*

ber 4-6, 2020, pages 41–61. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/nelson.

[37] A. Poetzsch-Heffter and M. Gawkowski. Towards proof generating compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):37–51, May 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.03.023. URL http://dx.doi.org/10.1016/j.entcs.2005.03.023.

[38] T. A. L. Sewell, 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, page 471–482, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462183. URL https://doi.org/10.1145/2491956.2462183.

[39] R. Sharma, 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, pages 391–406, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509509. URL http://doi.acm.org/10.1145/2509136.2509509.

[40] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for llvm. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 737–742, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL http://dl.acm.org/citation.cfm?id=2032305.2032364.

[41] O. Strichman and B. Godlin. Regression verification - a practical way to verify programs. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 496–501. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69147-1. doi: 10.1007/978-3-540-69149-5_54. URL http://dx.doi.org/10.1007/978-3-540-69149-5_54.

[42] R. Tate, 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*, pages 264–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480915. URL http://www.cs.cornell.edu/~ross/publications/eqsat/.

[43] J.-B. Tristan, 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 Implemen-*

tation, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993533. URL `http://doi.acm.org/10.1145/1993498.1993533`.

[44] A. Zaks 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, pages 35–51, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68235-6. doi: 10.1007/978-3-540-68237-0_5. URL `http://dx.doi.org/10.1007/978-3-540-68237-0_5`.

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

[46] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, Nov. 2005. ISSN 0925-9856. doi: 10.1007/s10703-005-3402-z. URL `http://dx.doi.org/10.1007/s10703-005-3402-z`.