COUNTEREXAMPLE-GUIDED VERIFICATION OF IMPERATIVE PROGRAMS AGAINST IMPLEMENTATION AGNOSTIC FUNCTIONAL SPECIFICATION

Indrajit Banerjee

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI

JUNE 2023

COUNTEREXAMPLE-GUIDED VERIFICATION OF IMPERATIVE PROGRAMS AGAINST IMPLEMENTATION AGNOSTIC FUNCTIONAL SPECIFICATION

by

Indrajit Banerjee

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of

Master of Science (Research)

to the

Indian Institute of Technology Delhi

JUNE 2023

Certificate

This is to certify that the thesis titled **Counterexample-Guided Verification of Imperative Programs Against Implementation Agnostic Functional Specification** being submitted by **Mr. Indrajit Banerjee** for the award of **Master of Science (Research)** in **Computer Science and Engineering** is a record of bona fide work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Sorav Bansal
Microsoft Chair Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi- 110016

Acknowledgements

I would like to sincerely thank my thesis supervisor Prof. Sorav Bansal for his continuous support during my study and research. His guidance, patience, motivation and long discussions provided a strong platform with clear visibility and research direction.

Besides my advisor, I would like to thank the following members of my Student Research Committee (SRC) for their insightful comments and encouragement that helped me to widen my research from various perspectives:
Prof. Sanjiva Prasad (Dept. of CSE, IIT Delhi)
Prof. Kumar Madhukar (Dept. of CSE, IIT Delhi)
Mr. Akash Lal (Microsoft Research Lab, India)

I am grateful to our research group members: Abhishek Rose, Shubhani at IIT Delhi for their help and motivating discussions on various topics related to my research.

Indrajit Banerjee

Abstract

We describe an algorithm capable of checking equivalence of two programs that manipulate recursive data structures such as linked lists, strings, trees and matrices. The first program, called specification, is written in a succinct and safe functional language with algebraic data types (ADT). The second program, called implementation, is written in C using arrays and pointers. Our algorithm, based on prior work on counterexample guided equivalence checking, automatically searches for a sound equivalence proof between the two programs.

We formulate an algorithm for discharging proof obligations containing relations between recursive data structure values across the two diverse syntaxes, which forms our first contribution. Our proof discharge algorithm is capable of generating falsifying counterexamples in case of a proof failure. These counterexamples help guide the search for a sound equivalence proof and aid in inference of invariants. As part of our proof discharge algorithm, we formulate a program representation of values. This allows us to reformulate proof obligations due to the top-level equivalence check into smaller nested equivalence checks. Based on this algorithm, we implement an automatic (push-button) equivalence checker tool named S2C, which forms our second contribution.

S2C is evaluated on implementations of common string library functions taken from popular C library implementations, as well as implementations of common list, tree and matrix programs. These implementations differ in data layout of recursive data structures as well as algorithmic strategies. We demonstrate that S2C is able to establish equivalence between a single specification and its diverse C implementations.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The problem of equivalence checking between a functional specification and an implementation written in a low level imperative language such as C has been of major research interest. On one side, the functional programming model closely resembles mathematical functions, which allows for comparatively easier proof of algorithmic correctness. On the other hand, a low level imperative language such as C trades the safer abstractions of a functional language for proximity to the machine language resulting in (usually) significantly faster executables, albeit at the cost of sacrificing safety and a drastically higher chance of algorithmic errors. Being able to establish equivalence between the two abstractions would allow the user to take advantage of both worlds – (a) easier proof of functional correctness and (b) more efficient executables. The applications of such an equivalence checker would include (a) program verification, where the equivalence checker is used to verify that the C implementation behaves according to the specification and (b) translation validation, where the equivalence checker attempts to generate a proof of equivalence across the transformations (and translations) performed by an optimizing compiler.

The verification of a C implementation against its manually written functional specification through manually-coded refinement proofs has been performed extensively in the seL4 micro-kernel [30]. Frameworks for program equivalence proofs have been developed in interactive theorem provers like Coq [19] where correlations and invariants are manually identified during proof codification. On the other hand, programming languages like Dafny [32] offer automated program reasoning for imperative languages with abstract data types such as sets and arrays. Such languages perform automatic compile-time checks for manually-specified correctness predicates through SMT solvers. Additionally, there exists significant prior work on translation validation [38, 50, 47, 49, 31, 52, 53, 43, 51, 33, 29, 34, 12, 46, 17, 26, 45, 37] across multiple programming languages with similar models of computation. In most of these applications, soundness in critial, i.e., if the equivalence checker determines the programs to be equivalent, then the programs are indeed equivalent and evidently has equivalent observable behaviour. On the other hand, a sound equivalence checker may be incomplete and fail to prove equivalence of a program pair, even if they were equivalent.

In this work, we present S2C, a *sound* algorithm to automatically search for a proof of equivalence between a functional specification and its optimized C implementations. We will demonstrate how S2C is capable of proving equivalence of multiple equivalent C implementations with vastly different (a) data layouts (e.g. array, linked list representations for a *list*) and (b) algorithmic strategies (e.g. alternate algorithms, optimizations) against a *single* functional specification. This opens the possibility of regression verification [48, 23], where S2C can be used to automate verification across software updates that change memory layouts of data structures.

1.1 A Motivating Example

We start by restricting our attention to programs that construct, read, and write to recursive data structures. In languages like C, pointer and array based implementations of these data-structures are prone to safety and liveness bugs. Similar recursive data structures are also available in safer functional languages like Haskell [35], where algebraic data types (ADTs) [14] ensure several safety properties. We define a minimal functional language, called Spec, that enables the safe and succinct specification of programs manipulating and traversing recursive data structures. Spec is equipped with ADTs as well as boolean (`bool`) and fixed-width bitvector (`i<N>`) types.

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

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

(a) Spec Program

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

(b) C Program with `malloc()`

Figure 1.1: Spec and C Programs constructing a Linked List.

Figures 1.1a and 1.1b show the construction of lists in Spec and C respectively. The `List` ADT in the Spec program is defined at line `A0` in fig. 1.1a. An empty `List` is represented by the *data constructor* `LNil`, whereas a non-empty list uses the `LCons` constructor to combine its first value (`val:i32`) and the remaining list (`tail:List`). The inputs to a Spec procedure (aka function) are its well-typed arguments, which may include recursive data structure (i.e. ADT) values. The inputs to a C procedure are its explicit arguments and the implicit state of program memory at procedure entry. Similarly, the output of a C procedure consists of its explicit return value and the state of program memory at procedure exit.

The Spec function `mk_list` (defined at line `A6` in fig. 1.1a), takes a bitvector of size `32` (`n:i32`). It returns a `List` value representing the list $[(\mathtt{n}-1), (\mathtt{n}-2), \ldots, 1, 0]$. On the other hand, the C function `mk_list` (defined at line `B4` in Figure 1.1b) constructs a *pointer based* linked list representing the list identical to the Spec function. Unlike Spec, the construction of the linked list in C requires explicit allocation of memory through calls to `malloc` in addition to stores to the memory. We are interested in showing that the Spec and C `mk_list` procedures are 'equivalent' i.e., given equal `n` inputs, they both construct lists that are 'equal'.

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

(a) (Abstracted) Spec IR

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

(b) (Abstracted) C IR

Figure 1.2: IRs for the Spec and C Programs in figs. 1.1a and 1.1b respectively.

For comparison, we represent both programs in a common abstract framework. This involves converting both `mk_list` functions to a common logical representation (intermediate representation or IR for short). Figures 1.2a and 1.2b show the IRs of the Spec and C `mk_list` procedures in figs. 1.1a and 1.1b respectively. For the Spec program, the tail-recursive function `mk_list_impl` is converted to a loop and inlined in the top-level function `mk_list` in the IR. In case of the C program in fig. 1.1b, the memory state is made explicit (represented by $\mathbb{m}$), and the size and memory layout of each type is concretized in the IR. For example, the `unsigned` and pointer types are encoded as the `i32` bitvector type. A comprehensive description of the logical model is given in section 2.2.

To reiterate, we are interested in showing equivalence of the Spec and C IRs. Since the argument `n` to both procedures have identical types (i.e. `i32`), their equality is trivially expressible as: $\mathtt{n}_S = \mathtt{n}_C$ [1]. However, Spec uses the `List` ADT to represent a list, whereas the C procedure represents its list using a collection of `lnode` objects linked through their `next` fields, and simply returns a value of type `i32` (`lnode*` in the original C program) pointing to the first `lnode` in the list (or the null value in case of an empty list). In order to express equality between these two list values (of types `List` and `i32`), we would like to 'adapt' one of the values so as to match their types. We choose to lift the C linked list (represented by the `i32` value and the C memory state) to a `List` value using an operator called a *lifting constructor*. Let us call this lifting constructor $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}$, where the expression $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(p : \mathtt{i32})$ represents a `List` value constructed from a C pointer p (pointing to a `lnode` object) in the memory state $\mathbb{m}$. We will give a definition of $\mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}$ later on in section 2.5. For now, such an operator allows us to express equality between the outputs of the Spec and C procedures as $\mathtt{ret}_S = \mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(\mathtt{ret}_C)$, where $\mathtt{ret}_S$ and $\mathtt{ret}_C$ represents the values returned by the respective Spec and C procedures in figs. 1.2a and 1.2b. To further emphasize the fact that we are comparing (a) a Spec ADT value with (b) an ADT value lifted from C values using a lifting constructor, we use '$\sim$' instead of '$=$' and call it a recursive relation: $\mathtt{ret}_S \sim \mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(\mathtt{ret}_C)$.

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

Figure 1.3: CFG representation for Spec and C IRs shown in figs. 1.2a and 1.2b

Consequently, we are interested in proving that given $\mathtt{n}_S = \mathtt{n}_C$ at the procedure entries, $\mathtt{ret}_S \sim \mathtt{Clist}_{\mathbb{m}}^{\mathtt{lnode}}(\mathtt{ret}_C)$ holds at the exits of both procedures. Before going into the proof method, we first introduce an alternate representation of IR, called the Control-Flow Graph (CFG for short).

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

Table 1.1: Node Invariants for Product-CFG in fig. 1.4

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

Figures 1.3a and 1.3b show the CFG representation of the Spec and C IRs in figs. 1.2a and 1.2b respectively. The CFG representation is fundamentally a labeled transition system representation of the corresponding IR, and is further explored in section 2.2. In essence, each node represents a PC location of its IR, and each edge represents (possibly conditional) transition between PCs through instruction execution. For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 1.3b, the edge `C5`→`C3` represents the path `C5`→`C6`→`C7`→`C8`→`C3` in fig. 1.2b.

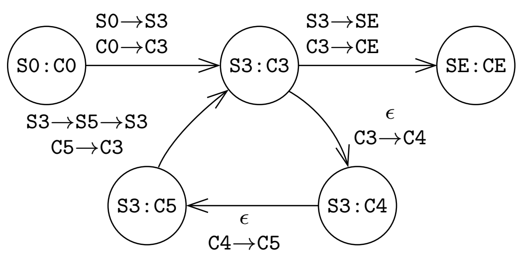


Figure 1.4: Product-CFG between the CFGs in figs. 1.3a and 1.3b

A common approach for showing equivalence between a pair of procedures involve finding a bisimulation relation across the said procedure-pair. Intuitively, a bisimulation relation (a) correlates program transitions across the specification and implementation procedures, and (b) asserts inductively-provable invariants between the machine states of the two procedures at the endpoints of each correlated transition [42]. Bisimulation itself can be represented as a program, called a *product program* [51] and its CFG representation is called a *product*-CFG. Figure 1.4 shows a product-CFG between the Spec and C `mk_list` procedures in figs. 1.3a and 1.3b respectively.

At each node of the product-CFG, *invariants* relate the states of the Spec and C procedures respectively. Table 1.1 lists invariants for the product-CFG in fig. 1.4. At the start node ($\texttt{S0:C0}$) of the product-CFG, the precondition (labeled P) ensures equality of input arguments $\texttt{n}_S$ and $\texttt{n}_C$ at the procedure entries. Inductive invariants (labeled I) need to be inferred at each intermediate product-CFG node (e.g., ($\texttt{S3:C3}$)) relating both programs' states. For example, at node ($\texttt{S3:C5}$), I6 $\texttt{i}_S = \texttt{i}_C$ is an inductive invariant. The inductive invariant I4 $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\texttt{m}}(\texttt{l}_C)$ is another example of a recursive relation and asserts equality between the intermediate Spec and C lists at their respective loop heads. Assuming that the precondition (P) holds at the entry node ($\texttt{S0:C0}$), a bisimulation check involves checking that the inductive invariants hold too, and consequently the postcondition (E) holds at the exit node ($\texttt{SE:CE}$). Checking correctness of a bisimulation relation involves checking whether an invariant holds (along with many other things). These checks result in proof queries which must be discharged by a solver (aka theorem prover).

1.2 Our Contributions

As previously summarized in section 1.1, an algorithm to find a bisimulation based proof of equivalence between a Spec and C procedure involves three major algorithms: A1 An algorithm for construction of a product-CFG by correlating program executions across the Spec and C programs respectively. A2 An algorithm for identification of inductively-provable invariants at intermediate correlated PCs. A3 An algorithm for solving proof obligations generated by A1 and A2 algorithms. Next we list our major contributions.

- Proof Discharge Algorithm: Solving proof obligations (A3) involving recursive relations (generated by A1 and A2) is quite interesting and forms our primary contribution. We describe a *sound* proof discharge algorithm capable of tackling proof obligations involving recursive relations using off-the-shelf SMT solvers. Our proof discharge algorithm is also capable of reconstruction of counterexamples for the original proof query from models returned by the individual SMT queries. These counterexamples form the foundation of counterexample-guided heuristics for A1 and A2 algorithms as we will see soon. As part of our proof discharge algorithm, we reformulate equality of ADT values (i.e. recursive relations) as equivalence of programs and discharge these proof queries using a nested (albeit much simpler) equivalence check.

- Spec-to-C Automatic Equivalence Checker Tool: Our second contribution is S2C, a *sound* equivalence checker tool capable of proving equivalence between a Spec and a C program automatically. S2C either successfully finds a bisimulation relation implying equivalence or it provides a (sound but incomplete) unknown verdict. S2C is based on the Counter tool[26] and uses specialized versions of (a) counterexample-guided correlation algorithm for incremental construction of a product-CFG ((A1)) and (b) counterexample-guided invariant inference algorithm for inference of inductive invariants at correlated PCs in the (partially constructed) product-CFG ((A2)). S2C discharges required verification conditions (i.e. proof obligations) using our Proof Discharge Algorithm. The counterexamples generated by the proof discharge algorithm help steer the search algorithms (A1) and (A2).

1.3 Organization of the Thesis

TODO

The rest of this thesis is divided into five chapter. We begin with a thorough presentation of the topics discussed till now. This includes our Spec language, the logical representation along with bisimulation in the context of equivalence checking. We finish with a discussion on the generated proof obligations and their language.

The next chapter illustrates our proof discharge algorithm using multiple Spec and C sample programs.

Next, we give a detailed description of our automatic Spec-to-C equivalence checker tool S2C along with its most crucial components.

We finish with a comprehensive evaluation of S2C and note its limitations. We conclude our work by reiterating the key ideas presented alongside some related work.

Chapter 2

Preliminaries

2.1 The Specification Language : Spec

We start with an introduction to our specification language, called Spec. Spec supports recursive algebraic data types (ADT) [14] similar to the ones available in functional languages such as Haskell [35] and SML [44]. Spec does not support universal types but does allow ADTs which are mutually recursive. Additionally, Spec is equipped with the following *scalar* types: `unit`, `bool` (boolean) and `i<N>` (fixed-width bitvectors). ADTs can be thought of as 'sum of product' types where each *data constructor* represents a variant (of the sum-type) and the arguments to each data constructor represents its *fields* (of the product-type). For example, the `List` type (defined at `A0` in fig. 1.1a) has two variants `LNil` and `LCons`. `LNil` has no fields while `LCons` has two fields `val` and `tail` of types `i32` and `List` respectively. Additionally, Spec follows *equirecursive* typing rules i.e. a `List` value l and $\texttt{LCons}(1_{\texttt{i32}}, l)$ have *equal* types. Later in section 4.4.9, we further expand on ADTs in the context of a graphical representation of types and values. The language also borrows its expression grammar heavily from functional languages. This includes `let-in`, `if-then-else`, `match` and function application. Pattern matching (i.e. deconstruction) of ADT values is archieved through `match`. Unlike functional languages, Spec only supports first order functions. Also, Spec does not support partial function application. Hence, we constrain our attention to C programs containing only first order functions. Spec is equipped with a special `assuming-do` construct for explicitly providing assertions. Spec also provides intrinsic scalar operators for expressing computation in C succintly yet explicitly. Examples of scalar operators include (a) logical operators (e.g., `and`), (b) bitvector arithmatic operators (e.g., `bvadd(+)`), and (c) relational operators for comparing bitvectors interpreted as unsigned or signed integers (e.g., $\leq_{\mathtt{u,s}}$). The equality operator (=) is only supported for scalar types.

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

Figure 2.1: Simplified expression grammar of Spec language

Figure 2.1 shows the simplified expression grammar for Spec language. ⟨data-cons⟩ represents a ADT data constructor. The '⟨expr⟩ `is` ⟨data-cons⟩' construct returns a `bool` and is used to test whether the top-level constructor of the ADT value ⟨expr⟩ is ⟨data-cons⟩. ⟨scalar-op⟩ includes the logical, arithmatic and relational operators supported by Spec.

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

fn sum_list_impl (l:List) (sum:i32) : i32 =
   match l with
   | LNil => sum
   | LCons(x, rest) => sum_list_impl(rest, sum + x).

fn sum_list (l:List) : i32 = sum_list_impl(l, 0_i32).
```

(a) Spec Program

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

unsigned sum_list(lnode* l) {
  unsigned sum = 0;
  while (l) {
    sum += l->val;
    l = l->next;
  }
  return sum;
}
```

(b) C Program

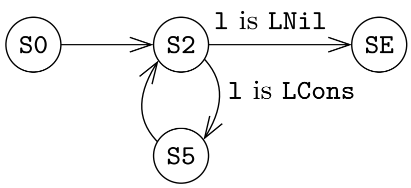
Figure 2.2: Spec and C Programs traversing a Linked List.

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

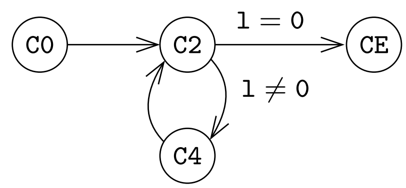
(a) (Abstracted) Spec IR

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

(b) (Abstracted) C IR



(c) CFG of Spec Program



(d) CFG of C Program

Figure 2.3: IRs and CFGs of the Spec and C Programs in figs. 2.2a and 2.2b respectively.

2.2 Abstract Representation of Programs

As outlined in section 1.1, we convert both Spec and C programs to a common abstract representation called the *Control Flow Graph* (CFG for short). This process involves converting both programs to a linear representation called the IR. This section presents both IR and CFG representations for the original Spec and C programs.

2.2.1 Conversion of Programs to their Intermediate Representation

IR is a Three-Address-Code (3AC) style intermediate representation. We often omit intermediate registers in the IR for brevity, and refer to this as the *abstracted* IR. We have already seen the IRs (in figs. 1.2a and 1.2b) for the Spec and C programs that construct lists in figs. 1.1a and 1.1b.

Figures 2.2a and 2.2b show Spec and C programs that traverse a list of integers and return the sum of all the values in it. The corresponding IR programs for the above are shown in figs. 2.3a and 2.3b.

The following major steps are performed during conversion of a Spec source to its IR representation:

1. `match` statements are converted to explicit `if-else` conditionals where each branch is associated with a `match` branch. The *sum-is* operator is used in the condition to query the top-level data constructor of an expression. The fields of the data constructor are bound to variables using the *product-access* or *accessor* operator. For example, the `match` statement in `A3` (in fig. 2.2a) is lowered to `if-else` in fig. 2.3a, where '`l` is `LCons`' is used to test whether `l` is of kind `LCons` and '`l.val`' is used to extract the `val` field of `LCons` data constructor. Importantly, the expression 'e.`fi`' is well-formed iff 'e is $V_{\texttt{fi}}$', where $V_{\texttt{fi}}$ represents the data constructor containing the field `fi`. The construction of the IR guarantees the well-formedness of all expressions.

2. All tail-recursive calls are converted to loops in the IR. However, all non-tail procedure calls are preserved as is. This transformation enables direct correlation (during equivalence checking) of tail-recursion in Spec with native loops in C. For example, the tail recursive function `sum_list_impl` in `A2` (in fig. 2.2a) is converted to a non-recursive function with a loop.

3. All *helper* functions[1] are inlined at their call-site. We are only interested in proving equivalence of non-helper functions in Spec with their C counterparts. For example, the helper function `sum_list_impl` (now non-recursive due to previous step), is inlined at call-site `A7` in fig. 2.2a.

Similarly, the following transformations are carried out during conversion of a C source to its IR:

[1]We use a special marker to designate a function as 'helper' in Spec. For simplicity, this marker is omitted and instead helper function names are ended with the '`_impl`' suffix.

1. Non-determinism in the original C program is determinized in the IR. This includes concretizing the size and memory layouts of both scalar (e.g. `int`) and compound (e.g., `struct`) types, along with fixing the order of evaluation in case it is unspecified. For example, during conversion of C program in fig. 1.1b to IR (in fig. 1.2b), the sizes of pointer and `unsigned` types are fixed to 32 bits (i.e. `i32`). Similarly, the memory layout (including alignment and offset) of `lnode` struct defined in `B0` (in fig. 1.1b) is chosen. The implications of determinizing the C program behaviour are further discussed in chapter 4. For now, it is sufficient to note that we are interested in equivalence between Spec and this determinized version of C.

2. The memory state of the C program is made explicit, represented using the byte-addressable array '$\mathbb{m}$'. Memory loads and stores are represented using explicit operations on $\mathbb{m}$, e.g., (a) memory load in `C3` in `C4` in fig. 2.3b, and (b) memory stores in `C5` and `C6` in fig. 1.2b. The memory load and store operators are defined promptly in section 2.2.2

3. We annotate calls to memory allocation functions (e.g., `malloc`) with their call-site. For example, $\mathtt{malloc}_{\mathtt{C4}}$ is annotated with its call-site `C4` in fig. 1.2b. These annotations are used by a points-to analysis done as part of our equivalence checking procedure, and defined subsequently in section 4.1.

2.2.2 IR Instructions

Note that both Spec and C programs are converted to the common IR. On the Spec side, IR supports scalar as well as ADT types defined in the Spec program under consideration. The IR also inherits the scalar operators available as part of Spec. Each ADT value can be thought of as a key-value dictionary that maps each of its field names to their respective values. These key-value pairs are accessed using the previously introduced *accessor* operator, e.g., l.`val` and l.`next` represents the first and second fields of the `LCons` data constructor in fig. 2.3a. Recall that, the IR also allows querying the top-level variant of an ADT value using the *sum-is* operator, e.g., l is `LNil` in fig. 2.3a. The `val` field is associated with the `LCons` data constructor and evidently, l.`val` (and l.`next`) is only *well-formed* under l is `LCons`. As discussed, the well-formedness of all *accessor* expressions are preserved during construction of IR for a Spec program. Using *accessor* and *sum-is* operators, a `List` value l can be expanded as:

$$U_S : l = \underline{\texttt{if}}\ l \text{ is } \texttt{LNil}\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(l.\texttt{val}, l.\texttt{next}) \tag{2.1}$$

In this expanded representation of l, the *sum-deconstruction* operator '$\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$' conditionally deconstructs the sum type into its variants `LNil` and `LCons`. The *underlined* $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ operator is a stricter version of `if-then-else`, and is only used for ADT values. An $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ expression e (for an ADT type T) must satisfy the following properties: (a) e has exactly one branch for each data construction of T (in the order they are defined), and (b) the branch associated with the data constructor V has the form $V(e_1, e_2, \dots)$ i.e. its top-level operator is V. For example, an $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ expression for the `List` type must be of the form: '$\underline{\texttt{if}}\ e_1\ \underline{\texttt{then}}\ \texttt{LNil}\ \underline{\texttt{else}}\ \texttt{LCons}(e_2, e_3)$' for some expressions e_1, e_2, e_3. Equation (2.1) is called the *unrolling procedure* for the `List` variable l. We can similarly define the unrolling procedure for any ADT variable (based on the definition of the ADT).

On the C side, the size of a pointer is fixed[2] and the memory state is modeled as a byte-addressable array over bitvectors (represented by $\mathtt{m}$). "$\mathtt{m}[p]_{\mathtt{T}}$" represents a memory load operation and is equal to the bytes at addresses $[p,\ p+\texttt{sizeof(T)})$ in $\mathtt{m}$, interpreted as a value of type `T`. Similarly, "$\mathtt{m}[p \leftarrow v]_{\mathtt{T}}$" represents a memory store operation and is equal to $\mathtt{m}$ everywhere except at addresses $[p,\ p+\texttt{sizeof(T)})$ which contains the value v of type `T` (e.g., `C5` in fig. 1.2b). We use the following two C-like syntaxes to represent more intricate memory loads succintly:

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

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

[2]We choose an address width of 4 bytes or 32 bits for our evaluation.

Recall that the size and layout of each type in C is concretized in the IR, and hence the values '`offsetof(T,f)`' and '`sizeof(T)`' are constants. We use the '`addrof()`' operator to extract the address of a memory load expression: "$\texttt{addrof(}\mathbb{m}[p]_{\texttt{T}}\texttt{)}$" is equivalent to p. For example, at PC `C5` in fig. 1.2b, $\texttt{addrof}(p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{val}) \Leftrightarrow p + \texttt{offsetof}(\texttt{lnode}, \texttt{val})$. Additionally, given a bitvector expression e, "$e[\texttt{ub} : \texttt{lb}]$" represents a bitvector extract operation that extracts the bits in the range [lb,ub] from e resulting in a $(ub - lb + 1)$-sized bitvector expression.

2.2.3 Control-Flow Graph Representation

Figures 2.3c and 2.3d show the Control-Flow Graph (CFG) representation of the Spec and C IRs in figs. 2.3a and 2.3b respectively. The Control-Flow Graph is an alternate graphical representation of an IR program that emphasizes the control flow structures of the static program. Each CFG node represents a program point (i.e. IR PC) and is denoted by n. The CFG representation is analogous to a deterministic labeled transition systems and uses a symbolic state Ω_n to represent the machine state at node n. An edge from n to n' (denoted by $\omega[n \to n']$) represents transition from n to n' through execution of instructions and is associated with:

1. An *edge condition* representing the condition that must be satisfied by Ω_n to trigger the edge ω.
2. A *transfer function* representing the symbolic state at n' ($\Omega_{n'}$) as a function of Ω_n i.e. how the machine state is mutated along the edge ω.
3. An *UB condition* representing the condition that must be satisfied by Ω_n for the transition ω to be well-defined. For a Spec function, assertions expressed using the `assuming-do` statement form the UB assumptions.

For brevity, we often represent a sequence of instructions with a single edge, e.g., in fig. 1.3b, the edge `C5`→`C3` represents the path `C5`→`C6`→`C7`→`C8`→`C3`. In such a case, the transfer function of the edge is the composition of the sequence of instructions. A CFG must contain exactly one entry node (representing the entry to the function), but may contain multiple exit nodes (each representing an exit from the function). An edge incident on an exit node is called an *exit edge* and is only associated with an *action* representing the values returned, as a function of the symbolic state at the source node. Actions form the observable behaviour of a CFG while transition through non-exit edges are internal to the program. Recall that for a C CFG, the action includes both the returned value (if non-void) and the memory state. We omit these transfer functions in the CFG figures (as they are shown in their corresponding IR) and only show the edge conditions (unless they are *true*). Henceforth, we refer to the CFGs of Spec and C functions as $\mathcal{S}$ and $\mathcal{C}$ respectively.

2.3 Equivalence Definition

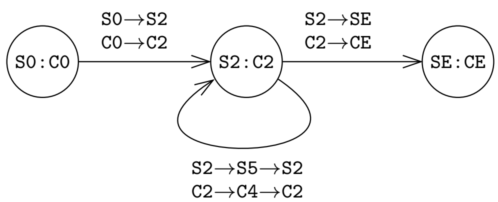
Given (1) a Spec function specification $\mathcal{S}$, (2) a C implementation $\mathcal{C}$, (3) a precondition Pre that relates the initial inputs $\texttt{Input}_S$ and $\texttt{Input}_C$ to $\mathcal{S}$ and $\mathcal{C}$ respectively, and (4) a postcondition $Post$ that relates the final outputs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

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

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

Recall that the observables of $\mathcal{S}$ and $\mathcal{C}$ are the actions associated with their exit edges (i.e. returned values). For $\mathcal{S}$, observables include the explicit value returned. For $\mathcal{C}$, observables

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



(a) Product-CFG

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

(b) Node Invariants of the Product-CFG

Figure 2.4: Product-CFG between the CFGs in figs. 2.3c and 2.3d. The inductive invariants of the Product-CFG are given in fig. 2.4b.

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

Sometimes, the user may be interested in constraining the nature of inputs to $\mathcal{C}$ for the purpose of checking equivalence only for *well-defined* inputs. In those circumstances, we use a combination of *Pre* and ($\mathcal{S}$ `def`) to constrain the execution of $\mathcal{C}$ to inputs for which we are interested in proving equivalence. For example, the C library function $\texttt{strlen(char* str}_C\texttt{)}$ is well-defined only if $\texttt{str}_C$ represents a valid null character terminated string. This includes the assumption that the pointer $\texttt{str}_C$ may not be null. Since Spec has no notion of pointers, we expose this conditional well-definedness of C strings through an explicit constructor e.g. `SInvalid` for the `String` ADT defined as:

$$\texttt{Str} = \texttt{SInvalid} \mid \texttt{SNil} \mid \texttt{SCons}(\texttt{i8}, \texttt{Str})$$

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

2.4 Bisimulation Relation

Recall that, we construct a *bisimulation relation* to identify equivalence between the CFGs of Spec and C procedures. A bisimulation relation correlates the transitions of $\mathcal{S}$ and $\mathcal{C}$ in lockstep, such that the lockstep execution ensures identical observable behaviour. A bisimulation relation between two programs can be represented using a *product program* [51] and the CFG representation of a product program is called a *product*-CFG. Figure 2.4a shows a product-CFG, that encodes the lockstep execution (bisimulation relation) between the CFGs in figs. 2.3c and 2.3d.

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

An edge in the product-CFG is formed by pairing a *path* (a sequence of edges) in $\mathcal{S}$ with a path in $\mathcal{C}$[4]. A product-CFG edge encodes the lockstep execution of its correlated paths. For example, the product-CFG edge (`S2:C2`)$\rightarrow$(`S2:C2`) is formed by pairing `S2`$\rightarrow$`S5`$\rightarrow$`S2` and `C2`$\rightarrow$`C4`$\rightarrow$`C2` in figs. 2.3c and 2.3d respectively, and represents that when $\mathcal{S}$ makes the transition `S2`$\rightarrow$`S5`$\rightarrow$`S2`, $\mathcal{C}$ makes the transition `C2`$\rightarrow$`C4`$\rightarrow$`C2` in lockstep. In general, a product-CFG edge e may correlate a finite path ρ_S in $\mathcal{S}$ with a finite path ρ_C in $\mathcal{C}$, written $e = (\rho_S, \rho_C)$. The empty path ϵ in $\mathcal{S}$ may be correlated with a finite path in $\mathcal{C}$, effectively simulating a *stuttering bisimulation* relation. However, a product-CFG is only well-formed (i.e. represents a valid bisimulation relation) if no loop path in $\mathcal{C}$ is correlated with ϵ in $\mathcal{S}$. For example, fig. 1.4 shows the product-CFG between the programs in figs. 1.3a and 1.3b respectively. The edges (`S3:C3`)$\rightarrow$(`S3:C4`) and (`S3:C4`)$\rightarrow$(`S3:C5`) correlate the empty path ϵ with the non-empty paths `C3`$\rightarrow$`C4` and `C4`$\rightarrow$`C5` respectively. However, the only loop path `C3`$\rightarrow$`C4`$\rightarrow$`C5`$\rightarrow$`C3` in $\mathcal{C}$ is still correlated with the non-empty path `S3`$\rightarrow$`S5`$\rightarrow$`S3` in $\mathcal{S}$ and thus, the product-CFG in fig. 1.4 satisfies this well-formedness criterion. This well-formedness condition is required to preserve *divergence*, i.e. either both programs terminate or

[4]TODO: to keep the discussion simple, we consider paths to be sequence of edges. however, a more general approach of pathset is used based on the Counter tool [26]. We will explore pathsets in more detail later on.

both continue indefinitely.

At the start node ($\mathtt{S0:C0}$) of the product-CFG in fig. 2.4a, the precondition *Pre* (labeled (P)) ensures equality of input lists $\mathtt{l}_S$ and $\mathtt{l}_C$ at procedure entries. *Inductive invariants* (labeled (I)) are inferred at each intermediate product-CFG node (e.g., ($\mathtt{S2:C2}$)) that relate the values of $\mathcal{S}$ with values and memory state of $\mathcal{C}$. At the exit node ($\mathtt{SE:CE}$) of the product-CFG, the postcondition *Post* (labeled (P)) represents equality of observable outputs and forms our overall proof obligation. Assuming that the precondition *Pre* ((P)) holds at the entry node ($\mathtt{S0:C0}$), a bisimulation check involves checking that the inductive invariants ((I)) hold too, and consequently the postcondition *Post* ((E)) holds at the exit node ($\mathtt{SE:CE}$). The input-output specification (i.e. $(Pre, Post)$) is manually provided by the user while all inductive invariants are identified by an invariant inference algorithm described in section 4.3.

2.5 Recursive Relation

In section 1.1, we briefly introduced a lifting constructor ($\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}$) and recursive relations. In fig. 2.4b, the precondition ((P)) is another instance of a recursive relation: "$\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$" where $\mathtt{l}_S$ and $\mathtt{l}_C$ represent the input arguments to the Spec and C procedures respectively, `lnode` is the C `struct` type that contains the `val` and `next` fields (defined at `B0` in fig. 2.2b), and $\mathbb{m}$ is the byte-addressable array representing the current memory state of the C program. $l_1 \sim l_2$ is read l_1 *is recursively equal to* l_2 and is semantically equivalent to $l_1 = l_2$. The '$\sim$' simply emphasizes that l_1 and l_2 are (possibly recursive) ADT values. The lifting constructor $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}$ 'lifts' a C pointer value p (pointing to an object of type `struct lnode`) and memory state $\mathbb{m}$ to a (possibly infinite in case of a circular list) `List` value, and is defined through its *unrolling procedure* as follows:

$$\begin{aligned} U_C : \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p:\mathtt{i32}) = \underline{\mathtt{if}}\ p = 0\ \underline{\mathtt{then}}\ \mathtt{LNil} \\ \underline{\mathtt{else}}\ \mathtt{LCons}(p \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{val}, \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{next})) \end{aligned} \tag{2.2}$$

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

2.6 Proof Obligations

As previously discussed, algorithms for (a) incremental construction of a Product-CFG and (b) inference of invariants at intermediate PCs in the (partially constructed) product-CFG, are based on prior work[26] and discussed subsequently in sections 4.2 and 4.3. For now, we discuss the proof obligations that arise from a given product-CFG. Recall that a bisimulation check involves checking that all inductive invariants (and the postcondition $Post$) hold at their associated product-CFG nodes.

We use relational Hoare triples to express these proof obligations [13, 27]. If ϕ denotes a predicate relating the machine states of $\mathcal{S}$ and $\mathcal{C}$, then for a product-CFG edge $e = (\rho_S, \rho_C)$, $\{\phi_s\}(e)\{\phi_d\}$ denotes the condition: if any machine states σ_S and σ_C of programs $\mathcal{S}$ and $\mathcal{C}$ are related through precondition $\phi_s(\sigma_S, \sigma_C)$ and the finite paths ρ_S and ρ_C are executed in $\mathcal{S}$ and $\mathcal{C}$ respectively, then execution terminates normally in states $\sigma_S^{'}$ (for $\mathcal{S}$) and $\sigma_C^{'}$ (for $\mathcal{C}$) and postcondition $\phi_d(\sigma_S^{'}, \sigma_C^{'})$ holds.

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

$$(\phi_s \wedge \texttt{pathcond}_{\rho_S} \wedge \texttt{pathcond}_{\rho_C} \wedge \texttt{ubfree}_{\rho_S}) \Rightarrow \texttt{WP}_{\rho_S,\rho_C}(\phi_d) \tag{2.3}$$

Here, $\texttt{pathcond}_{\rho_X}$ represents the condition that path ρ is taken in program X and $\texttt{ubfree}_{\rho_S}$ represents the condition that execution of $\mathcal{S}$ along path ρ_S is free of undefined behaviour. $\texttt{WP}_{\rho_S,\rho_C}(\phi_d)$ represents the weakest-precondition of the predicate ϕ_d across the product-CFG edge $e = (\rho_S, \rho_C)$. From now on, we will use '$\texttt{LHS}$' and '$\texttt{RHS}$' to refer to the antecedent and consequent of the implication operator '$\Rightarrow$' in eq. (2.3).

For example, checking that the loop invariant (I2) $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$ holds at $(\texttt{S2:C2})$ in fig. 2.4a requires us to prove the following two proof obligations: (1) $\{\phi_{\texttt{S0:C0}}\}(\texttt{S0}{\rightarrow}\texttt{S2}, \texttt{C0}{\rightarrow}\texttt{C2})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)\}$ and (2) $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}{\rightarrow}\texttt{S5}{\rightarrow}\texttt{S2}, \texttt{C2}{\rightarrow}\texttt{C4}{\rightarrow}\texttt{C2})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)\}$. Using weakest precondition predicate transformer, the proof obligation (2) reduces to the following first-order logic formula:

$$
\begin{aligned}
\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \land \texttt{sum}_S = \texttt{sum}_C \land (\texttt{l}_S \text{ is } \texttt{LCons}) \land (\texttt{l}_C \neq 0)& \\
\Rightarrow \texttt{l}_S.\texttt{next} \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{next})&
\end{aligned}
\tag{2.4}
$$

Due to the presence of recursive relations, these proof queries (e.g., eq. (2.4)) cannot be solved directly by off-the-shelf solvers and require special handling. The next chapter illustrates our proof discharge algorithm for solving proof queries involving recursive relations.

Chapter 3

Proof Discharge Algorithm through Illustative Examples

This section demonstrates our proof discharge algorithm through examples. We consider proof obligations generated due to invariants shown in table 1.1 and fig. 2.4b for the product-CFGs in figs. 1.4 and 2.4a respectively. We start by describing the properties of the proof discharge algorthm. We also list the properties of the proof obligations generated by our equivalence checker; these properties are essential for the correctness of our proof discharge algorithm. Next, the proof discharge algorithm is explored using sample proof obligations, and we finish with an overview of the algorithm.

3.1 Properties of Proof Discharge Algorithm

3.1.1 Soundness of Proof Discharge Algorithm

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

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

3.1.2 Conjunctive Recursive Relation Property of Proof Obligations

Resolving the truth value of a proof obligation that contains a recursive relation such as $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)$ is unclear. Fortunately, the shapes of the proof obligations generated by our equivalence checker are restricted. Our equivalence checking algorithm ensures that, for an invariant $\phi_s = (\phi^{1}_{\mathtt{s}} \wedge \phi^{2}_{\mathtt{s}} \wedge ... \wedge \phi^{\mathtt{k}}_{\mathtt{s}})$, at any node s of a product-CFG, if a recursive relation appears in ϕ_s, it must be one of ϕ^1_s, ϕ^2_s, ..., or ϕ^k_s. We call this the *conjunctive recursive relation* property of an invariant ϕ_s.

A proof obligation $\{\phi_s\}(e)\{\phi_d\}$, where $e = (\rho_S, \rho_C)$, gets lowered using $\mathtt{WP}_e(\phi_d)$ (as shown in eq. (2.3)) to a first-order logic formula of the following form:

$$(\eta^l_1 \wedge \eta^l_2 \wedge ... \wedge \eta^l_m) \Rightarrow (\eta^r_1 \wedge \eta^r_2 \wedge ... \wedge \eta^r_n) \tag{3.1}$$

Thus, due to the conjunctive recursive relation property of ϕ_s and ϕ_d, any recursive relation in eq. (3.1) must appear as one of η_i^l or η_j^r. To simplify proof obligation discharge, we break a first-order logic proof obligation P of the form in eq. (3.1) into multiple smaller proof obligations of the form $P_j : (\texttt{LHS} \Rightarrow \eta_{\mathtt{j}}^{\mathtt{r}})$, for $j = 1..n$. Each proof obligation P_j is then discharged separately. We call this conversion from a bigger query to multiple smaller queries, `RHS`-*breaking*.

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

3.2 Iterative Unification and Rewriting Procedure

We begin with some definitions. An expression e whose top-level constructor is a lifting constructor, e.g., $e = \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(\texttt{l}_C)$, is called a *lifted expression*. An expression e of the form $v.\texttt{a}_1.\texttt{a}_2...\texttt{a}_\texttt{n}$ i.e. a variable nested within *zero* or more *accessors*, is called a *pseudo-variable*. Note that, a variable is itself a pseudo-variable. An expression e in which (a) each accessor (e.g., '_.`tail`') and (b) each *sum-is* operator (e.g., '_ is `LCons`') operate on a pseudo-variable, is called a *canonical expression*. It is possible to convert any expression e into its canonical form $\hat{e}$. For example, the canonical form of $a + \texttt{LCons}(b, l).\texttt{tail}.\texttt{val}$ is given by $a + l.\texttt{val}$, where $l.\texttt{val}$ is a pseudo-variable. The pseudo-code for the canonicalization procedure is given in section 4.4.2.

Consider the expression tree of a canonical expression $\hat{e}$. The internal nodes of $\hat{e}$ represents ADT data constructors and the <u>`if`</u>-<u>`then`</u>-<u>`else`</u> sum-deconstruction operator. The leaves of $\hat{e}$ (also called *atoms* of $\hat{e}$) are the pseudo-variables (of scalar and ADT type), the scalar expressions (of `unit`, `bool` and `i<N>` types), and lifted expressions.

The *expression path* to a node v in $\hat{e}$'s tree is the path from the root of $\hat{e}$ to the node v. The *expression path condition* represents the conjunction of all the <u>`if`</u> conditions (if the <u>`then`</u> branch of taken along the path), or their negation (if the <u>`else`</u> branch is taken along the path) for each if-then-else along the path. For example, in the expression if c then a else b, the expression path condition of c is `true`, of a is c, and of b is $\neg c$.

3.2.1 Unification Procedure

An unification procedure attempts to unify two expressions by unifying their tree structures created by data constructors and the if-then-else operator. The unification procedure either fails to unify, or it returns tuples $\langle p_1, a_1, p_2, e_2 \rangle$ where atom a_1 at expression path condition p_1 in one expression is correlated with expression e_2 at expression path condition p_2 in the other expression.

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

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

If the top-level operator in both e_1 and e_2 is if-then-else, we unify each child (condition and branch expressions) of the corresponding if-then-else operators. Recall that the if-then-else operator (introduced in section 2.2) for an ADT T must have exactly one branch for each data constructor of T, and the branch associated with the data constructor V has V in its top-level. Whenever we descend down an if-then-else operator, we conjunct the if condition (if then branch is taken) or its negation (if else branch is taken) with its associated expression path condition. This allows us to keep track of the expression path conditions for both expressions during recursive descent to their children.

If one of e_1 and e_2 (say e_2) is atomic, unification always succeeds and returns $\langle p_2, e_2, p_1, e_1 \rangle$. The pseudo-code for the unification procedure is given in section 4.4.3. With each atom of an ADT type, we associate an *unrolling procedure*. By definition, an ADT atom is either a pseudo-variable or a lifted expression. Each (pseudo-)variable is associated with its unrolling procedure governed by its type. For example, the unrolling procedure for a `List` variable l is given by U_S (eq. (2.1)). For lifted expressions, the unrolling procedure is given by its definition, e.g., U_C (eq. (2.2)) for the lifting constructor $\mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}$.

3.2.2 Unification under Rewriting

Given two *canonical* expressions e_a and e_b at expression path conditions p_a and p_b respectively, an *iterative unification and rewriting procedure* $\Theta(p_a, e_a, p_b, e_b)$ is used to identify a set of correlation tuples between the atoms in the two expressions. This iterative procedure begins with an attempt to unify e_a and e_b. If this unification fails, we return a failure for the original expressions e_a and e_b. Else, we obtain correlation tuples between atoms and expressions (with their expression path conditions). If the unification correlates an atom a_1 at expression path condition p_1 with another atom a_2 at expression path condition p_2, we add $\langle p_1, a_1, p_2, a_2 \rangle$ to the final output. Otherwise, if the unification correlates an atom a_1 at expression path condition p_1 to a non-atomic expression e_2 at expression path condition p_2, we *rewrite* a_1 using its unrolling procedure to obtain expression e_1. The unification algorithm then proceeds by unifying e_1 and e_2 through a recursive call to $\Theta(p_1, e_1, p_2, e_2)$. The maximum number of rewrites performed by $\Theta(p_a, e_a, p_b, e_b)$ (before termination) is bounded by the sum of number of ADT data constructors in e_a and e_b. The pseudo-code for iterative unification and rewriting is given in section 4.4.4.

3.2.3 Decomposition of Recursive Relations

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

$$l_1 \sim l_2 \Leftrightarrow \bigwedge_{i=1}^{n} (p_1^i \wedge p_2^i \rightarrow (a_1^i = a_2^i)) \tag{3.2}$$

For example, the unification of '$\underline{\texttt{if}}$ c_1 $\underline{\texttt{then}}$ $\texttt{LNil}$ $\underline{\texttt{else}}$ $\texttt{LCons}(0, l_1)$' and '$\underline{\texttt{if}}$ c_2 $\underline{\texttt{then}}$ $\texttt{LNil}$ $\underline{\texttt{else}}$ $\texttt{LCons}(i, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$' yields the correlation tuples: $\langle true, c_1, true, c_2\rangle$, $\langle \neg c_1, 0, \neg c_2, i\rangle$ and $\langle \neg c_1, l_1, \neg c_2, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2)\rangle$. Consequently, the recursive relation "$\underline{\texttt{if}}$ c_1 $\underline{\texttt{then}}$ $\texttt{LNil}$ $\underline{\texttt{else}}$ $\texttt{LCons}(0, l_1)$ $\sim$ $\underline{\texttt{if}}$ c_2 $\underline{\texttt{then}}$ $\texttt{LNil}$ $\underline{\texttt{else}}$ $\texttt{LCons}(i, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$" decomposes into $(c_1 = c_2) \wedge (\neg c_1 \wedge \neg c_2 \rightarrow 0 = i) \wedge (\neg c_1 \wedge \neg c_2 \rightarrow l_1 \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$. Similarly, the decomposition of $l_1 \sim \texttt{LCons}(42, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$ is given by $(l_1 \text{ is } \texttt{LCons}) \wedge (l_1 \text{ is } \texttt{LCons} \rightarrow l_1.\texttt{val} = 42) \wedge (l_1 \text{ is } \texttt{LCons} \rightarrow l_1.\texttt{next} \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$ [1]. In case of a failed unification, the *decomposition* is defined to be *false*, e.g., $\texttt{LNil} \sim \texttt{LCons}(0, l)$ decomposes into $false$.

Each conjunctive clause of the form $(p_1^i \wedge p_2^i \rightarrow (a_1^i = a_2^i))$[2] in the decomposition is called a *decomposition clause*. A decomposition clause may relate only atomic values, i.e., it may relate (a) two scalars or (b) two ADT variable(s) and/or lifted expression(s). However, we restrict the shapes of recursive relation invariants such that each recursive relation in its decomposition *strictly* relates ADT values to lifted expressions. The invariant shapes along with the invariant inference procedure is discussed in section 4.3. We *decompose* a recursive relation by replacing it with its decomposition. We *decompose* a proof obligation by decomposing all recursive relations in it.

3.3 Categorization of Proof Obligations

We *unroll a recursive relation* $l_1 \sim l_2$ by rewriting the top-level expressions l_1 and l_2 through their unrolling procedures (if possible) and decomposing it. We *unroll an expression* e by unrolling each recursive relation in e. More generally, the k-unrolling of e is found by unrolling the $(k-1)$-unrolling of e recursively. For a decomposed proof obligation $P_D : \texttt{LHS} \Rightarrow \texttt{RHS}$, we identify its k-unrolling (say P_K), where k is a fixed parameter called the *unrolling parameter*. After k-unrolling, we *eliminate* those decomposition clauses $(p_1 \wedge p_2 \rightarrow (a_1 = a_2))$ in P_K whose $(p_1 \wedge p_2)$ evaluates to false under LHS ignoring all recursive relations, yielding an equivalent proof obligation, say

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

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

P_E. For example, the one-unrolling of P : $\mathtt{LHS} \Rightarrow l \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(0)$, after elimination, yields P_E : $\mathtt{LHS} \Rightarrow l$ is $\mathtt{LNil}$. We categorize a proof obligation P : $\mathtt{LHS} \Rightarrow \mathtt{RHS}$ based on the k-unrolled form of its decomposition (i.e. P_E) as follows:

- Type I: P_E does not contain recursive relations
- Type II: P_E contains recursive relations *only* in the $\mathtt{LHS}$
- Type III: P_E contains recursive relations in the $\mathtt{RHS}$

The categorization method is *sound* as long as the elimination of decomposition clauses is sound (but possibly not precise). In other words, it is possible that we are unable to eliminate a recursive relation in P_K, due to an imprecise algorithm for elimination of decomposition clauses. However, our proof discharge algorithm remains sound irrespective of such imprecision during categorization. Henceforth, we will simply use k-unrolling of P to refer to P_E directly. Next, we describe the algorithm for each type of proof obligations in sections 3.4 to 3.6.

3.4 Handling Type I Proof Obligations

In fig. 1.4, consider a proof obligation generated across the product-CFG edge $(\mathtt{S0}\!:\!\mathtt{C0}) \rightarrow (\mathtt{S3}\!:\!\mathtt{C3})$ while checking if the I4 invariant in table 1.1, $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)$ holds at $(\mathtt{S3}\!:\!\mathtt{C3})$: $\{\phi_{\mathtt{S0:C0}}\}(\mathtt{S0}\rightarrow\mathtt{S3}, \mathtt{C0}-$ $\mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)\}$. The precondition $\phi_{\mathtt{S0:C0}} \equiv (\mathtt{n}_S = \mathtt{n}_C)$ does not contain a recursive relation. When lowered to first-order logic through $\mathtt{WP}_{\mathtt{S0}\rightarrow\mathtt{S3},\mathtt{C0}\rightarrow\mathtt{C3}}$, this translates to $\mathtt{n}_S = \mathtt{n}_C \Rightarrow \mathtt{LNil} \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(0)$. Here, $\mathtt{LNil}$ is obtained for $\mathtt{l}_S$ and $\mathtt{0}$ (null) is obtained for $\mathtt{l}_C$. The one-unrolled form of this proof obligation yields $\mathtt{n}_S = \mathtt{n}_C \Rightarrow \mathit{true}$ which trivially resolves to true.

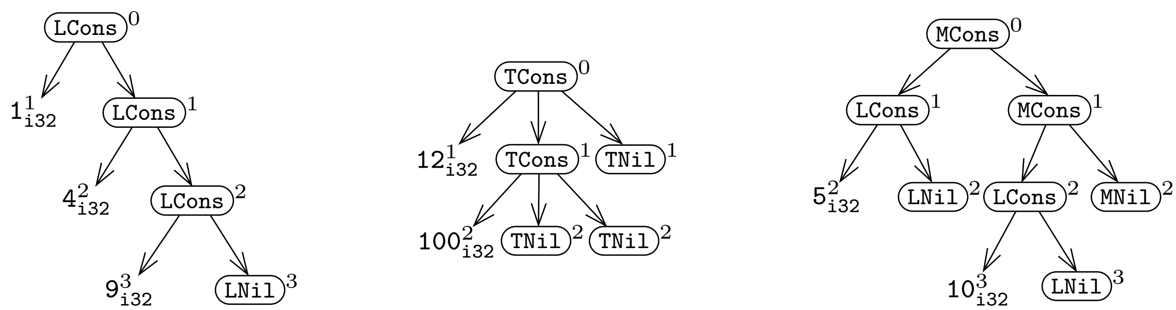
Consider the following example of a proof obligation: $\{\phi_{\mathtt{S0:C0}}\}(\mathtt{S0}\rightarrow\mathtt{S3}\rightarrow\mathtt{S5}\rightarrow\mathtt{S3}, \mathtt{C0}\rightarrow\mathtt{C3})\{\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)\}$. Notice, we have changed the path in $\mathcal{S}$ (with CFG fig. 1.3a) to $\mathtt{S0}\rightarrow\mathtt{S3}\rightarrow\mathtt{S5}\rightarrow\mathtt{S3}$ here. In this case, the corresponding first-order logic formula evaluates to: $(\mathtt{n}_S = \mathtt{n}_C) \wedge (0 <_u \mathtt{n}_S) \Rightarrow \mathtt{LCons}(0, \mathtt{LNil}) \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(0)$, where $(0 <_u \mathtt{n}_S)$ is the path condition for the path $\mathtt{S0}\rightarrow\mathtt{S3}\rightarrow\mathtt{S5}\rightarrow\mathtt{S3}$. One-unrolling of this proof obligation decomposes $\mathtt{RHS}$ into false due to failed unification of LCons and LNil. The proof obligation is further discharged using an SMT solver which provides a counterexample (model) that evaluates the formula to false. For example, the counterexample $\{\mathtt{n}_S \mapsto 42, \mathtt{n}_C \mapsto 42\}$ evaluates this formula to false. These counterexamples assist in faster convergence of our correlation search and invariant inference procedures (as we will discuss later in sections 4.2 and 4.3).

Thus for type I queries, k-unrolling reduces all (if any) recursive relations in the original proof obligation into scalar equalities. The resulting query is further discharged using an SMT solver. Section 4.4 contains a deeper analysis of the following aspects of our proof discharge algorithm: (a) translation of formula to SMT logic (section 4.4.7), and (b) reconstruction of counterexamples from models returned by the SMT solver (section 4.4.8). Assuming a capable enough SMT solver, all proof obligations in type I can be discharged precisely, i.e., we can always decide whether the proof obligation evaluates to true or false. If it evaluates to false, we also obtain counterexamples.

3.5 Handling Type II Proof Obligations

Consider the proof obligation for I2 invariant $\mathtt{sum}_S = \mathtt{sum}_C$ across edge $(\mathtt{S2{:}C2}) \rightarrow (\mathtt{S2{:}C2})$ in fig. 2.4a: $\{\phi_{\mathtt{S2:C2}}\}(\mathtt{S2}\rightarrow\mathtt{S5}\rightarrow\mathtt{S2}, \mathtt{C2}\rightarrow\mathtt{C4}\rightarrow\mathtt{C2})\{\mathtt{sum}_S = \mathtt{sum}_C\}$, where the node invariant $\phi_{\mathtt{S2:C2}}$ contains the recursive relation $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)$. The corresponding (simplified) first-order logic formula for this proof obligation is: $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C) \wedge (\mathtt{sum}_S = \mathtt{sum}_C) \wedge (\mathtt{l}_S \text{ is } \mathtt{LCons}) \wedge (\mathtt{l}_C \neq 0) \Rightarrow (\mathtt{sum}_S + \mathtt{l}_S.\mathtt{val}) = (\mathtt{sum}_C + \mathtt{l}_C \xrightarrow{\mathfrak{m}}_{\mathtt{lnode}} \mathtt{val})$. We fail to remove the recursive relation on the LHS even after k-unrolling for any finite unrolling parameter k because both sides of $\sim$ represent list values of arbitrary length. In such a scenario, we do not know of an efficient SMT encoding for the recursive relation $\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathfrak{m}}(\mathtt{l}_C)$. Ignoring this recursive relation will incorrectly (although soundly) evaluate the proof obligation to false; however, for a successful equivalence proof, we need the proof discharge algorithm to evaluate it to true. Let's call this requirement R1.

Now, consider the proof obligation formed by correlating two iterations of the loop in program $\mathcal{S}$ (with CFG fig. 2.3c) with one iteration of the loop in program $\mathcal{C}$ (with CFG fig. 2.3d):



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

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

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

Figure 3.1: Tree representation of three values, each of type `List`, `Tree` and `Matrix` respectively. The depths are shown as superscripts for each node in the trees.

$\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}\rightarrow\texttt{S5}\rightarrow\texttt{S2}, \texttt{C2}\rightarrow\texttt{C4}\rightarrow\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$. The equivalent first-order logic formula is: $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathfrak{m}}(\texttt{l}_C) \wedge (\texttt{sum}_S = \texttt{sum}_C) \wedge (\texttt{l}_S \text{ is } \texttt{LCons}) \wedge (\texttt{l}_S.\texttt{tail} \text{ is } \texttt{LCons}) \Rightarrow (\texttt{sum}_S + \texttt{l}_S.\texttt{val} + \texttt{l}_S.\texttt{tail}.\texttt{val}) = (\texttt{sum}_C + \texttt{l}_C \overset{\mathfrak{m}}{\rightarrow}_{\texttt{lnode}} \texttt{val})$. Similar to the prior proof obligation, its equivalent first-order logic formula contains a recursive relation in the `LHS`. Clearly, this proof obligation should evaluate to false. Whenever a proof obligation evaluates to false, we expect an ideal proof discharge algorithm to generate counterexamples that falsify the proof obligation. Let's call this requirement R2. Recall that these counterexamples help in faster convergence of our correlation search and invariant inference procedures.

To tackle requirements R1 and R2, our proof discharge algorithm converts the original proof obligation $P : \{\phi_s\}(e)\{\phi_d\}$ into two approximated proof obligations $(P_{pre-o} : \{\phi_s^{o_{d_1}}\}(e)\{\phi_d\})$ and $(P_{pre-u} : \{\phi_s^{u_{d_2}}\}(e)\{\phi_d\})$. Here $\phi_s^{o_{d_1}}$ and $\phi_s^{u_{d_2}}$ represent the over- and under-approximated versions of precondition ϕ_s respectively, and d_1 and d_2 represent *depth parameters* that indicate the degree of over- and under-approximation. To explain our over- and under-approximation scheme, we first introduce the notion of *depth of an ADT value*.

3.5.1 Depth of ADT Values

To define depth of an ADT value v, we view the value as a tree $\mathcal{T}(v)$. The internal nodes of $\mathcal{T}(v)$ represent ADT data constructors and the leafs (also called *terminals*) represent scalar values (e.g. bitvector literals). The depth of a data constructor or a scalar in v is simply the depth of its associated node in $\mathcal{T}(v)$. The *depth* of ADT value v is defined as the depth of $\mathcal{T}(v)$. For example, the depth of $\texttt{LCons}(1, \texttt{LCons}(4, \texttt{LNil}))$ is 2. Figure 3.1 shows the tree representation and depths for multiple ADT values.

3.5.2 Approximation of Recursive Relations

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

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

$$l_1 \approx_a l_2 \Rightarrow l_1 \approx_b l_2 \Rightarrow l_1 \sim l_2 \Rightarrow l_1 \sim_b l_2 \Rightarrow l_1 \sim_a l_2 \tag{3.3}$$

3.5.3 Reduction of Approximate Recursive Relations

Unlike the original recursive relation $l_1 \sim l_2$, its approximations $l_1 \sim_d l_2$ and $l_1 \approx_d l_2$ can be reduced into equivalent conditions absent of recursive relations. Hence, these approximations can be encoded and subsequently discharged by a SMT solver.

- $l_1 \sim_d l_2$ is equivalent to the condition that the tree structures of l_1 and l_2 are identical till depth d *and* the corresponding terminal values in both d-depth identical structures are also equal. Note that these conditions only require scalar equalities. $l_1 \sim_d l_2$ can be identified through unification of l_1 and l_2 till depth d. This algorithm is similar to the 'iterative unification and rewriting procedure' in section 3.2 and further described in section 4.4.6. In this modified unification algorithm, we eagerly expand atomic ADT expressions till depth d, whereas 'iterative unification and rewriting procedure' terminates unification whenever a correlation tuple relates (possibly ADT) atomic expressions. Finally, we only keep those correlation tuples at depth $\leq d$ that relate scalar values and discard the recursive relations.

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

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

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

$l \approx_2 \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p) \Leftrightarrow \Gamma_2(l) \wedge \Gamma_2(\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)) \wedge l \sim_2 \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$. The algorithms for reduction of over- and under-approximate recursive relations are given in section 4.4.6 respectively.

3.5.4 Summary of Type II Proof Discharge Algorithm

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

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

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

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

Similarly, the proof obligation $\{\phi_{\texttt{S2:C2}}\}(\texttt{S2}\to\texttt{S5}\to\texttt{S2}\to\texttt{S5}\to\texttt{S2}, \texttt{C2}\to\texttt{C4}\to\texttt{C2})\{\texttt{sum}_S = \texttt{sum}_C\}$ successfully evaluates to false using a depth 2 underapproximation of the precondition $\phi_{\texttt{S2:C2}}$. In the depth 2 underapproximate version, we try to prove that if the equal lists $\texttt{l}_S$ and $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$ have exactly two nodes[3], then the sum of the two values in $\texttt{l}_S$ is equal to the value stored in the first node in $\texttt{l}_C$. This proof obligation will return counterexample(s) that map program variables to their concrete values. The following is a possible counterexample to the depth 2 underapproximate proof obligation.

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

This counterexample maps variables to values (e.g., $\texttt{sum}_S$ maps to an `i32` value 3 and $\texttt{l}_S$ maps to a `List` value `LCons`(42,`LCons`(43,LNil))). It also maps the C program's memory state $\mathbb{m}$ to an array that maps the regions starting at addresses `0x123` and `0x456` (regions of size '`sizeof(lnode)`') to memory objects of type `lnode` (with the `val` and `next` fields shown for each object). All other addresses (except the ones for which an explicit mapping is available), $\mathbb{m}$ provides a default byte-value 77 (shown as `() ↦ 77`) in this counterexample.

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

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

and $\texttt{sum}_C$ are 3+42+43=88 and 3+42=45 respectively. Evidently, the counterexample falsifies the proof condition because these values are not equal (as required by the postcondition).

3.6 Handling Type III Proof Obligations

In fig. 1.4, consider a proof obligation generated across the product-CFG edge $(\texttt{S3:C5}) \rightarrow (\texttt{S3:C3})$ while checking if the (I4) invariant, $\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$, holds at (S3:C3): $\{\phi_{\texttt{S3:C5}}\}(\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3}, \texttt{C5}\rightarrow\texttt{C3})\{\texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)\}$. Here, a recursive relation is present both in the precondition $\phi_{\texttt{S3:C5}}$ ((I8)) and in the postcondition ((I4)) and we are unable to remove them after k-unrolling. When lowered to first-order logic through $\texttt{WP}_{\texttt{S3}\rightarrow\texttt{S5}\rightarrow\texttt{S3},\texttt{C5}\rightarrow\texttt{C3}}$, this translates to (showing only relevant relations):

$$\begin{aligned} (\texttt{i}_S = \texttt{i}_C \wedge \texttt{p}_C = \texttt{malloc()} \wedge \texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)) \\ \Rightarrow (\texttt{LCons}(\texttt{i}_S, \texttt{l}_S) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{p}_C)) \end{aligned} \tag{3.4}$$

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

$$\mathbb{m}' \Leftrightarrow \mathbb{m}[\texttt{addrof}(\texttt{p}_\texttt{C} \rightarrow_{\texttt{lnode}} \texttt{val}) \leftarrow \texttt{i}_C]_{\texttt{i32}}[\texttt{addrof}(\texttt{p}_\texttt{C} \rightarrow_{\texttt{lnode}} \texttt{next}) \leftarrow \texttt{l}_C]_{\texttt{i32}} \tag{3.5}$$

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

3.6.1 LHS-to-RHS Substitution and RHS Decomposition

We start by utilizing the $\sim$ relationships in the LHS (antecedent) of '⇒' to rewrite eq. (3.4) so that the ADT variables (e.g., $\texttt{l}_S$) in its RHS (consequent) are substituted with the lifted $\mathcal{C}$ values (e.g., $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$). Thus, we rewrite eq. (3.4) to:

$$\begin{aligned}(\texttt{i}_S = \texttt{i}_C \wedge \texttt{p}_C = \texttt{malloc}() \wedge \texttt{l}_S \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C))\\ \Rightarrow (\texttt{LCons}(\texttt{i}_S, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{p}_C))\end{aligned} \tag{3.6}$$

Next, we decompose the RHS by decomposing the recursive relation in the RHS followed by RHS-breaking. This process reduces eq. (3.6) into the following smaller proof obligations (LHS denotes the antecedent of the proof obligation in eq. (3.6)): (a) $\texttt{LHS} \Rightarrow (\texttt{p}_C \neq 0)$, (b) $\texttt{LHS} \wedge (\texttt{p}_C \neq 0) \Rightarrow (\texttt{i}_S = \texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{val})$, and (c) $\texttt{LHS} \wedge (\texttt{p}_C \neq 0) \Rightarrow \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{next})$.

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

1. The first proof obligation with postcondition $(\texttt{p}_C \neq 0)$ evaluates to *true* because the LHS ensures that $\texttt{p}_C$ is the return value of an allocation function (i.e. `malloc`) which must be non-zero due to the ($\mathcal{C}$ `fits`) assumption.
2. The second proof obligation with postcondition $(\texttt{i}_S = \texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{val})$ also evaluates to *true* because $\texttt{i}_C$ is written at address $\texttt{p}_C + \texttt{offsetof}(\texttt{lnode}, \texttt{val})$ in $\mathbb{m}'$ (eq. (3.5)) and the LHS ensures that $\texttt{i}_S = \texttt{i}_C$.

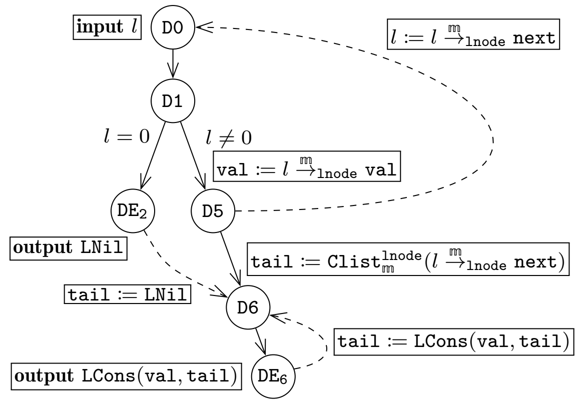
For ease of exposition, we simplify the postcondition of the third proof obligation by rewriting $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{p}_C \xrightarrow{\mathbb{m}'}_{\texttt{lnode}} \texttt{next})$ to $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C)$. This simplification is valid because $\texttt{l}_C$ is written to address $\texttt{p}_C + \texttt{offsetof}(\texttt{lnode}, \texttt{next})$ in $\mathbb{m}'$ (eq. (3.5)). Also, we have already shown that $(\texttt{p}_C \neq 0)$ holds due to the ($\mathcal{C}$ `fits`) assumption. This simplification-based rewriting is only done for ease of exposition, and has no effect on the operation of the algoritm. Thus, the third proof obligation can be rewritten as a recursive relation between two lifted expressions:

$$\texttt{LHS} \Rightarrow \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C) \tag{3.7}$$

Hence, we are interested in proving equality between two `List` values lifted from $\mathcal{C}$ values under a

```
D0: List Clist_m^lnode(i32 l) {
D1:  if l = 0:
D2:   return LNil;
D3:  else:
D4:   i32  val  := l ->m_lnode val;
D5:   List tail := Clist_m^lnode(l ->m_lnode next);
D6:   return LCons(val, tail);
DE: }
```

(a) (Abstracted) IR of Deconstruction Program



(b) CFG of Deconstruction Program

Figure 3.2: IR and CFG representation of deconstruction program based on the lifting constructor $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$ defined in eq. (2.2). The edge D5→D6 contains a recursive function call. In fig. 3.2b, the square boxes show the transfer functions for the deconstruction program. The dashed edges represent the recursive function call in the CFG representation as shown in fig. 3.2b.

precondition. Next, we show how the above can be reposed as the problem of showing equivalence between two procedures through bisimulation.

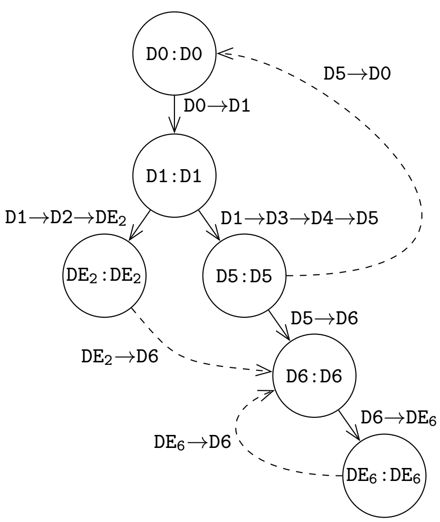
3.6.2 Deconstruction Programs for Lifted Values

Consider a program that recursively calls the definition (i.e. unrolling procedure) of $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$ (eq. (2.2)) to deconstruct $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l)$. For example, $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l)$ may yield a recursive call to $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{next})$ and so on, until the argument becomes zero. This program essentially deconstructs $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l)$ into its terminal (scalar) values and reconstructs a `List` value equal to the value represented by $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l)$. We call this program a *deconstruction program* based on the lifting constructor $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$. Figure 3.2 show the IR and CFG representations of the deconstruction program for the lifting constructor $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$.

Theorem 1. *Under an antecedent* LHS, $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C)$ *holds* if and only if *the two deconstruction programs $\mathcal{D}_1$ and $\mathcal{D}_2$, based on $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\texttt{l}_C)$ and $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}'}(\texttt{l}_C)$, are* *equivalent. The equivalence must ensure that the observables generated by both programs (i.e. output* `List` *values) are equal, given the that input* $\mathtt{l}_C$ *is provided to both programs respectively* and *the antecedent* `LHS` *holds at the program entries.*

Proof Sketch. The proof follows from noting that the only observables of $\mathcal{D}_1$ and $\mathcal{D}_2$ are their output `List` values. Also, the value represented by a lifted expression is equal to the output of its deconstruction program. Thus, a successful equivalence proof ensures equal values represented by the lifting constructors and vice versa. □

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



(a) Decons-PCFG

PC-Pair	**Invariants**
(D0:D0)	(P) $\mathtt{l}^{fst} = \mathtt{l}^{snd}$
(D1:D1)	(I1) $\mathtt{l}^{fst} = \mathtt{l}^{snd}$
(D5:D5)	(I2) $\mathtt{val}^{fst} = \mathtt{val}^{snd}$ (I3) $\mathtt{l}^{fst} \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{next} = \mathtt{l}^{snd} \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{next}$
(D6:D6)	(I4) $\mathtt{val}^{fst} = \mathtt{val}^{snd}$ (I5) $\mathtt{tail}^{fst} = \mathtt{tail}^{snd}$
(DE$_2$:DE$_2$) (DE$_6$:DE$_6$)	(E) $\mathtt{ret}^{fst} = \mathtt{ret}^{snd}$

(b) Invariants Table

Figure 3.3: Decons-PCFG and invariants table for the deconstruction programs of $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$ and $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C)$ respectively.

3.6.3 Checking Bisimulation between Deconstruction Programs

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

The decons-PCFG for the proof obligation in eq. (3.7) is shown in fig. 3.3a. We distinguish states between the first and second programs using superscripts: fst and snd respectively. However, these are omitted in case the states are equal in both programs (e.g., $\mathtt{p}_C$). To check bisimulation between the programs that deconstruct $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$ and $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C)$ (i.e. $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ respectively), the decons-PCFG correlates one unrolling of the first program with one unrolling of the second program, as defined by the unrolling procedure in eq. (2.2). Thus, the PC-transition correlations of $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$ are trivially obtained by unifying the static program structures as shown in fig. 3.3a. A node is created in the decons-PCFG that encodes the correlation of the entries of both programs; we call this node the *recursive-node* in the decons-PCFG (e.g., (D0:D0) in fig. 3.3a). A recursive call becomes a back-edge in the decons-PCFG that terminates at the recursive-node. Furthermore, a bisimulation check involves identification of invariants at correlated PC-pairs strong enough to ensure observable equivalence. At the start of both deconstruction programs, (P) $\mathtt{l}^{fst} = \mathtt{l}^{snd} = \mathtt{l}_C$ — the same $\mathtt{l}_C$ is passed to both $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$, only the memory states $\mathbb{m}$ and $\mathbb{m}'$ (defined in eq. (3.5)) are different. The observables include the returned `List` values at correlated program exits ($\mathtt{DE}_2$:$\mathtt{DE}_2$) and ($\mathtt{DE}_6$:$\mathtt{DE}_6$), which forms the postcondition (labeled (E) in fig. 3.3b). Next, the bisimulation check involves identification of inductive invariants (labeled (I) in fig. 3.3b) at correlated PC-pairs. The proof obligations arising due to this bisimulation check include:

1. The $\underline{\mathtt{if}}$ condition ($\mathtt{l}^{fst} = 0$) in $\mathcal{D}^{fst}$ is equal to the corresponding $\underline{\mathtt{if}}$ condition ($\mathtt{l}^{snd} = 0$) in $\mathcal{D}^{snd}$: ($\mathtt{l}^{fst} = 0$) = ($\mathtt{l}^{snd} = 0$).
2. If the $\underline{\mathtt{if}}$ condition evaluates to false in both $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$, then observable values $\mathtt{val}^{fst}$ and $\mathtt{val}^{snd}$ along the path (D1:D1)$\rightarrow$(D5:D5) (used in the construction of the output lists)

are equal. This forms the invariant (I2) in fig. 3.3b and lowers to the following proof obligation:
$(\mathtt{l}^{fst} \neq 0) \wedge (\mathtt{l}^{snd} \neq 0) \Rightarrow \mathtt{l}^{fst} \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{val} = \mathtt{l}^{snd} \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val}$.

3. If the <u>`if`</u> condition evaluates to false in both $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$, then the preconditions are satisfied at the beginning of the programs invoked through the recursive call. This involves checking that, along the path (`D1`:`D1`)$\rightarrow$(`D5`:`D5`), the actual arguments to the recursive call satisfies the precondition (P) at the beginning of the procedure i.e. the recursive-node (`D0`:`D0`). This forms the invariant (I3) in fig. 3.3b and lowers the following proof obligation:
$(\mathtt{l}^{fst} \neq 0) \wedge (\mathtt{l}^{snd} \neq 0) \Rightarrow \mathtt{l}^{fst} \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{next} = \mathtt{l}^{snd} \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{next}$.

   A successful discharge of the above invariant ((I3)), by induction, ensures that postcondition ((E)) is satisfied by the values returned by the recursive call at product-CFG node (`D6`:`D6`). Hence, we can assume that invariant (I5) holds at (`D6`:`D6`). This special case of correlating procedure call edges is further discussed in section 4.2.3 as part of our overall product-CFG construction algorithm.

The first check succeeds due to the precondition (P) $\mathtt{l}^{fst} = \mathtt{l}^{snd}$ at the recursive-node. For the second and third checks, we additionally need to reason that the memory objects $\mathtt{l}^{snd} \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{l}^{snd} \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{next}$ cannot alias with the writes in $\mathbb{m}'$ (eq. (3.5)) to the newly allocated objects $\mathtt{p}_C \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val}$ and $\mathtt{p}_C \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{next}$. We capture this aliasing information using a points-to analysis described next in section 3.6.4.

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

3.6.4 Points-to Analysis

To reason about aliasing (as required during bisimulation check in section 2.4), we conservatively compute *may-point-to* information for each program value using an interprocedural flow-sensitive version of Andersen's algorithm [10]. The range of this computed may-point-to function is the set of *region labels*, where each region label identifies a set of memory objects. The sets of memory objects identified by two distinct region labels are necessarily disjoint. We write $p \rightsquigarrow \{\mathtt{R}_1, \mathtt{R}_2\}$ to represent the condition that value *p may point to* an object belonging to one of the region labels $\mathtt{R}_1$ or $\mathtt{R}_2$ (but may not point to any object outside of $\mathtt{R}_1$ and $\mathtt{R}_2$).

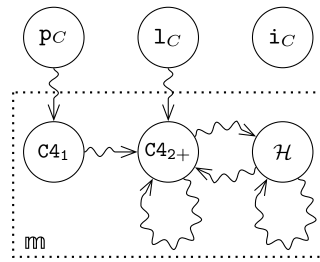
We populate the set of all region labels using *allocation sites* of the $\mathcal{C}$ program i.e., PCs where a call to `malloc` occurs. For example, `C4` in fig. 1.2b is an allocation site. For each allocation site A, we create two region labels: (a) the first region label, called A_1, identifies the set of memory objects that were allocated by the most recent execution of A, and (b) the second region label, called A_{2+}, identifies the set of memory objects that were allocated by older (not the most recent) executions of A. We also include a special heap region, $\mathcal{H}$ to represent the rest of the memory not covered by the allocation site regions.

For example, at the start of PC `C7` in fig. 1.2b, $\mathtt{i}_C \rightsquigarrow \emptyset$, $\mathtt{p}_C \rightsquigarrow \{\mathtt{C4}_1\}$, and $\mathtt{l}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$. Since the may-point-to analysis determines the sets of objects pointed-to by $\mathtt{p}_C$ and $\mathtt{l}_C$ to be disjoint, ($\mathtt{C4}_1$ against $\mathtt{C4}_{2+}$), any memory accessed through $\mathtt{p}_C$ and $\mathtt{l}_C$ cannot alias at `C7` (for accesses within the bounds of the allocated objects).

The may-point-to information is computed not just for program values (e.g., $\mathtt{p}_C$, $\mathtt{l}_C$) but also for each region label. For region labels `R1`, `R2` and `R3`: $\mathtt{R1} \rightsquigarrow \{\mathtt{R2}, \mathtt{R3}\}$ represents the condition that the values (pointers) stored in objects identified by `R1` may point to objects identified by either `R2` or `R3` (but not to any other object outside `R2` and `R3`). In fig. 1.2b, at PC `C7`, we get $\mathtt{C4}_1 \rightsquigarrow \{\mathtt{C4}_{2+}\}$ and $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$. The condition $\mathtt{C4}_1 \rightsquigarrow \{\mathtt{C4}_{2+}\}$ holds because the `next` pointer of the object pointed-to by $\mathtt{p}_C$ (which is a $\mathtt{C4}_1$ object at `C7`) may point to a $\mathtt{C4}_{2+}$ object (e.g., object pointed to by $\mathtt{l}_C$). On the other hand, pointers within a $\mathtt{C4}_{2+}$ object may not point to a $\mathtt{C4}_1$ object.

Points-to Invariants	
J1 $\texttt{p}_C \rightsquigarrow \{\texttt{C4}_1\}$	J2 $\texttt{C4}_1 \rightsquigarrow \{\texttt{C4}_{2+}\}$
J3 $\texttt{l}_C \rightsquigarrow \{\texttt{C4}_{2+}\}$	J4 $\texttt{C4}_{2+} \rightsquigarrow \{\texttt{C4}_{2+}, \mathcal{H}\}$
J5 $\texttt{i}_C \rightsquigarrow \emptyset$	J6 $\mathcal{H} \rightsquigarrow \{\texttt{C4}_{2+}, \mathcal{H}\}$

(a) Points-to Invariants at `C5` of fig. 1.2b



(b) Points-to Graph at `C5` of fig. 1.2b

Figure 3.4: Points-to Invariants and Points-to Graph at `C5` of fig. 1.2b

3.6.5 Transferring Points-to Information to Decons-PCFG

Recall that in section 3.6.3, we reduce the condition $\texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}}(\texttt{l}_C) \sim \texttt{Clist}^{\texttt{lnode}}_{\mathtt{m}'}(\texttt{l}_C)$ to an equivalence check between their deconstruction programs: $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$. Also, recall that we discharge the equivalence check through construction of a decons-PCFG encoding the lockstep execution of the two deconstruction programs. During this bisimulation check, we need to prove that, $\texttt{l}^{fst} \overset{\mathtt{m}}{\rightarrow}_{\texttt{lnode}} \{\texttt{val}, \texttt{next}\}$ and $\texttt{l}^{snd} \overset{\mathtt{m}'}{\rightarrow}_{\texttt{lnode}} \{\texttt{val}, \texttt{next}\}$ are equal. Recall that the invariant I1 (in fig. 3.3b) asserts $\texttt{l}^{fst} = \texttt{l}^{snd}(\triangleq \texttt{l})$. Thus, to successfully discharge these proof obligations, it suffices to show that `l` cannot alias with the memory writes that distinguish $\mathtt{m}$ and $\mathtt{m}'$.

Figure 3.4a shows the may-point-to relations identified by our points-to analysis on the $\mathcal{C}$ in fig. 1.2b at the program point `C5`. The points-to analysis determines that at `C5` (i.e. start of the product-CFG edge (`S3`:`C5`)→(`S3`:`C3`) across which the proof obligation is generated), the pointer to the *head* of the list, i.e. J3 $\texttt{l}_C \rightsquigarrow \{\texttt{C4}_{2+}\}$. It also determines that the distinguishing writes (in eq. (3.5)) modify memory regions belonging to $\texttt{C4}_1$ only (i.e. J1). Further, we get J4 $\texttt{C4}_{2+} \rightsquigarrow \{\texttt{C4}_{2+}, \mathcal{H}\}$ at PC `C5`. Figure 3.4b shows the points-to graph for the $\mathcal{C}$ variables and the memory regions (in $\mathtt{m}$). This graphical representation clearly illustrates that the objects pointed to by $\texttt{p}_C$ (i.e. $\texttt{C4}_1$) and $\texttt{l}_C$ (i.e. $\texttt{C4}_{2+}$) are mutually isolated.

However, notice that these determinations only rule out aliasing of the list-head with the distinguishing writes. We also need to confirm non-aliasing of the internal nodes of the linked list with the distinguishing writes. For this, we need to identify a points-to invariant: $\mathtt{l}^{snd} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$, at the recursive-node of the decons-PCFG (shown in fig. 3.3a). To identify such points-to invariants, we run our points-to analysis on the deconstruction programs (i.e. $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$) before comparing them for equivalence. To model procedure calls, A *supergraph* is created with edges representing control flow to (and from) the entry (and exits) of the program respectively (e.g., dashes edges in fig. 3.2b). To see why $\mathtt{l}^{snd} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$ is an inductive invariant at ($\mathtt{D0}$:$\mathtt{D0}$):

> (Base case) the invariant holds at entry of the decons-PCFG because $\mathtt{l}^{snd} = \mathtt{l}_C$ at entry and (J3) $\mathtt{l}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$, which is a stronger condition.
>
> (Inductive step) if $\mathtt{l}^{snd} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$ holds at the entry node, it also holds at the start of a recursive call. This follows from (J4) $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$ and (J5) $\mathcal{H} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$ (points-to information at PC $\mathtt{C5}$), which ensures that $\mathtt{l}_C \xrightarrow{\mathtt{m}'}_{\mathtt{lnode}} \mathtt{next}$ may point to only $\mathtt{C4}_{2+}$ and $\mathcal{H}$ objects.

The same analysis is run for both $\mathcal{C}$, and the deconstruction programs $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$. For a deconstruction program $\mathcal{D}$, the boundary condition (at entry) for the points-to analysis is based on the results of the points-to analysis on $\mathcal{C}$ at the PC where the proof obligation is being discharged. For example, the points-to information of $\mathcal{C}$ PC $\mathtt{C5}$ (in fig. 1.2b) is used during the points-to analysis on $\mathcal{D}^{fst}$ and $\mathcal{D}^{snd}$.

During proof query discharge, the points-to invariants are encoded as SMT constraints. This allows us to complete the bisimulation proof on the decons-PCFG in fig. 3.3a, and consequently, successfully discharge the proof obligation $\{\phi_{\mathtt{S3:C5}}\}(\mathtt{S3}\rightarrow\mathtt{S5}\rightarrow\mathtt{S3}, \mathtt{C5}\rightarrow\mathtt{C3})\{\mathtt{l}_S \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(\mathtt{l}_C)\}$ in table 1.1. The points-to analysis is further discussed in section 4.1.

3.6.6 Summary of Type III Proof Discharge Algorithm

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

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

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

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

Function $Prove(\{\phi_s\}(e)\{\phi_d\}, k, d_o, d_u)$
    $F \hookleftarrow LowerToFOL(\{\phi_s\}(e)\{\phi_d\})$;
    foreach $\texttt{LHS} \Rightarrow \texttt{RHS}_i$ **in** $RHSBreak(F)$ **do**
        if $Solve(\texttt{LHS}, \texttt{RHS}_i, k, d_o, d_u) = \texttt{False}(\Gamma)$ **then**
            return $\texttt{False}(\Gamma)$;
        end
    end
    return $\texttt{True}$;
end
Function $Solve(\texttt{LHS}, \texttt{RHS}, k, d_o, d_u)$
    $(\texttt{LHS}_k, \texttt{RHS}_k) \hookleftarrow DecomposeAndUnroll(\texttt{LHS}, \texttt{RHS}, k)$;
    switch $Categorize(\texttt{LHS}_k, \texttt{RHS}_k)$ **do**
        case `Type I` **do**
            return $SMTProve(\texttt{LHS}_k \Rightarrow \texttt{RHS}_k)$;
        case `Type II` **do**
            $\texttt{LHS}_o \hookleftarrow Overapproximate(\texttt{LHS}, d_o)$;
            if $SMTProve(\texttt{LHS}_o \Rightarrow \texttt{RHS}_k) = \texttt{True}$ **then**
                return $\texttt{True}$;
            end
            $\texttt{LHS}_u \hookleftarrow Underapproximate(\texttt{LHS}, d_u)$;
            if $SMTProve(\texttt{LHS}_u \Rightarrow \texttt{RHS}_k) = \texttt{False}(\Gamma)$ **then**
                return $\texttt{False}(\Gamma)$;
            end
            return $\texttt{False}(\emptyset)$;
        case `Type III` **do**
            $\texttt{RHS}' \hookleftarrow RewriteRHSUsingLHS(\texttt{LHS}, \texttt{RHS})$;
            foreach $\texttt{P}_i \Rightarrow \texttt{RHS}_i$ **in** $DecomposeAndRHSBreak(\texttt{RHS}')$ **do**
                if $\texttt{RHS}_i = l_1 \sim l_2$ **then**
                    $(\mathcal{D}_1, \mathcal{D}_2) \hookleftarrow GetDeconstructionPrograms(l_1, l_2)$;
                    if $CheckEquivalence(\texttt{LHS} \wedge \texttt{P}_i, \mathcal{D}_1, \mathcal{D}_2) = \texttt{False}$ **then**
                        return $\texttt{False}(\emptyset)$;
                    end
                else if $Solve(\texttt{LHS} \wedge \texttt{P}_i, \texttt{RHS}_i, k, d_o, d_u) = \texttt{False}(\Gamma)$ **then**
                    return $\texttt{False}(\Gamma)$;
                end
            end
            return $\texttt{True}$;
        end
    end
end

Algorithm 1: Summary of the Proof Discharge Algorithm

3.7 Overview of Proof Discharge Algorithm

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

$Solve()$ is responsible for discharging these smaller queries. Inputs include `LHS`, `RHS` (representing the proof obligation P : `LHS` $\Rightarrow$ `RHS`); along with the parameters: k, d_o and d_u. $Solve()$ begins by finding the k-unrolled form of P and categorizes it into one of the three types. As discussed in section 3.4, we simply discharge a type I query using SMT solvers (through $SMTProve()$). $SMTProve()$ is responsible for (a) translating the input formula (absent of recursive relations) to SMT logic, (b) discharging SMT solver queries, and (c) reconstruction of counterexamples from the models returned by the SMT solvers. The steps (a) and (c) are further explored in sections 4.4.7 and 4.4.8 respectively. As summarized in section 3.5.4, for a type II query, we attempt to prove its overapproximate version first. In case of a failure, we attempt to disprove its underapproximate version (and generate counterexamples). If both attempts fail, we *soundly* return `False` (without counterexamples). Lastly, a type III query P is discharged as detailed in section 3.6.6. In brief, we decompose and perform `RHS`-breaking on P. This results in smaller proof obligations; ones without a recursive relation in its `RHS`, are type II queries and discharged through a recursive call to $Solve()$. For those containing a recursive relation $l_1 \sim l_2$ in their `RHS`, we reformulate the query as an equivalence check between the deconstruction program of l_1 and l_2 respectively. If any one of these queries fail, we immediately return `False` with the counterexamples (if available). Otherwise, we have successfully proven a type III query and return `True`.

Chapter 4

Spec-to-C Equivalence Checker

This chapter presents our automatic equivalence checker algorithm S2C. Given a Spec and a C program along with the input-output specification for each function-pair, S2C searches for a proof of equivalence between the CFGs of each Spec and C procedures: $\mathcal{S}$ and $\mathcal{C}$. Recall that CFGs represent deterministic programs and evidently, the C procedure is determinized during conversion to $\mathcal{C}$. Hence, S2C checks equivalence between a Spec procedure and the determinized C procedure. A translation validator (such as the Counter tool [26]) can be used to check equivalence between the same determinized C procedure against its generated assembly. By restricting the deterministic choices made by the C compiler to the same choices made during construction of $\mathcal{C}$, we can effectively establish end-to-end equivalence between a Spec procedure against its assembly. We start with a dataflow formulation of the points-to analysis used as part of S2C on $\mathcal{C}$ as well as deconstruction programs during discharge of type III proof obligations in section 4.1. As stated in section 1.2, S2C is based on three primary interdependent algortihms: A1 an algorithm to incrementally construct a product-CFG by correlating program executions across $\mathcal{S}$ and $\mathcal{C}$, A2 an algorithm to identify inductive invariants at intermediate PCs in the (partially constructed) product-CFG, and A3 an algorithm for solving proof obligations generated by A1 and A2 algorithms. We describe our counterexample-guided best-first search algorithm for construction of a product-CFG (A1) in section 4.2. This is followed by a dataflow formulation of our counterexample-guided invariant inference algorithm (A2) in section 4.3. In the last chapter, we walked through our proof discharge algorithm ((A3)) using examples, ending with the pseudo-code of proof discharge algorithm (algorithm 1 in section 3.7). In this chapter, we present pseudo-codes of the major subprocedures used as part of the proof discharge algorithm[1].

Table 4.1: Dataflow Formulation of the Points-to Analysis

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

4.1 Points-to Analysis

Recall that in section 3.6.3, we needed to reason about aliasing to successfully discharge a type III proof obligation. These aliasing relationships are described in section 3.6.4 and used in section 3.6.5 to successfully discharge a type III proof obligation. A points-to analysis is used to identify these relationships in $\mathcal{C}$ as well as each deconstruction program $\mathcal{D}$. Table 4.1 presents a dataflow formulation of our points-to analysis. We start by identifying the set $\mathbb{R}$ of all region labels representing mutually non-overlapping regions of the $\mathcal{C}$ memory $\mathtt{m}$. For each call to `malloc()` at PC A, we add A_1 and A_{2+} to $\mathbb{R}$. Recall that A_1 represents the region of memory returned by the *most recent* execution of A. A_{2+} represents the region of memory returned by older (i.e. all but most recent) executions of A. $\mathbb{R} = \bigcup_A \{A_1, A_{2+}\} \cup \{\mathcal{H}\}$, where $\mathcal{H}$ is the region of memory $\mathtt{m}$ not covered by the labels associated with allocation sites. Note that $\mathbb{R}$ is computed

[1] TODO:keep the full proof discharge algo here or keep it at the end of examples where it is now?

globally i.e. in case C consists of multiple procedures, $\mathbb{R}$ is identical for each and contains regions associated with allocation sites of `malloc()` calls for all procedures.

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

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

Next, we discuss the transfer function f_e for our points-to analysis. For an IR instruction $\mathtt{x} \coloneqq \mathtt{c}$, for constant $\mathtt{c}$, the transfer function updates $\Delta(\mathtt{x}) \coloneqq \emptyset$. For instruction $\mathtt{x} \coloneqq \mathtt{y}\ \mathtt{op}\ \mathtt{z}$ (for some arithmetic or logical operator $\mathtt{op}$), we update $\Delta(\mathtt{x}) \coloneqq \Delta(\mathtt{y}) \cup \Delta(\mathtt{z})$. For a load instruction $\mathtt{x} \coloneqq \mathbb{m}[y]_{\mathtt{T}}$, we update $\Delta(\mathtt{x}) \coloneqq \bigcup_{t \in \Delta(y)} \Delta(t)$. For a store instruction $\mathbb{m} \coloneqq \mathbb{m}[x \leftarrow y]_{\mathtt{T}}$, for all $t \in \Delta(\mathtt{x})$, we update $\Delta(t) \coloneqq \Delta(t) \cup \Delta(y)$. For a malloc instruction $\mathtt{x} \coloneqq \mathtt{malloc}_A()$ (where A represents the allocation site), we perform the following steps (in order):

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

2. Update $\Delta(\mathtt{x}) \coloneqq \{A_1\}$.

3. Update $\Delta(A_{2+}) \coloneqq \Delta(A_{2+}) \cup \Delta(A_1)$.

4. Update $\Delta(A_1) \coloneqq \emptyset$.

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

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

For function calls, a *supergraph* is created by adding control flow edges from the call-site to the procedure head (copying actual arguments to the formal arguments) and from the procedure exit to the program point just after the call-site (copying returned value to the variable assigned at the callsite), e.g., in fig. 3.2b, the dashed edges represent supergraph edges.

The allocation-site abstraction (with a bounded-depth call stack) is known to be effective at disambiguating memory regions belonging to different data structures [28, 15, 11]. In our work, we also need to reason about non-aliasing of the most-recently allocated object (through a `malloc` call) and the previously-allocated objects (as in the `List` construction example). The coarse-grained $\{1, 2+\}$ categorization of allocation recency is effective for such disambiguation.

4.2 Counterexample-guided Product-CFG Construction

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

Anchor nodes are identified in $\mathcal{S}$ and $\mathcal{C}$, and represents the CFG nodes (i.e. IR PCs) being considered for correlation. The algorithm ensures that every cycle in both $\mathcal{S}$ and $\mathcal{C}$ contains at least one anchor node. The start and exit nodes are always anchor nodes. Also, for every function call, the nodes just before and after its callsite are considered anchor nodes. For example, in fig. 1.3b, `C4` and `C5` are anchor nodes around the call to `malloc`. The selected anchor nodes for the CFGs in figs. 1.3a and 1.3b are: $\{\texttt{S0}, \texttt{S3}, \texttt{SE}\}$ and $\{\texttt{C0}, \texttt{C3}, \texttt{C4}, \texttt{C5}, \texttt{CE}\}$ respectively. For each anchor node in $\mathcal{C}$, our search algorithm searches for a correlated anchor node in $\mathcal{S}$ — if a (partially constructed) product-CFG π contains a product-CFG node $(n_S\!:\!n_C)$, then π correlates node n_C in $\mathcal{C}$ with node n_S in $\mathcal{S}$. The search procedure begins with a single partially-constructed product-CFG π_{init}. π_{init} contains exactly one node (`S0:C0`) that encodes the correlation of the entry nodes (i.e. `S0` and `C0`) of $\mathcal{S}$ and $\mathcal{C}$.

4.2.1 Correlating Pathsets

At each step of the incremental construction process, a node $(n_S\!:\!n_C)$ is chosen in a product-CFG π and a 1-*pathset* ξ_C in $\mathcal{C}$ starting at n_C (and ending at an anchor node in $\mathcal{C}$) is selected. Then, we enumerate potentially correlated μ-*pathsets* in $\mathcal{S}$ for the pathset ξ_C in $\mathcal{C}$. A *pathset* ξ is essentially a set of paths with the following additional properties: (a) all paths $\rho \in \xi$ begin at the same node and terminate at the same node, (b) all paths $\rho \in \xi$ are mutually-exclusive i.e. *at most* one of $\texttt{pathcond}_\rho$ can be true. A μ-pathset ξ is a pathset where each path $\rho \in \xi$ contains *at most* μ occurrences of any node. μ-pathsets are based on earlier work on the Counter tool [26][2] and helps improve the completeness of the bisimulation search by attempting to correlate a set of $\mathcal{C}$ paths with a set of $\mathcal{S}$ paths, where individual paths of $\mathcal{C}$ and $\mathcal{S}$ may be uncorrelated. An example of such a scenario is depicted in section 5.1.1. Section 4.4.1 briefly discusses the techniques of handling Hoare triples (i.e. proof obligations) involving pathsets.

Revisiting our incremental construction process, recall that we choose a 1-pathset ξ_C in $\mathcal{C}$ and enumerate potentially correlated μ-pathsets ξ_S in $\mathcal{S}$. Hence, μ represents the maximum number of iterations of a loop (in $\mathcal{S}$), that may be correlated with a pathset in $\mathcal{C}$ consisting of acyclic paths.

[2]Counter tool [26] considers (μ, δ)-unrolled pathsets in its correlation search algorithm. Our μ-pathsets are a special case of (μ, δ)-unrolled pathsets where $\delta = \mu$.

μ is a fixed parameter of the S2C algorithm and is called the *unroll factor*. For example, during construction of the product-CFG shown in fig. 1.4, say we select the product-CFG node (`S3:C3`). We choose the $\mathcal{C}$ path(set) `C3`$\rightarrow$`C4` and enumerate its potential correlations (i.e. μ-pathsets in $\mathcal{S}$ starting at `S3`): ϵ, `S3`$\rightarrow$`S5`$\rightarrow$`S3`, ..., `S3`$\rightarrow$(`S5`$\rightarrow$`S3`)$^{(\mu-1)}$. Importantly, for pathsets ξ_S (in $\mathcal{S}$) and ξ_C (in $\mathcal{C}$) to be considered for correlation, they must originate and terminate at anchor nodes, i.e. the path `S3`$\rightarrow$`S5` is skipped during enumeration. Moreover, the pathset ξ_C may only contain anchor nodes as its source and destination. Hence, the path `C3`$\rightarrow$`C4`$\rightarrow$`C5` is not considered for ξ_C, instead we attempt to correlate the subpaths `C3`$\rightarrow$`C4` and `C4`$\rightarrow$`C5` individually.

For each enumerated correlation possibility (ξ_S, ξ_C), a separate product-CFG π' is created (by cloning π) and a new product-CFG edge $e = (\xi_S, \xi_C)$ 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 pathsets ξ_S and ξ_C. For example, the node (`S3:C4`) is added to the product-CFG if it correlates pathsets ϵ and `C3`$\rightarrow$`C4` starting at (`S3:C3`). Recall that, we consider ϵ as a candidate for ξ_S, but not ξ_C. The algorithm ensures that no cycle in $\mathcal{C}$ is correlated with ϵ in $\mathcal{S}$ (to preserve divergence discussed in section 2.4). For each node n in a product-CFG π, we maintain a small number of concrete machine state pairs (of $\mathcal{S}$ and $\mathcal{C}$). The concrete machine state pairs at n are obtained as counterexamples to an unsucessful proof obligation $\{\phi_s\}(s \rightarrow d)\{\phi_d\}$ (for some edge $s \rightarrow d$ and node d in π). Thus, by construction, these counterexamples represent concrete state pairs that may potentially occur at n during the lockstep execution encoded by π.

4.2.2 Best-First Ranking of Partial Product-CFGs

To evaluate the promise of a possible correlation (ξ_S, ξ_C) starting at node n in product-CFG π, we examine the execution behaviour of the counterexamples at n on the product-CFG edge $e = (s \rightarrow d) = (\xi_S, \xi_C)$. 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 [26]. A best-first search (BFS) procedure based on this ranking criterion is used to incrementally construct a product-CFG (starting from π_{init}). For each intermediate candidate product-CFG π generated during this search procedure, an automatic invariant inference procedure (discussed next in section 4.3) is used to identify invariants at all the nodes in π. The counterexamples obtained from the proof obligations generated by this invariant inference procedure are added to the respective nodes in π; these counterexamples help rank future correlations starting at those nodes.

If after invariant inference, we realize that an intermediate candidate product-CFG π_1 is not promising enough, we backtrack and choose another candidate product-CFG π_2 and explore the potential correlations that can be added to π_2. Thus, a product-CFG is constructed one edge at a time. If at any stage, a product-CFG π contains correlations for every path in $\mathcal{C}$ and invariants ensure equal observables (i.e. *Post* holds at correlated exit nodes), we have successfully shown equivalence. This counterexample-guided BFS procedure is similar to the one described in prior work on the Counter algorithm [26].

4.2.3 Correlation in the Presence of Function Calls

Recall that $\mathcal{S}$ and $\mathcal{C}$ may make function calls (including self calls), e.g., allocation of memory in C, recursive traversal of a tree data structure. Recall that the nodes just before and after a function call are always considered anchor nodes. Calls to memory allocation functions in $\mathcal{C}$ (i.e. `malloc`) are handled by correlating the function call edge with the empty path (ϵ) in $\mathcal{S}$. For example, in the product-CFG shown in fig. 1.4, the `malloc` edge `C4`$\rightarrow$`C5` in $\mathcal{C}$ is correlated with ϵ in $\mathcal{S}$.

For all other calls, our correlation algorithm (in section 4.2) ensures that the anchor nodes around such a callsite are correlated one-to-one across both procedures. For example, let there be a call to procedure δ in $\mathcal{S}$ at PC n_S, i.e. n_S is the call-site. Let us denote the program point just after this call-site as n'_S. Let $\texttt{args}_{n_S}$ represent the values of the actual arguments of this function call (at n_S). Let $\texttt{ret}_{n'_S}$ represent the value returned by this function call (at n'_S). Similarly, for a procedure call δ in $\mathcal{C}$, let n_C, n'_C, $\texttt{args}_{n_C}$ and $\texttt{ret}_{n'_C}$ represent the function call-site, program point just after the call-site, the values of the actual arguments and the value returned respectively. Our algorithm ensures that the only correlation possible in a product-CFG π for these program points are $(n_S : n_C)$ and $(n'_S : n'_C)$.

Table 4.2: Dataflow Formulation of the Invariant Inference Algorithm

Domain	$\left\{ (\phi_n, \Gamma_n) \;\middle|\; \begin{array}{c} \phi_n \text{ is a conjunction of predicates drawn from} \\ \mathbb{G} \text{ (in fig. 4.1b)}, \Gamma_n \text{ is a set of counterexamples} \end{array} \right\}$
Direction	Forward
Boundary Condition	(ϕ_n, Γ_n) for start node : $\phi_n \leftarrow Pre, \quad \Gamma_n \leftarrow \emptyset$
Initialization to $\top$	(ϕ_n, Γ_n) for non-start nodes : $\phi_n \leftarrow \texttt{false}, \quad \Gamma_n \leftarrow \emptyset$
Transfer function across edge $e = (s \to d)$	$(\phi_d, \Gamma_d) = f_e(\phi_s, \Gamma_s)$ (shown in fig. 4.1a)
Meet operator $\otimes$	$(\phi_n, \Gamma_n) \leftarrow (\phi_n^1, \Gamma_n^1) \otimes (\phi_n^2, \Gamma_n^2)$ $\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2, \quad \phi_n \leftarrow StrongestInvCover(\Gamma_n)$

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

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

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

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

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

Figure 4.1: Transfer function f_e and Predicate grammar $\mathbb{G}$ for invariant inference dataflow analysis in table 4.2. Given invariants ϕ_s and counterexamples Γ_s at node s, f_e returns the updated invariants ϕ_d and counterexamples Γ_d at node d. $StrongestInvCover(\Gamma)$ computes the strongest invariant cover for counterexamples Γ. $\mathtt{exec}_e(\Gamma)$ (concretely) executes counterexamples Γ over edge e. $Prove(P)$ (in algorithm 1) discharges the proof obligation P, and returns either `True` or $\mathtt{False}(\Gamma)$.

4.3 Invariant Inference and Counterexample Generation

We formulate our counterexample-guided invariant inference algorithm as a dataflow analysis as shown in table 4.2. The invariant inference procedure is responsible for inferring invariants ϕ_n at each intermediate node n of a (partially constructed) product-CFG, while also generating a set of counterexamples Γ_n that represents the potential concrete machine states at n.

Given the invariants and counterexamples at node s: (ϕ_s, Γ_s), the transfer function initializes the new candidate set of counterexamples at d (Γ_d^{can}) with the current set of counterexamples at d (Γ_d) *union*-ed with the counterexamples obtained by executing Γ_s on edge e (through $\mathtt{exec}_e$). The candidate invariant at d (ϕ_d^{can}) is computed as the strongest cover of Γ_d^{can} ($StrongestInvCover()$). At each step, the transfer function attempts to prove $\{\phi_s\}(e)\{\phi_d^{can}\}$ (through a call to $Prove()$). If the proof succeeds ($Prove()$ returns `True`), the candidate invariant ϕ_d^{can} is returned along with the counterexamples Γ_d^{can} learned so far. Otherwise, $Prove()$ returns $\mathtt{False}(\gamma_s)$. The candidate invariant ϕ_d^{can} is weakened using the counterexamples obtained (i.e. γ_s) and the proof attempt is repeated.

The candidate invariants are drawn from the predicate grammar $\mathbb{G}$ shown in fig. 4.1b. In addition to affine and inequality relations between bitvectors in $\mathcal{S}$ and $\mathcal{C}$, $\mathbb{G}$ supports recursive relations between an ADT variable in $\mathcal{S}$ and a lifted expression in $\mathcal{C}$. The candidate lifting constructors of the form $\texttt{Clift}^{\texttt{T}}_{\mathtt{m}}$ (where $\mathtt{m}$ is the current memory state in $\mathcal{C}$) are derived from the lifting constructors present in the precondition *Pre* and the postcondition *Post*, as supplied by the user. More sophisticated strategies for inference of new lifting constructors is left as future work.

$StrongestInvCover()$ for affine relations involve identifying the basis vectors of the kernel of the matrix formed by the counterexamples in the bitvector domain [36, 17]. For inequality relations, $StrongestInvCover(\Gamma)$ returns *true* (i.e. the weakest invariant) iff any counterexample in Γ evaluates the relation to false — this effectively simulates the Houdini approach [24]. Similarly, in case of a recursive relation $l_1 \sim l_2$, $StrongestInvCover(\Gamma)$ returns *true* iff any counterexample in Γ evalutes its η-depth over-approximation $l_1 \sim_\eta l_2$ to false (effectively falsifying a weaker condition), where η is a fixed parameter of the algorithm.

4.4 More on Proof Discharge Algorithm

TODO: write 1-2 lines about what ”more” information this section contains

4.4.1 Handling Proof Obligations On Pathsets

Recall that our correlation algorithm attempts to correlate pathsets (instead of paths) between $\mathcal{S}$ and $\mathcal{C}$. Evidently, each edge of a (possibly partial) product-CFG π is associated with a pair of pathsets (ξ_S, ξ_C). Proof obligations originating across a product-CFG edge $e[s \to d] = (\xi_S, \xi_C)$ are of the form $\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\}$. A Hoare triple of the above form can be broken down into a conjunction of Hoare triples involving purely paths as follows:

$$\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\} \Leftrightarrow \bigwedge_{\substack{\rho_S \in \xi_S \\ \rho_C \in \xi_C}} \{\phi_s\}(\rho_S, \rho_C)\{\phi_d\} \tag{4.1}$$

Recall that our proof discharge algorithm requires that proof obligations satisfy the conjunctive recursive relation property (defined in section 3.1). If the original proof obligation $\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\}$ satisfies this property, so does each of the smaller proof obligation $\{\phi_s\}(\rho_S, \rho_C)\{\phi_d\}$. Hence, the proof discharge algorithm presented in chapter 3 is capable of handling these smaller proof obligations and by eq. (4.1), also the original proof obligations. If $|\xi|$ represents the number of paths in a pathset ξ, the proof obligation $\{\phi_s\}(\xi_S, \xi_C)\{\phi_d\}$ results in $|\xi_S| \times |\xi_C|$ smaller proof obligations. In practice, the number of paths in $\mathcal{S}$ that are correlated with a path in $\mathcal{C}$ is quite low and consequently, most of these proof obligations usually end up with a `false` `LHS` when lowered to first-order logic (through eq. (2.3)), due to the presence of $\texttt{pathcond}_{\rho_S}$ and $\texttt{pathcond}_{\rho_C}$. Our proof discharge procedure begins with an attempt to disprove `LHS` (overapproximated in case of recursive relations), which trivially resolves the smaller proof obligations involving uncorrelated paths to `true`.

4.4.2 Canonicalization Procedure

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

Algorithm 3: Pseudo-code for Canonicalization Procedure

Algorithm 3 shows the pseudo-code for the canonicalization procedure. *Canonicalize*(e) is responsible for converting an expression e to its canonical form $\hat{e}$ (introduced in section 3.2). Recall that a pseudo-variable is an expression of the form $v.\mathtt{a_1}.\mathtt{a_2}...\mathtt{a_n}$, where v is a variable. Also recall that, an expression e is canonical iff each *accessor* and *sum-is* expression operate on a pseudo-variable. An ADT expression with a data constructor, a lifting constructor or the <u>`if`</u>-<u>`then`</u>-<u>`else`</u>-operator at its top-level, is called a *foldable* expression. *Canonicalize*(e) iteratively folds each *accessor* and *sum-is* subexpressions of e that operate on a foldable argument. Thus, *Canonicalize*(e) returns an expression where none of the *accessor* or *sum-is* subexpressions is foldable. This condition entails the requirements of the canonical form. For example, $a + \mathtt{LCons}(b, l).\mathtt{tail}.\mathtt{val}$ and

$\mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(p)$ is `LNil` canonicalizes to $a + l.\mathtt{val}$ and $(p = 0)$ respectively.

4.4.3 Unification Procedure

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

Algorithm 4: Pseudo-code for Unification Procedure

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

is an atomic expression. In case both e_1 and e_2 contains a data constructor at their top-level, θ attempts to recursively unify the data constructors and their corresponding children. If exactly one of e_1 and e_2 is a `if`-`then`-`else` expression, θ attempts to unify both branches of `if`-`then`-`else` (along with the path conditions) with the other expression and return whichever succeeds. If both e_1 and e_2 are `if`-`then`-`else` expressions, θ attempts to recursively unify their children. θ uses the $\sqcup$ -operator to combine the results of successive self-calls. $A \sqcup B$ is equal to $\texttt{Succ}(S_1 \cup S_2)$ if $A = \texttt{Succ}(S_1)$ and $B = \texttt{Succ}(S_2)$; otherwise (if one of A and B is `Fail`), $A \sqcup B = \texttt{Fail}$. Additionally, for a `if`-`then`-`else` expression with `if` condition c, c is well-formed under the expression path condition. Hence, when conjuncting c to the expression path condition, we use an 'ordered and' operator $\underline{\wedge}$; $e_1 \underline{\wedge} e_2$ is equivalent to $e_1 \wedge (e_1 \rightarrow e_2)$.

4.4.4 Iterative Unification and Rewriting Procedure

Function $\Theta(p_a, e_a, p_b, e_b)$
    $R \hookleftarrow \emptyset$;
    $S \hookleftarrow \theta(p_a, e_a, p_b, e_b)$;
    if $S = \texttt{Fail}$ **then return** `Fail`;
    foreach $\langle p_1, a_1, p_2, e_2 \rangle$ **in** S **do**
        if e_2 **is** *atomic* **then**
            $R \hookleftarrow R \cup \{\langle p_1, a_1, p_2, e_2 \rangle\}$;
        else
            $e_1 \hookleftarrow rewrite(a_1)$;
            $R_1 \hookleftarrow \Theta(p_1, e_1, p_2, e_2)$;
            if $R_1 = \texttt{Fail}$ **then return** `Fail`;
            $R \hookleftarrow R \cup R_1$;
        end
    end
    return $\texttt{Succ}(R)$;
end

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

Algorithm 5 shows the pseudo-code for the iterative unification and rewriting procedure introduced in section 3.2. $\Theta(p_a, e_a, p_b, e_b)$ is responsible for unifying expressions e_a and e_b under the expression path conditions p_a and p_b respectively. Θ either fails to unify with the `Fail` output, or it successfully returns $\texttt{Succ}(S)$, where S is the set of correlation tuples that relate *only* atomic expressions. Θ attempts to iteratively (a) unify the expressions (through a call to the unification procedure θ in section 4.4), and (b) perform rewriting (of atom a_1 for those correlation tuples $\langle p_1, a_1, p_2, e_2 \rangle$ where e_2 is non-atomic), followed by a recursive call to Θ. For example, the unification of l_1 and $\texttt{LCons}(42, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$ yields the correlation tuples: $\langle true, l_1 \text{ is } \texttt{LCons}, true, true \rangle$, $\langle l_1 \text{ is } \texttt{LCons}, l_1.\texttt{val}, true, 42 \rangle$ and $\langle l_1 \text{ is } \texttt{LCons}, l_1.\texttt{tail}, true, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2) \rangle$.

4.4.5 Decomposition Procedure for Recursive Relations

Function $Decompose(l_1, l_2)$
  $ret \hookleftarrow true$;
  $\hat{l_1} \hookleftarrow Canonicalize(l_1)$;
  $\hat{l_2} \hookleftarrow Canonicalize(l_2)$;
  $R \hookleftarrow \Theta(true, \hat{l_1}, true, \hat{l_2})$;
  if $R = \texttt{Fail}$ **then return** $false$;
  foreach $\langle p_1, a_1, p_2, a_2 \rangle$ **in** R **do**
    if a_1 **is** $scalar$ **then**
      $ret \hookleftarrow ret \wedge ((p_1 \wedge p_2) \rightarrow (a_1 = a_2))$;
    else
      $ret \hookleftarrow ret \wedge ((p_1 \wedge p_2) \rightarrow (a_1 \sim a_2))$;
    end
  end
  return ret;
end

Algorithm 6: Pseudo-code of Decomposition Procedure for Recursive Relations

Algorithm 6 shows the pseudo-code for the decomposition algorithm defined in section 3.2. $Decompose(l_1, l_2)$ is responsible for computing the decomposition of the recursive relation $l_1 \sim l_2$. Recall that, decomposition of a recursive relation $l_1 \sim l_2$ requires the unification of (canonicalized) l_1 and l_2 through the top-level invocation of $\Theta(true, l_2, true, l_2)$. If the n correlation tuples obtained after a successful unification are $\langle p_1^i, a_1^i, p_2^i, a_2^i \rangle$ (for $i = 1 \ldots n$), then the decomposition of $l_1 \sim l_2$ is defined by eq. (3.2). If the unification fails (with a $\texttt{Fail}$ output), the decomposition is defined to be $\texttt{false}$. For example, recall that the unification of l_1 and $\texttt{LCons}(42, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$ yields the correlation tuples: $\langle true, l_1 \text{ is } \texttt{LCons}, true, true \rangle$, $\langle l_1 \text{ is } \texttt{LCons}, l_1.\texttt{val}, true, 42 \rangle$ and $\langle l_1$ is $\texttt{LCons},$ Consequently, $l_1 \sim \texttt{LCons}(42, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$ decomposes into the conjunctive predicate: $(l_1 \text{ is } \texttt{LCons}) \wedge (l_1 \text{ is } \texttt{LCons} \rightarrow l_1.\texttt{val} = 42) \wedge (l_1 \text{ is } \texttt{LCons} \rightarrow l_1.\texttt{next} \sim \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(l_2))$.

4.4.6 Reduction Procedures for Approximate Recursive Relations

Recall that type II proof obligations (summarized in section 3.5.4) are discharged by over- and under-approximating the `LHS` (resulting in a weaker and a stronger proof obligation respectively), followed by discharging both proof obligations through SMT solvers. We overapproximate `LHS` by substituting each recursive relation $l_1 \sim l_2$ in the `LHS` with its d_o-depth overapproximation $l_1 \sim_{d_o} l_2$. Similarly, the `LHS` is underapproximated by substituting each recursive relation $l_1 \sim l_2$ with its d_u-depth underapproximation $l_1 \sim_{d_u} l_2$.

D-depth Iterative Unification and Rewriting Procedure

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

Algorithm 7: Pseudo-code for D-Depth Unification Procedure

```
Function Θ_D(p_a, e_a, p_b, e_b, d)
    R ↤ ∅;
    S ↤ Θ_D(p_a, e_a, p_b, e_b, d);
    if S = Fail then return Fail;
    foreach ⟨p_1, a_1, p_2, e_2⟩_d' in S do
        if e_2 is atomic then
            if d' ≤ D and a_1 is not scalar then
                e_1 ↤ rewrite(a_1);
                e_2^r ↤ rewrite(e_2);
                R_1 ↤ Θ_D(p_1, e_1, p_2, e_2^r, d') if R_1 = Fail then return Fail;
                R ↤ R ∪ R_1;
            else
                R ↤ R ∪ {⟨p_1, a_1, p_2, e_2⟩_d'};
            end
        else
            e_1 ↤ rewrite(a_1);
            R_1 ↤ Θ_D(p_1, e_1, p_2, e_2, d');
            if R_1 = Fail then return Fail;
            R ↤ R ∪ R_1;
        end
    end
    return Succ(R);
end
```

Algorithm 8: Pseudo-code for D-Depth Iterative Unification and Rewriting Procedure

Recall that, section 3.5.3 briefly describes the process of reducing an overapproximate recursive relation into its SMT-encodable equivalent absent of recursive relations. We use modified versions of unification and 'iterative unification and rewriting' procedures (defined in sections 4.4.3 and 4.4.4 respectively) to reduce an overapproximate recursive relation into its SMT-equivalent. The D-depth unification and 'iterative unification and rewriting' procedures are represented by $\Theta_D(p_a, e_a, p_b, e_b, d)$ and $\theta_D(p_1, e_1, p_2, e_2, d)$ respectively, where D is a parameter of the algorithm. The pseudo-code for these two procedures are shown in algorithms 7 and 8 respectively. The D-depth 'iterative unification and rewriting' returns depth-augmented correlation tuples of the form $\langle p_1, a_1, p_2, a_2\rangle_d$ such that $d \geq D$ for all correlation tuples relating ADT values. Unlike Θ which terminates unification iff both expressions are atomic, Θ_D performs rewriting of both ADT atomic expressions and continues to unify deeper into their respective expression trees until all correlation tuples relate expressions at depth $\geq D$. For example, the 1-depth unification of l and $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$ yields the (depth augmented) correlation tuples: $\langle true, l \text{ is } \texttt{LNil}, true, p = 0\rangle_0$, $\langle l \text{ is } \texttt{LCons}, l.\texttt{val}, p \neq 0, p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{val}\rangle_1$ and $\langle l \text{ is } \texttt{LCons}, l.\texttt{tail}, p \neq 0, \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{next})\rangle_1$.

Reduction Procedure for Overapproximate Recursive Relations

Function $Overapprox_D(l_1, l_2)$
  $ret \hookleftarrow true$;
  $\hat{l_1} \hookleftarrow Canonicalize(l_1)$;
  $\hat{l_2} \hookleftarrow Canonicalize(l_2)$;
  $R \hookleftarrow \Theta_D(true, \hat{l_1}, true, \hat{l_2}, 0)$;
  if $R =$ `Fail` **then return** $false$;
  foreach $\langle p_1, a_1, p_2, a_2 \rangle_d$ **in** R **do**
    if a_1 **is** $scalar$ **and** $d \leq D$ **then**
      $ret \hookleftarrow ret \wedge (p_1 \wedge p_2) \rightarrow (a_1 = a_2)$;
    end
  end
  return ret;
end

Algorithm 9: Pseudo-code of Reduction Procedure for Overapproximated Recursive Relations

$Overapprox_D(l_1, l_2)$ is responsible for reducing the overapproximation $l_1 \sim_D l_2$ into its SMT-encodable equivalent condition. $Overapprox_D$ is similar to the decomposition procedure in section 4.4.5 except it only preserves scalar equalities till a maximum depth of D (inclusive). This essentially asserts that both l_1 and l_2 have identical structures (by equating expression path conditions) and equal scalar values (by equating scalar leaf expressions) till a depth of D. For example, recall that l and $\texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(p)$ 1-depth unifies into the correlation tuples: $\langle true, l \text{ is } \texttt{LNil}, true, p = 0 \rangle_0$, $\langle l \text{ is } \texttt{LCons}, l.\texttt{val}, p \neq 0, p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{val} \rangle_1$ and $\langle l \text{ is } \texttt{LCons}, l.\texttt{tail}, p \neq 0, \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{next}) \rangle_1$. Keeping only the scalar clauses till depth 1, $l \sim_1 \texttt{Clist}_{\mathbb{m}}^{\texttt{lnode}}(p)$ reduces to: $((l \text{ is } \texttt{LNil}) = (p = 0)) \wedge ((l \text{ is } \texttt{LCons}) \wedge (p \neq 0) \rightarrow l.\texttt{val} = p \xrightarrow{\mathbb{m}}_{\texttt{lnode}} \texttt{val})$.

Reduction Procedure for Underapproximate Recursive Relations

Function $isDepthBounded_D(l, p, d)$
  if $d > D$ **then return** $\neg p$;
  if l **is** *atomic* **then**
    if l **is** *scalar* **then return** *true*;
    else
      $l_r \hookleftarrow rewrite(l)$;
      return $isDepthBounded_D(l_r, p, d)$;
    end
  else if $l = V^{(n)}(l^1, l^2, \ldots, l^n)$ **then**
    return $\bigwedge_{i\in[1,n]} isDepthBounded_D(l^i, p, d+1)$
  else $l = \underline{\texttt{if}}\ c\ \underline{\texttt{then}}\ l^{\texttt{th}}\ \underline{\texttt{else}}\ l^{\texttt{el}}$
    $cond^{\texttt{th}} \hookleftarrow isDepthBounded_D(l^{\texttt{th}}, p \underline{\wedge} c, d)$;
    $cond^{\texttt{el}} \hookleftarrow isDepthBounded_D(l^{\texttt{el}}, p \underline{\wedge} \neg c, d)$;
    return $cond^{\texttt{th}} \wedge cond^{\texttt{el}}$
  end
end

Algorithm 10: Pseudo-code for *isDepthBounded* Procedure

Recall that, an underapproximate recursive relation $l_1 \approx_{d_u} l_2$ is equivalent to $\Gamma_{d_u}(l_1) \wedge \Gamma_{d_u}(l_2) \wedge l_1 \sim_{d_u} l_2$, where $\Gamma_d(l)$ asserts that l has a depth of at most d (defined in section 3.5.2). The function responsible for computing $\Gamma_D(l)$ is $isDepthBounded_D(l, p, d)$, where p and d represents the current expression path condition and depth of l, and D is a parameter of the algorithm. Algorithm 10 gives the pseudo-code for $isDepthBounded_D$. The top-level invocation is given by $isDepthBounded_D(l, true, 0)$. $isDepthBounded_D$ recursively traverses the expression tree of l (while rewriting as necessary), until it reaches a node at depth $> D$, at which point it returns the condition asserting the unreachability of such a node. For example, for a `List` variable l, $\Gamma_2(l)$ (i.e. $isDepthBounded_2(l, true, 0)$) returns the predicate: $\neg(l$ is $\texttt{LCons} \underline{\wedge} l.\texttt{tail}$ is $(LCons))$. Expanding $e_1 \underline{\wedge} e_2$ to $e_1 \wedge (e_1 \rightarrow e_2)$ and using the identity $\neg(e$ is $\texttt{LNil}) = e$ is `LCons`, the above reduces to the equivalent condition: $(l$ is $\texttt{LNil}) \vee ((l$ is $\texttt{LCons}) \wedge (l.\texttt{tail}$ is $\texttt{LNil}))$.

Function $Underapprox_D(l_1, l_2)$
  $\hat{l_1} \hookleftarrow Canonicalize(l_1)$;
  $\hat{l_2} \hookleftarrow Canonicalize(l_2)$;
  $overapprox \hookleftarrow Overapprox_D(\hat{l_1}, \hat{l_2})$;
  $depthbound_1 \hookleftarrow isDepthBounded_D(\hat{l_1}, true, 0)$;
  $depthbound_2 \hookleftarrow isDepthBounded_D(\hat{l_2}, true, 0)$;
  return $overapprox \wedge depthbound_1 \wedge depthbound_2$;
end

Algorithm 11: Pseudo-code of Reduction Procedure for Underapproximated Recursive Relations

Finally, $Underapprox_D(l_1, l_2)$ is responsible for reducing the underapproximation $l_1 \approx_D l_2$ into its SMT-encodable equivalent condition. The pseudo-code for $Underapprox_D$ is given in algorithm 11. For example, $l \approx_1 \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p) \Leftrightarrow \Gamma_1(l) \wedge \Gamma_1(\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p)) \wedge l \sim_1 \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p)$. $\Gamma_1(l)$ and $\Gamma_1(\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p))$ reduces to the conditions l is $\mathtt{LNil}$ and $(p = 0)$ resectively. Finally, $l \approx_1 \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(p)$ reduces to the condition: l is $\mathtt{LNil} \wedge (p = 0) \wedge ((l \text{ is } \mathtt{LNil}) = (p = 0)) \wedge ((l \text{ is } \mathtt{LCons}) \wedge (p \neq 0) \rightarrow l.\mathtt{val} = p \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{val})$.

4.4.7 SMT Encoding of First Order Logic Formula

As summarized in algorithm 1, our proof discharge algorithm solves a proof obligation $P : \mathtt{LHS} \Rightarrow \mathtt{RHS}$, through a sequence of queries $P_i : \mathtt{LHS}_i \Rightarrow \mathtt{RHS}_i$ to off-the-shelf SMT solvers. Recall that P may contain recursive relations. However, our algorithm ensures that each P_i is free of recursive relations and only contain scalar equalities. We encode each query P_i in SMT logic with bitvector and array theories. In this section, we describe the process of encoding a proof obligation P_i into SMT logic. We begin by converting $P_i : \mathtt{LHS}_i \Rightarrow \mathtt{RHS}_i$ into its canonical form $\hat{P_i}$ (as described in section 4.4.2). Although $\hat{P_i}$ does not contain recursive relations, it may still contain ADT variables alongside *accessor* and *sum-is* expressions. Due to canonicalization, all top-level *accessor* and *sum-is* expressions must be of the form $v.\mathtt{a_1}.\mathtt{a_2}...\mathtt{a_n}$ and $v.\mathtt{a_1}.\mathtt{a_2}...\mathtt{a_n}$ is V respectively. We call such an expression e *flattenable* and the ADT variable v is called the *index* of e. $\hat{P_i}$ is lowered into an intermediate expression P_i^f through a process called *flattening*. This involves 'flattening' all flattenable expressions to variables such that P_i^f only contains scalar values with scalar and memory operations (but importantly not ADT values). The flattening process is described below.

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

2. For each ADT T with data constructors $V_1, V_2, \ldots, V_k$, we define an enumeration type $\mathcal{E}(T)$ in SMT logic with items $\mathcal{E}(V_1), \mathcal{E}(V_2), \ldots, \mathcal{E}(V_k)$ respectively. In the canonical form, each *sum-is* expression e must operate on a pseudo-variable. The last step guarantees that e must be of the form: $e = v$ is V. We replace e with the its SMT equivalent: $(v \parallel tag) = \mathcal{E}(V)$ [3].

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

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

[3] Spec does not allow naming a field of a data constructor `tag`. Furthermore, fields cannot contain the '_' character. Combined, these two conditions prevent collision between variable names obtained due to flattening.

4.4.8 Reconciliation of Counterexamples

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

Algorithm 12: Pseudo-code for Reconciliation Procedure

As detailed in section 4.4.7, each ADT variable v gets lowered into a set of scalar variables $\mathcal{F}(v)$ during SMT encoding. Evidently, the models returned by SMT solvers map these variables (in $\mathcal{F}(v)$ instead of v) to constant values. We are interested in recovering a counterexample for the original query from a model returned by the SMT solver. Recall that, these counterexamples help guide the correlation search (in section 4.2) and invariant inference (in section 4.3) procedures. The process of constructing a constant for v from the constant values returned for $\mathcal{F}(v)$ by an SMT solver is called *reconciliation*. Obviously, the reconciled counterexample must be a valid counterexample to the original proof obligation. $\mathit{Reconcile}(v : T, \gamma)$ is responsible for performing reconciliation for variable v (of type T) from the model γ (returned by a SMT solver). $\mathit{Rand}(T)$ returns an arbitrary constant of type T. For example, consider the rather contrived proof obligation P : `true` $\Rightarrow l$ is `LNil`. Clearly, any valuation of l where l is a non-empty list is a valid counterexample to P. However, a counterexample γ returned by an SMT solver would instead contain the mapping $\{l\_tag \mapsto \mathcal{E}(\texttt{LCons})\}$. During reconciliation, we find that l_tag is mapped to the data constructor `LCons` and recurse for each of its fields `val` and `tail`. Since γ do not contain a mapping for either of these fields, we soundly generate random constants for these instead. Indeed, *Reconcile* correctly constructs a non-empty but otherwise arbitrary list for l, which is a counterexample to P.

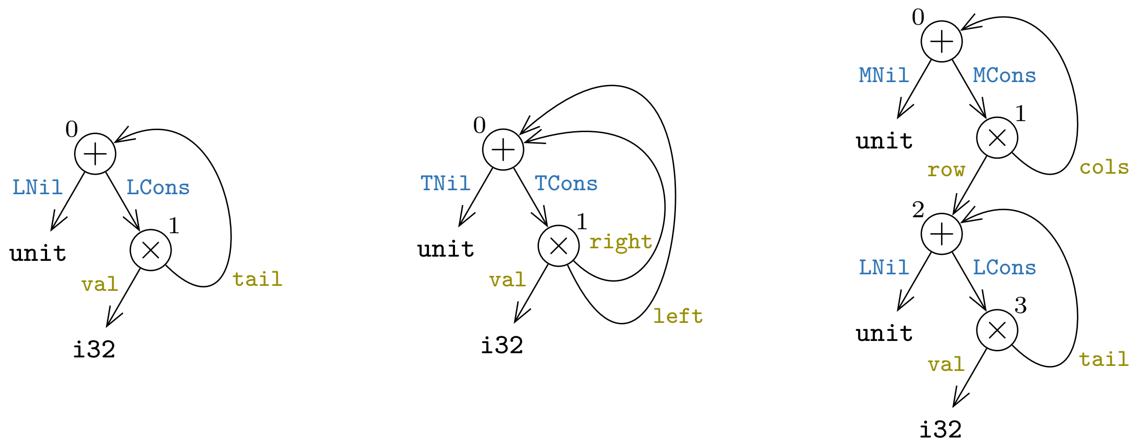
4.4.9 Value Tree Representation

This section presents a graphical representation of expressions that helps us simplify the implementation of multiple subprocedures used by our proof discharge algorithm. This includes the process of canonicalization, reducing approximate recursive relations as well as construction of deconstruction programs as part of type III proof obligations. We call this the *Value Tree* representation and use $\mathcal{V}(e)$ to denote a value tree associated with e. We give an algorithm to convert an expression e into $\mathcal{V}(e)$ and list its applications.

Before diving into value trees, we start by introducing an analogous (but simpler) representation for types, called *Type Trees*. We use $\mathcal{T}(\tau)$ to denote a type tree associated with τ. Recall that ADTs are simply 'sum of product' types where each data construction represents a variant (of the sum-type) and each data construction contains values for each of its fields (of the product-type). On top of ADTs, IR has build-in scalar types: `unit`, `bool` and `i<N>`. Types in IR can be represented in *first order recursive types* [25] using the product ($\times$) and sum ($+$) type constructors; and the scalar types (i.e. nullary type constructors). The type system is characterized by the grammar $\mathbb{T}$ as follows:

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

Every IR type can be encoded as a closed term (i.e. term without free variables) in $\mathbb{T}$. For example, the `List` type can be written as $\mu\alpha.\texttt{unit} + (\texttt{i32} \times \alpha)$. Note the use of a type variable α which is bound using μ to represent recursion. Similarly, the `Matrix` type is represented by the term $\mu\alpha.\texttt{unit} + ((\mu\beta.\texttt{unit} + (\texttt{i32} \times \beta)) \times \alpha)$, where the type variables α and β are used to bind recursive types `Matrix` and `List` at their definitions respectively.



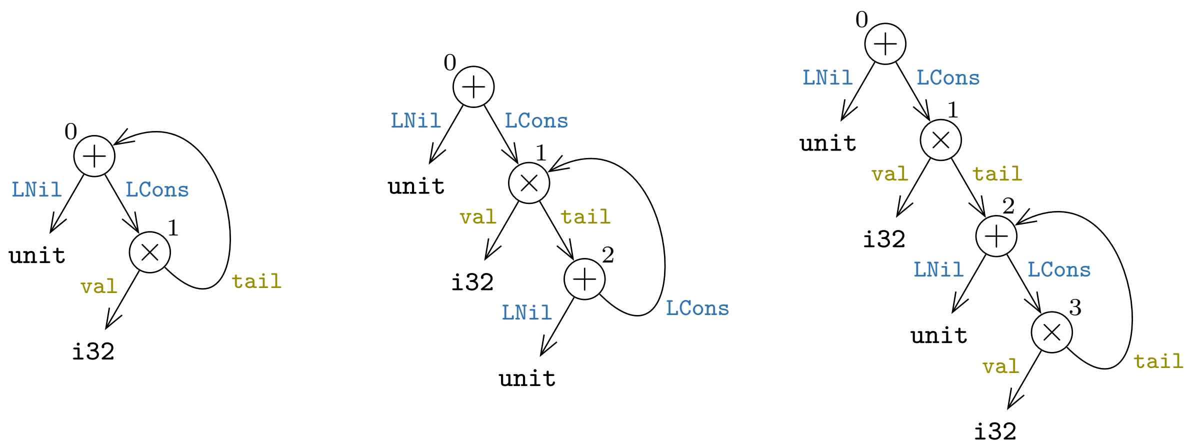
(a) `List` = `LNil` | `LCons`(`i32`, `List`)

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

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

Figure 4.2: Type trees for the ADTs `List`, `Tree` and `Matrix` respectively.

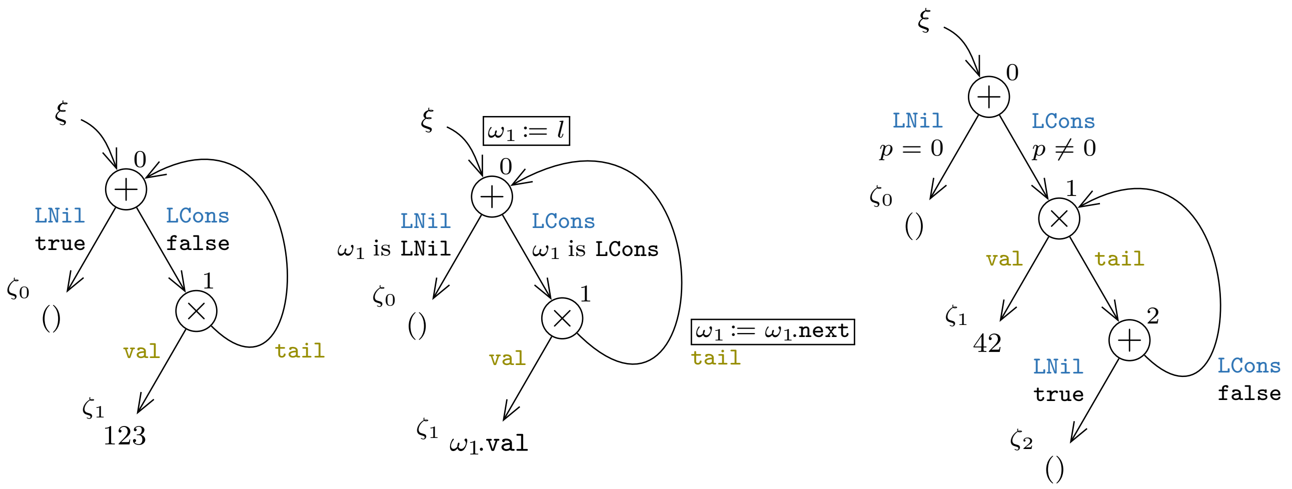
Figure 4.2 shows the type trees for three ADTs `List`, `Tree`, and `Matrix` respectively. In a type tree, each internal node represents either a product ($\otimes$) or a sum ($\oplus$) type constructor. The leaf nodes are the scalar types. Each outgoing edge of a $\oplus$ node is associated with a data constructor of the corresponding ADT (i.e. `LCons` for `List`). Similarly, each outgoing edge of a $\otimes$ node is associated with a field of the corresponding data constructor (i.e. `val` for `LCons`). We assign integer indices to the internal nodes and use $[v \rightarrow \texttt{label}]$ to identify the edge outgoing at v associated with `label`, where `label` is either a data constructor or a field name. The root node is denoted by v_0. The edges going outward from the root node are called *tree-edges* e.g., $[0 \rightarrow \texttt{LCons}]$ and $[1 \rightarrow \texttt{val}]$ in fig. 4.2a. Edges that are not tree-edges, are called *back-edges* e.g., $[1 \rightarrow \texttt{cols}]$ in fig. 4.2c. Every back-edge induces an unique simple cycle in the type tree representation.



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

Figure 4.3: Three type trees for `List` ADT. Figure 4.3a shows the type tree for the canonical form of `List`. Figure 4.3b is obtained by peeling the back-edge [1 → `tail`] in fig. 4.3a. Figure 4.3c is obtained by unrolling the back-edge [1 → `tail`] in fig. 4.3a or by peeling the back-edge [2 → `LCons`] in fig. 4.3b respectively.

Recall that types in Spec (and in IR) follow equirecursive typing rules i.e. types $\mu\alpha.T$ and $T[\mu\alpha.T/\alpha]$ in $\mathbb{T}$ are *equal* types, where $T[\mu\alpha.T/\alpha]$ represents the new type obtained by substituting all free instances of α with $\mu\alpha.T$, and is defined as the *unfolding* of $\mu\alpha.T$. In general, under equirecursive typing, two types are equal iff their infinite expansions (through unfolding) are equal. In the type tree representation, two types are equal iff their infinite expansions are equivalent. Such type trees are called isomorphic and two types are isomorphic iff they represent the same type. An unfolding in the term representation corresponds to *unrolling* one iteration of a simple cycle in its type tree. Figure 4.3 shows three type trees for the `List` type. Figure 4.3a corresponds to the canonical (intuitively the 'smallest') type tree for the `List` type. The type trees figs. 4.3b and 4.3c are obtained by *peeling* and unrolling the back-edge [1 → `tail`] (in fig. 4.3a) respectively. Peeling is a form of partial unrolling which only extracts the starting node of the cycle. In practice, equality of two types (encoded in $\mathbb{T}$) can be reposed as syntactic equality of their *canonical forms* [18]. In general, type trees may contain cycles (due to back-edges) and hence are not quite 'trees'. However, they represent the actual (possibly infinite) trees obtained through repeated unrolling of cycles.



(a) `LNil` **(b)** l:`List` **(c)** $\underline{\texttt{if}}\ p = 0\ \underline{\texttt{then}}$ `LNil` $\underline{\texttt{else}}$ `LCons`$(42,$ `LNil`$)$

Figure 4.4: Value trees of three `List`-typed expressions

With type trees out of the way, we are ready to present their value analogue called 'value trees'. Figure 4.4 shows the value trees for three `List` expressions. Note that, all three value trees are isomorphic to one of the `List` type trees shown in fig. 4.3, e.g., fig. 4.4c is isomorphic to fig. 4.3b. In general, for an expression e of type τ, its value tree $\mathcal{V}(e)$ resembles its type tree with the following distinctions:

1. Similar to a type tree, each internal node is either a $\oplus$ or a $\otimes$ node.

2. Instead of a scalar type τ_s, each leaf node in $\mathcal{V}(e)$ contains an expression of type τ_s.

3. Similar to the Control-Flow Graph representation (presented in section 2.2.3), each node v is associated with a symbolic state Ω_v.

4. In addition to a data constructor, each edge originating at a $\oplus$ node v also contains an edge condition expression (a boolean valued function over Ω_v). We identify such an edge

with $[v \xrightarrow{c} V]$, where v is the sum node, V is a data constructor and c is the edge condition. The set of edge conditions for all outgoing edges at a $\oplus$ node must be mutually exclusive and exhaustive.

5. In addition to a field name, each edge $v \rightarrow v'$ originating at a $\otimes$ node v also contains a transfer function ($\Omega_{v'}$ as a function of Ω_v). We identify such an edge with $[v \xrightarrow{\texttt{tf}} \texttt{fi}]$, where v is the product node, `fi` is a field name and `tf` is the transfer function.

6. Additionally, a value tree also contains a special node (called the *entry node*), and a special edge (called the *entry edge*) from the entry node to v_0 (i.e. the root of the tree). We use ξ to denote the entry node. The entry edge is associated with a transfer function $\texttt{tf}_\xi$. We often omit the entry node in figures for brevity.

7. A value tree $\mathcal{V}(e)$ can be converted to a type tree $\mathcal{T}$ as follows: (a) remove the entry node and edge pair, (b) remove edge conditions and transfer functions associated with all edges, and (c) replace each leaf node expression of (scalar) type τ_s with τ_s itself. The resulting type tree $\mathcal{T}$ represents the type τ of the expression e.

Intuitively, a value tree simultaneously represents the value of the expression as well as the CFG of its *abstracted* deconstruction program. We will subsequently discuss these properties along with their applications in the context of our proof discharge algorithm. Next, we give an algorithm to convert an expression e to its value tree representation $\mathcal{V}(e)$.

4.4.10 Conversion of Expressions to their Value Trees

In this section, we present an algorithm to recursively construct a value tree for any arbitrary expression e. We take a visual approach to match the graphical nature of value trees.

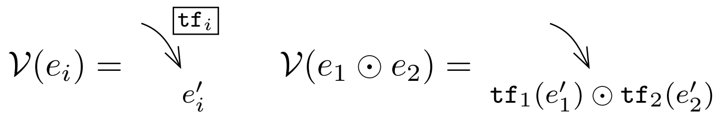


Figure 4.5: Construction of $\mathcal{V}(e_1 \odot e_2)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$. $\odot$ represents an arbitrary scalar operator.

Scalar Operators

Given an expression $e = e_1 \odot e_2$, fig. 4.5 shows the construction of $\mathcal{V}(e)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$ respectively. Since e_1 and e_2 have scalar types, their value trees must have exactly one node (i.e. the leaf node) containing an expression (e_1' and e_2' respectively) of the same type. $\odot$ represents an aribtrary scalar operator, i.e. an operator whose arguments are scalar-typed values (e.g., bitvector arithmatic and relational operators). Given an expression s and a transfer function `tf`, $\mathtt{tf}(s)$ represents the expression obtained by applying `tf`, interpreted as a substitution, to s. This is equivalent to the weakest-precondition of s along an edge associated with the `tf`. The construction shown in fig. 4.5 can be generalized to n-ary scalar operators for $n > 2$.

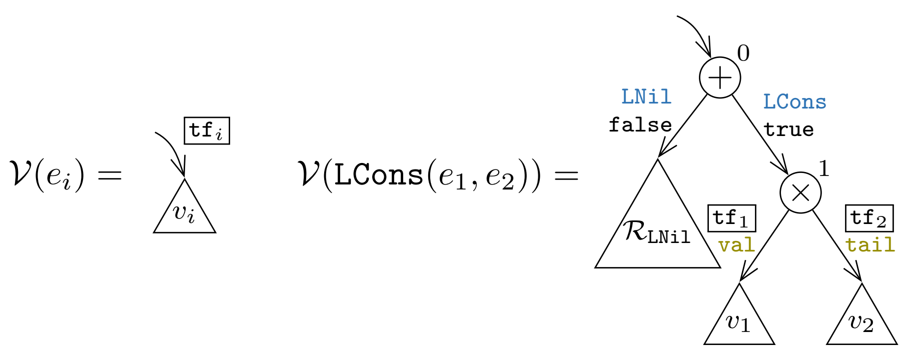


Figure 4.6: Construction of $\mathcal{V}(\mathtt{LCons}(e_1, e_2))$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$. $\mathcal{R}_{\mathtt{LNil}}$ represents an arbitrary value tree corresponding to the product-type (in $\mathbb{T}$) associated with `LNil`.

ADT Data Constructors

Given an expression $e = \texttt{LCons}(e_1, e_2)$, fig. 4.6 depicts the construction of $\mathcal{V}(e)$ from $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$ respectively. In general, for an arbitrary data constructor V of ADT T, the process begins with a $\oplus$ node (0 in fig. 4.6) such that the outgoing edge associated with the value constructor V (LCons in fig. 4.6) has an edge condition of `true` while all other edges are assigned the edge condition `false`. For each data constructor $V' \neq V$ of T, we append a random value tree corresponding to the product-type associated with V' in $\mathbb{T}$. For example, given `List` is associated with the sum-type $\mu\alpha.\texttt{Unit} + (\texttt{i32} \times \alpha)$, the product-types associated with `LNil` and `LCons` are: `Unit` and $\mu\alpha.\texttt{i32} \times (\texttt{Unit} + \alpha)$ respectively. We use $\mathcal{R}_\tau$ to denote an arbitrary (i.e. random) value tree of type τ. For the outgoing edge associated with the data constructor V ($[0 \xrightarrow{\texttt{true}} \texttt{LCons}]$ in fig. 4.6), we construct a product node (1 in fig. 4.6) and append the value trees corresponding to the arguments e_i as children of the product node.

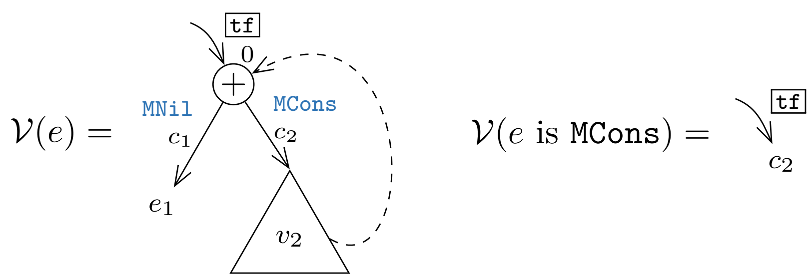


Figure 4.7: Construction of $\mathcal{V}(e \text{ is } \texttt{MCons})$ from $\mathcal{V}(e)$.
The dashed edge represents the (possibly empty) set of backedges originating in v_2 that terminates at 0.

Sum-Is Operator

Given a *sum-is* expression $e' = e$ is `MCons`, fig. 4.7 shows the construction of $\mathcal{V}(e')$ from $\mathcal{V}(e)$. The process is rather straightforward and for a general expression e is $V_{\texttt{i}}$, entails extracting the edge condition c (c_2 in fig. 4.7) from the $\mathcal{V}(e \text{ is } V_{\texttt{i}})$ edge $[v_0 \xrightarrow{c} \texttt{V}]$ ($[0 \xrightarrow{c_2} \texttt{MCons}]$ in fig. 4.7). Notice that the entry transfer function $\texttt{tf}_\xi$ during this conversion.

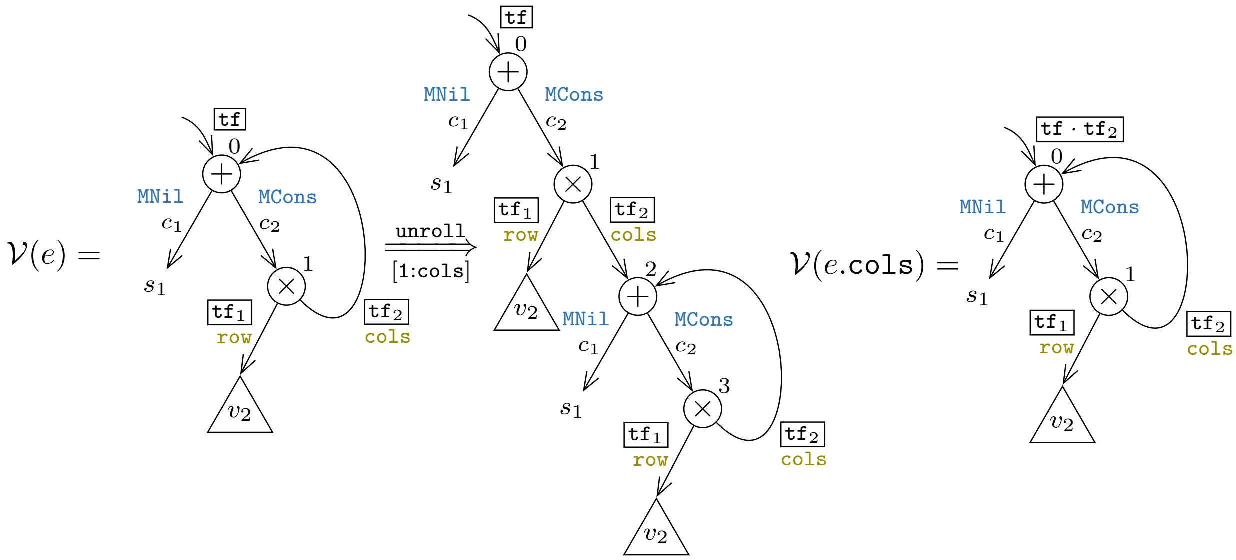


Figure 4.8: Construction of $\mathcal{V}(e.\texttt{cols})$ from $\mathcal{V}(e)$.
Similar to type trees, `unroll` $[1 \rightarrow \texttt{cols}]$ represents the operation of hoisting one iteration of the cycle 0→1→0.

Product-Access Operator

Given an expression $e' = e.\texttt{cols}$, fig. 4.8 depicts the construction of $\mathcal{V}(e')$ from $\mathcal{V}(e)$. Intuitively, $\mathcal{V}(e')$ represents the subtree of $\mathcal{V}(e)$ rooted at the $\oplus$ node reached by taking the edges $[0 \xrightarrow{c_2} \texttt{MCons}]$ followed by $[1 \xrightarrow{\Omega_2} \texttt{cols}]$. However, this path may contain backedges or the subtree itself may contain backedges leaving the subtree. In such a case, we perform peeling until all such backedges strickly terminate within this subtree. For example, in fig. 4.8, the edge $[1 \xrightarrow{\Omega_2} \texttt{cols}]$ is a backedge and hence we peel it once. In the resulting (equivalent) value tree, the subtree (rooted at 2) contains a backedge leaving the subtree which requires one more peeling operation. The resulting value tree contains the subtree rooted at the $\oplus$ node 2 which satisfies the two conditions above and hence $\mathcal{V}(e')$ is simply constructed by extracting the subtree rooted at $\oplus$ node 2. Note that we preserve the transfer functions from the entry to the $\oplus$ node 2 during extraction. $\texttt{tf}_1 \cdot \texttt{tf}_2$ represents the composition of the transfer functions $\texttt{tf}_1$ and $\texttt{tf}_2$ respectively.

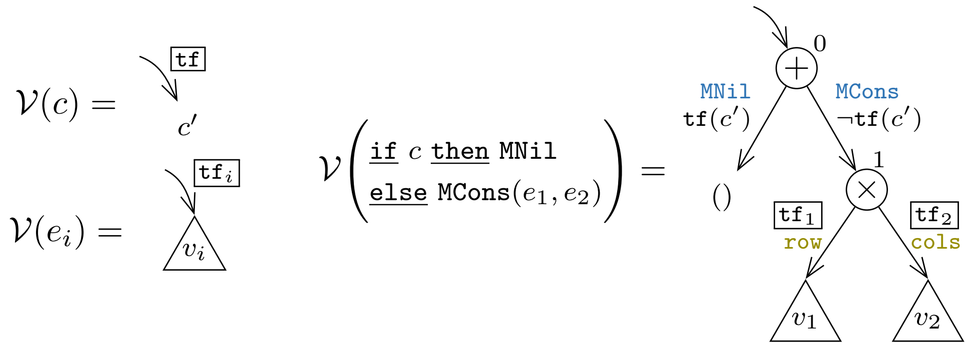


Figure 4.9: Construction of $\mathcal{V}($ $\underline{\texttt{if}}$ c $\underline{\texttt{then}}$ `MNil` $\underline{\texttt{else}}$ `MCons`(e_1, e_2) $)$ from $\mathcal{V}(c)$, $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$.

<u>If</u>-<u>Then</u>-<u>Else</u> Operator

Given an expression $e =$ $\underline{\texttt{if}}$ c $\underline{\texttt{then}}$ `MNil` $\underline{\texttt{else}}$ `MCons`(e_1, e_2), fig. 4.9 describes the construction of $\mathcal{V}(e)$ using $\mathcal{V}(c)$, $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$. Let us consider a general $\underline{\texttt{if}}$-$\underline{\texttt{then}}$-$\underline{\texttt{else}}$ expression e (associated with the ADT T with data constructors $V_1, V_2, \ldots, V_n$) such that the branch associated with V_i is given by $V_i(e_i^1, e_i^2, \ldots)$. We begin with the construction of a $\oplus$ root node (0 in fig. 4.9) such that the outgoing edge associated with V_i has the edge condition equal to the expression path condition of the branch $V_i(e_i^1, e_i^2, \ldots)$ ($\texttt{tf}(c')$ and $\neg\texttt{tf}(c')$ for `MNil` and `MCons` respectively in fig. 4.9). For each outgoing edge associated with the data constructor V_i, we construct a $\otimes$ node (1 for `MCons` in fig. 4.9) and append the value trees corresponding to the arguments e_i^j as its children.

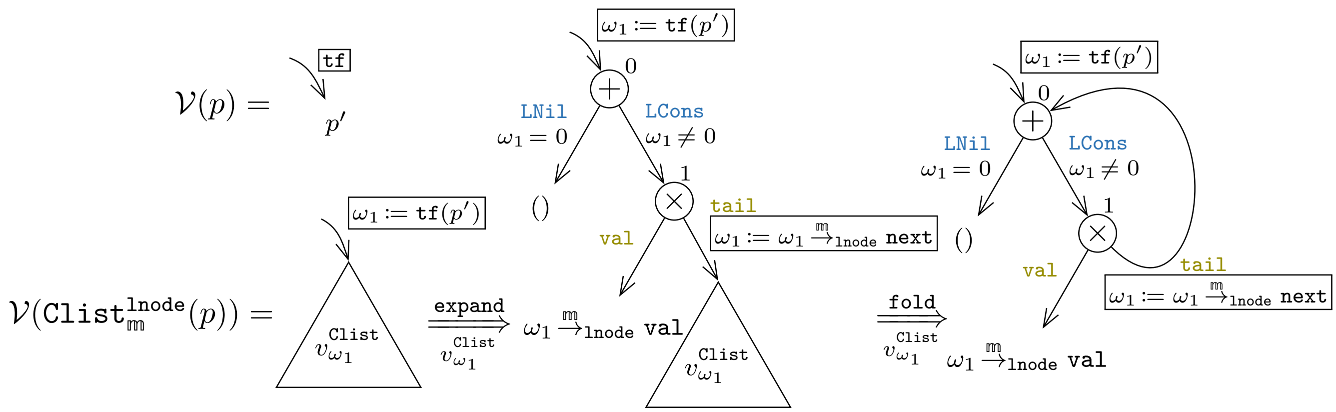


Figure 4.10: Construction of $\mathcal{V}(\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p))$ from $\mathcal{V}(p)$.
The process involves assuming $v^{\texttt{Clist}}_{\omega_1}$ to be the value tree of $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\omega_1)$, followed by expansion of $v^{\texttt{Clist}}_{\omega_1}$ using the definition of $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$ (in eq. (2.2)) and, finally folding the tree-edge $[1 \rightarrow \texttt{tail}]$ incident on the self-referencial subtree $v^{\texttt{Clist}}_{\omega_1}$ into a backedge.

Lifting Constructor

Given an expression $e = \texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$, fig. 4.10 shows the construction of $\mathcal{V}(e)$ from $\mathcal{V}(p)$. Recall the recursive definition of the lifting constructor $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$ given in eq. (2.2). We start by assuming that $v^{\texttt{Clist}}_{\omega_1}$ is the value tree for the lifted expression $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(\omega_1)$. Hence, the value tree of $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}(p)$ is identical to $v^{\texttt{Clist}}_{\omega_1}$ except we assign the actual argument (i.e. $\texttt{tf}(p')$) to the formal argument ω_1 along the entry edge. Next, we expand the subtree $v^{\texttt{Clist}}_{\omega_1}$ based on the unrolling procedure of $\texttt{Clist}^{\texttt{lnode}}_{\mathbb{m}}$ (defined in eq. (2.2)) until the entire value tree becomes a self-referencial structure. For example, after expanding through eq. (2.2) once in fig. 4.10, the value tree contains an tree-edge $[1 \rightarrow \texttt{tail}]$ incident on the self-referencial subtree $v^{\texttt{Clist}}_{\omega_1}$. In the last step, we fold all self-referencial tree-edges ($[1 \rightarrow \texttt{tail}]$ in fig. 4.10) by converting them into backedges terminating at the root of the subtree being referenced (0 for $v^{\texttt{Clist}}_{\omega_1}$ in fig. 4.10).

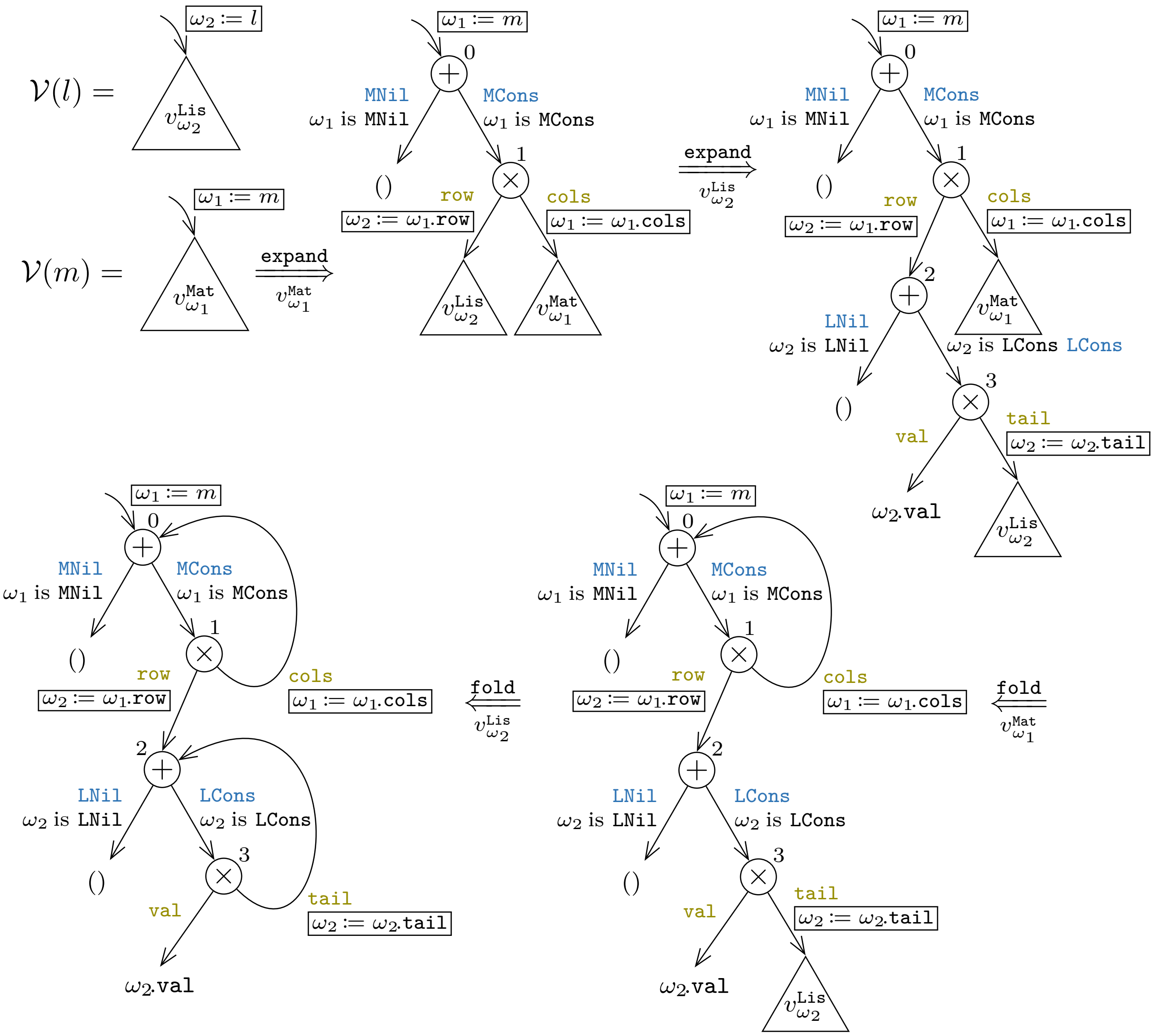


Figure 4.11: Construction of $\mathcal{V}(m)$ for a `Matrix` variable m. The process is identical to the construction of value trees for lifted expressions as shown in fig. 4.10.

Variables

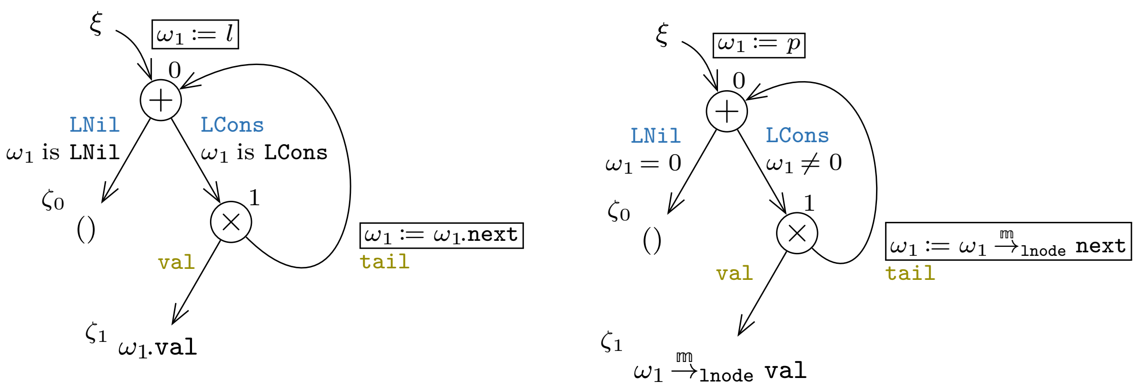
Finally, we are interested in constructing the value tree for a variable. Recall that, every ADT (pseudo-)variable is associated with an unrolling procedure characterized by the ADT itself. e.g. eq. (2.1) for the `List` variable l. The `Matrix` ADT is defined as: $\texttt{Matrix} = \texttt{MNil} \mid \texttt{MCons}(\texttt{List}, \texttt{Matrix})$, and thus the unrolling procedure for a `Matrix` variable m is given by:

$$m = \underline{\texttt{if}}\ m \text{ is } \texttt{MNil}\ \underline{\texttt{then}}\ \texttt{MNil}\ \underline{\texttt{else}}\ \texttt{MCons}(m.\texttt{row}, m.\texttt{cols}) \tag{4.2}$$

Figure 4.11 illustrates the construction $\mathcal{V}(m)$ for the `Matrix` variable m. The process consists of the same three steps used to construct the value tree for a lifted expression – assume, expand and fold. First, we assume that $v^{\texttt{Mat}}_{\omega_1}$ and $v^{\texttt{Lis}}_{\omega_2}$ are the value trees corresponding to the pseudo-variables ω_1 and ω_2 of `Matrix` and `List` types respectively. Thus, $\mathcal{V}(m)$ is equal to $v^{\texttt{Mat}}_{\omega_1}$ with the entry edge transfer function $\{\omega_1 \hookleftarrow m\}$. We expand the definitions of $v^{\texttt{Mat}}_{\omega_1}$ and $v^{\texttt{Lis}}_{\omega_2}$ once each before the value tree becomes self-referencial. Finally, we fold the treeedges $[1 \to \texttt{cols}]$ and $[3 \to \texttt{tail}]$ into the back-edges terminating at the roots of the subtrees representing $v^{\texttt{Mat}}_{\omega_1}$ and $v^{\texttt{Lis}}_{\omega_2}$ respectively (nodes 0 and 3 in fig. 4.11).

4.4.11 Applications of Value Trees

With the conversion algorithm out of the way, we next discuss a handful of properties of value trees along with their applications, in the context of our proof discharge algorithm. Recall that, $[v \to \texttt{label}]$ represents the edge outgoing at a $\oplus$ or $\otimes$ node v and associated with the label `label`. Generalizing, $[v \to \texttt{label}_1, \texttt{label}_2, \ldots \texttt{label}_\texttt{n}]$ represents the path starting at node v where its n edges are associated with the labels $\texttt{label}_\texttt{i} \forall i \in [1, n]$ in order. Also, for a field `fi`, let `Ctor(fi)` represent the data constructor of which `fi` is a field, e.g., $\texttt{Ctor}(\texttt{tail}) = \texttt{LCons}$.



(a) $\mathcal{V}(l)$ **(b)** $\mathcal{V}(\mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(p))$

Figure 4.12: Value trees for a `List` variable l and a lifted expression $\mathtt{Clist}_{\mathtt{m}}^{\mathtt{lnode}}(p)$ respectively.

Canonical Deconstruction Property

Let e be an ADT expression and $\mathcal{V}(e)$ be its value tree. For any *scalar* expression $e_s = e.\mathtt{a}_1.\mathtt{a}_2 ... \mathtt{a}_\mathtt{n}$, let $\mathtt{Path}(e_s)$ represent the path $\xi \rightarrow [v_0 \rightarrow \mathtt{Ctor}(\mathtt{a}_1), \mathtt{a}_1, \ldots, \mathtt{Ctor}(\mathtt{a}_\mathtt{n}), \mathtt{a}_\mathtt{n}]$ in the value tree $\mathcal{V}(e)$. Recall that ξ and v_0 represents the entry and root nodes respectively. Moreoever, since e_s is a scalar expression, the path $\mathtt{Path}(e_s)$ must also terminate at a leaf node (say v_s). Let, $\mathtt{pathcond}(\mathtt{Path}(e_s))$ and $\mathtt{value}(\mathtt{Path}(e_s))$ represent the path condition and the value associated with v_s along the path $\mathtt{Path}(e_s)$. The *canonical deconstruction property* for a value tree ensures that:

1. The condition under which e_s is accessible is equal to $\mathtt{pathcond}(\mathtt{Path}(e_s))$.
2. The value e_s is equivalent to $\mathtt{value}(\mathtt{Path}(e_s))$.
3. Both $\mathtt{pathcond}(\mathtt{Path}(e_s))$ and $\mathtt{value}(\mathtt{Path}(e_s))$ are in the canonical form (section 4.4.2).

Consider the value tree corresponding to a `List` variable l, as shown in fig. 4.12a. For the expression $e_s = l.\mathtt{tail}.\mathtt{val}$, $\mathtt{Path}(e_s)$ is given by $\xi \to [v_0 \to \mathtt{LCons}, \mathtt{tail}, \mathtt{LCons}, \mathtt{val}]$ i.e. $\xi{\to}\mathtt{0}{\to}\mathtt{1}{\to}\mathtt{0}{\to}\mathtt{1}{\to}\zeta_1$. The corresponding $\mathtt{pathcond}(\mathtt{Path}(e_s))$ and $\mathtt{value}(\mathtt{Path}(e_s))$ are given by (l is $\mathtt{LCons}$)$\triangle$($l.\mathtt{next}$ is $\mathtt{LCons}$) and $l.\mathtt{next}.\mathtt{val}$ respectively.

Reduction of Approximate Recursive Relations

Since an ADT represents a 'sum of product' type, each level of an ADT value (in its expression tree as shown in section 3.5.1) corresponds to two levels – $\oplus$ and $\otimes$ in the value tree representation. Combining the above observation with the canonical deconstruction property allows us to construct the value trees of over- and under-approximate recursive relations directly from the value trees of its argument pair. A d-depth over-approximation of $l_1 \sim l_2$ simply asserts equality of all leaf values (along with their path conditions) between $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ up to a depth of $2 \cdot d$. Similarly, the d-depth under-approximation of $l_1 \sim l_2$ asserts the above in conjunction with the unreachability of all paths outgoing at depth $2 \cdot d$ (i.e. paths incident at depth $2 \cdot d{+}1$). Let $\mathtt{Path}_D(e_1, e_2)$ be the set of all path pairs $\langle \xi \to [v_0 \to \mathtt{label}_1, \mathtt{label}_2, \ldots \mathtt{label}_\mathtt{n}], \xi \to [v_0 \to \mathtt{label}_1, \mathtt{label}_2, \ldots \mathtt{label}_\mathtt{n}] \rangle$ in $\mathcal{V}(e_1)$ and $\mathcal{V}(e_2)$ respectively that – (a) originate at their entry nodes and (b) terminate at leaf nodes at depth equal to D (i.e. spans $(D+1)$ edges). For example, consider the value trees shown in fig. 4.12 corresponding to the $\mathtt{List}$ values $e_1 = l$ and $e_2 = \mathtt{Clist}^{\mathtt{clode}}_{\mathtt{m}}(p)$ respectively. Then, $\mathtt{Path}_0(e_1, e_2) = \emptyset$, $\mathtt{Path}_1(e_1, e_2) = \{\langle \xi{\to}\mathtt{0}{\to}\zeta_\mathtt{0}, \xi{\to}\mathtt{0}{\to}\zeta_\mathtt{0} \rangle\}$, and $\mathtt{Path}_2(e_1, e_2) = \{\langle \xi{\to}\mathtt{0}{\to}\mathtt{1}{\to}\zeta_\mathtt{1}, \xi{\to}\mathtt{0}{\to}\mathtt{1}{\to}\zeta_\mathtt{1} \rangle\}$. Finally, the d-depth approximations of $l_1 \sim l_2$ are given by

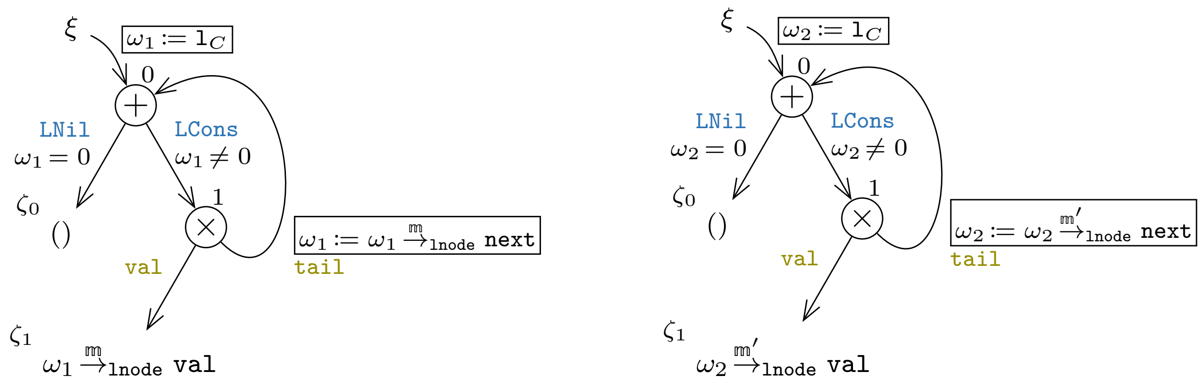
$$
\begin{aligned}
l_1 \sim_d l_2 &= \sum_{\delta=0}^{2\cdot d} \Bigg(\bigwedge_{\substack{\langle p_1, p_2 \rangle \in \\ \mathtt{Path}_\delta(l_1, l_2)}} \begin{matrix} \mathtt{pathcond}(p_1) = \mathtt{pathcond}(p_2) \\ \mathtt{value}(p_1) = \mathtt{value}(p_2) \end{matrix} \Bigg) \\
l_1 \approx_d l_2 &= \sum_{\delta=0}^{2\cdot d} \Bigg(\bigwedge_{\substack{\langle p_1, p_2 \rangle \in \\ \mathtt{Path}_\delta(l_1, l_2)}} \begin{matrix} \mathtt{pathcond}(p_1) = \mathtt{pathcond}(p_2) \\ \mathtt{value}(p_1) = \mathtt{value}(p_2) \end{matrix} \Bigg) \wedge \Bigg(\bigwedge_{\substack{\langle p_1, p_2 \rangle \in \\ \mathtt{Path}_{2\cdot d+1}(l_1, l_2)}} \begin{matrix} \neg\mathtt{pathcond}(p_1) \\ \neg\mathtt{pathcond}(p_2) \end{matrix} \Bigg)
\end{aligned}
\tag{4.3}
$$

Recall that the canonical deconstruction property ensures that the reductions in eq. (4.3) are in the canonical form. Conider the recursive relation $l \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathtt{m}}(p)$ and the value trees of its two arguments shown in figs. 4.12a and 4.12b respectively. For $d = 1$, The above formulae reduces to the following conditions:

The value trees $\mathcal{V}(l_1 \sim_d l_2)$ and $\mathcal{V}(l_1 \approx_d l_2)$ contains a single (leaf) node with the boolean expressions given in eq. (4.3) and indentity entry transfer functions. The conversion algorithm presented in section 4.4.10 together with the handling of approximate recursive relations described above allows us to reduce a proof obligation P without recursive relations directly to its canonical form by – (a) constructing $\mathcal{V}(P)$ and (b) extracting the boolean expression at its root.

Bisimilarity of Value Trees

Recall that, $l_1 \sim l_2$ asserts exact equality of l_1 and l_2 up to an arbitrary depth i.e. $l_1 \sim l_2 \approx l_1 \sim_\infty l_2$. This boils down to checking equality of all pairwise paths between $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$. We start by giving a program interpretation of a value tree which allows us to repose $l_1 \sim l_2$ as a bisimilarity (and thus equivalence) check between $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$. In the program interpretation, the inputs include the free variables at the entry node ξ and each leaf node is interpreted as an observable action returning its associated value. In this interpretation, value trees are analogous to non-deterministic Control-Flow Graphs. To make the search for a bisimulation relation easier, we can peel $\mathcal{V}(l_1)$ and $\mathcal{V}(l_2)$ to unify their static tree structures. Once unified, the bisimulation relation requires us to prove that the outputs of each leaf node pair are identical. Similar to deconstruction programs, we run our points-to analysis before the check for bisimilarity to identify points-to invariants to help in the bisimulation search.



(a) $\mathcal{V}(\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C))$ **(b)** $\mathcal{V}(\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C))$

Node Pair	Invariants
$(\xi : \xi)$	(P) LHS in eq. (4.4)
$(0 : 0)$ $(1 : 1)$	(I1) $\omega_1 = \omega_2$ (I2) $\mathtt{p}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$ (I3) $\omega_1 \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$ (I4) $\omega_2 \rightsquigarrow \{\mathtt{C4}_1, \mathcal{H}\}$ (I5) $\mathtt{C4}_1 \rightsquigarrow \{\mathtt{C4}_{2+}\}$ (I6) $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$ (I7) $\mathcal{H} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$
$(\zeta_0 : \zeta_0)$	(E1) $() = ()$
$(\zeta_1 : \zeta_1)$	(E2) $\omega_1 \xrightarrow{\mathbb{m}}_{\mathtt{lnode}} \mathtt{val} = \omega_2 \xrightarrow{\mathbb{m}'}_{\mathtt{lnode}} \mathtt{val}$

(c) Invariants table for bisimulation relation between fig. 4.13a and fig. 4.13b.

Figure 4.13: Value trees of $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$ and $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C)$ along with the invariants table.

Recall the type III proof obligation illustrated in section 3.6:

$$\mathtt{LHS} \Rightarrow \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C) \sim \mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C) \tag{4.4}$$

Also recall the points-to invariants available at C5 (showing only the relevant ones): $\mathtt{p}_C \rightsquigarrow \{\mathtt{C4}_1\}$ $\mathtt{l}_C \rightsquigarrow \{\mathtt{C4}_{2+}\}$, $\mathtt{C4}_1 \rightsquigarrow \{\mathtt{C4}_{2+}\}$, $\mathtt{C4}_{2+} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$, and $\mathcal{H} \rightsquigarrow \{\mathtt{C4}_{2+}, \mathcal{H}\}$. Figure 4.13 shows the value trees of $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}}(\mathtt{l}_C)$ and $\mathtt{Clist}^{\mathtt{lnode}}_{\mathbb{m}'}(\mathtt{l}_C)$ along with the table of invariants required for a successful bisimulation check.

Chapter 5

Evaluation

We have implemented S2C on top of the Counter tool [26]. We use *four* SMT solvers running in parallel for solving SMT proof obligations discharged by our proof discharge algorithm: `z3-4.8.7`, `z3-4.8.14` [21], `Yices2-45e38fc` [22], and `cvc4-1.7` [1]. An unroll factor of *four* is used to handle loop unrolling in the C implementation. We use a default value of *eight* for over- and under-approximation depths (d_o and d_u). The default value of our unrolling parameter k (used for categorization of proof obligations) is *five*. We use a value of *five* for η (used by *StrongestInvCover()* during weakening of recursive relation invariants).

S2C requires the user to provide a Spec program S (specification), a C implementation C, and a file that contains their input-output specifications. For each function pair, S2C attempts to find equivalence between their CFGs $\mathcal{S}$ and $\mathcal{C}$ under their respective *Pre* and *Post* (given as part of input-output specification). An equivalence check requires the identification of lifting constructors to relate C values to the ADT values in Spec through recursive relations. Such relations may be required at the entry of both programs (i.e. in the precondition *Pre*), in the middle of both programs (i.e., in the invariants at intermediate product-CFG nodes), and at the exit of both programs (i.e., in the postcondition *Post*). *Pre* and *Post* are user-specified, whereas the inductive invariants are inferred automatically by our algorithm. During invariant inference, S2C derives the candidate lifting constructors from the user-specified *Pre* and *Post*.

More sophisticated approaches to finding lifting constructors are left as future work.

5.1 Experiments

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

Table 5.1: String lifting constructors and their definitions.

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

5.1.1 String

We wrote a single specification in Spec for each of the following common string library functions: `strlen`, `strchr`, `strcmp`, `strspn`, `strcspn`, and `strpbrk`. For each specification program, we took multiple C implementations of that program, drawn from popular libraries like `glibc` [3], `klibc` [4], `newlib` [7], `openbsd` [8], `uClibc` [9], `dietlibc` [2], `musl` [5], and `netbsd` [6]. Some of these libraries implement the same function in two ways: one that is optimized for code size and another that is optimized for runtime. All these library implementations use a *null character* terminated array to represent a string, and the corresponding lifting constructor is $\mathtt{Cstr}_{\mathbb{m}}^{\mathtt{u8[]}}$. u<N> represents the N-bit unsigned integer type in C. For example, `u8` represents `unsigned char` type.

Further, we implemented custom C programs for these functions that uses 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 $\mathtt{Cstr}_{\mathbb{m}}^{\mathtt{lnode(u8)}}$ and $\mathtt{Cstr}_{\mathbb{m}}^{\mathtt{clnode(u8)}}$ respectively. These lifting constructors are defined in table 5.1. $\mathtt{Cstr}_{\mathbb{m}}^{\mathtt{lnode(u8)}}$ requires a single argument p representing the pointer to the list node. On the other hand, $\mathtt{Cstr}_{\mathbb{m}}^{\mathtt{clnode(u8)}}$ 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.

An Example : strchr

Additionally, we define an optional string type `OptStr` to specify behaviour of functions that conditionally return a string (e.g., `strchr`, `strpbrk`). The `OptStr` ADT along with its three lifting constructors for the three layouts of the `Str` ADT are shown in table 5.1. `strchr` accepts a string t and a character c[1] and returns the longest substring of t that begins with c, otherwise it returns a null pointer to indicate failure to find c in the string t. In case c is the null character, `strchr` is defined to return the empty string (*not* the null value). Figures 5.1a and 5.1b shows the IRs of the `strchr` Spec specification and a generic C implementation respectively. We demonstrate

[1]Due to historical reasons, the type of c is declared as `int` to maintain backward compatibility with pre C-98 code. However, the function is specified to cast it to a character and use it instead.

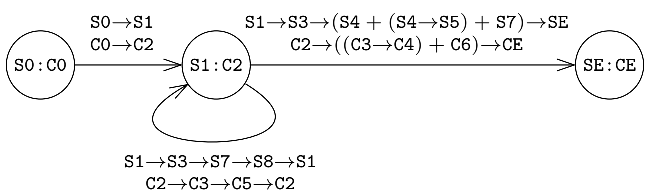
```
S0: OptStr strchr (Str s, i8 c) {
S1:   while true:
S2:     assume ¬(s is SInvalid);
S3:     if s is SNil:
S4:       if c = 0_i8: return Found(s);
S5:       return NotFound();
S6:     i8 ch ≔ s.ch; // (s is SCons)
S7:     if c = ch: return Found(s);
S8:     s ≔ s.tail;
SE: }
```

(a) Strchr Specification

```
    char* strchr(char* t, int c);

C0: i32 strchr (i32 t, i32 c) {
C1:   i8 ch ≔ c[7:0];
C2:   while t[0_i32]^i8_m ≠ ch:
C3:     if t[0_i32]^i8_m = 0_i8:
C4:       return 0_i32;
C5:     t ≔ t + 1_i32
C6:   return t;
CE: }
```

(b) Strchr Implementation using Array



(c) Product CFG for programs figs. 5.1a and 5.1b

PC-Pair	Invariants
(S0:C0)	(P1) $\mathtt{s}_S \sim \mathtt{Cstr}^{\mathtt{u8[]}}_{\mathtt{m}}(\mathtt{t}_C)$
	(P2) $\mathtt{c}_S = \mathtt{c}_C[7:0]$
(S1:C2)	(I1) $\mathtt{s}_S \sim \mathtt{Cstr}^{\mathtt{u8[]}}_{\mathtt{m}}(\mathtt{t}_C)$
	(I2) $\mathtt{c}_S = \mathtt{ch}_C$
(SE:CE)	(E) $\mathtt{ret}_S \sim \mathtt{Coptstr}^{\mathtt{u8[]}}_{\mathtt{m}}(\mathtt{ret}_C)$

(d) Invariants table for fig. 5.1c

Figure 5.1: Figures 5.1a and 5.1b shows the (abstracted) IRs of Spec specification and a generic array-based C implementation. Figure 5.1c presents the Product-CFG showing a bisimulation relation between figs. 5.1a and 5.1b. The node invariants for the product-CFG in fig. 5.1c are shown in fig. 5.1d.

two important aspects of S2C using this example – (a) use of ($\mathcal{S}$ `def`) and Pre to restrict the C implementation to only well-formed inputs (in section 2.3), and (b) need for correlating pathsets (instead of paths) (in section 4.2.1).

Recall that a null-character terminated C string is only well-formed if the string itself does not belong to a region of memory containing the null pointer. This wellformedness condition is necessary to prove that the pointer to the string returned in `C6` (in fig. 5.1b) does not equal the null pointer (used uniquely to indicate a failure to find the character c in the string t). As previously discussed in section 2.3, we expose this wellformedness condition in the specification using the explicit `Str` data constructor `SInvalid`. Finally, we assert that $\mathtt{s}_{\mathcal{S}}$ in fig. 5.1a is wellformed using the `assuming-do` statement (`S3` in fig. 5.1a) and relate the non-null wellformedness condition of the C input string $\mathtt{t}_{C}$ with the condition of $\mathtt{s}_{\mathcal{S}}$ being `SInvalid` using Pre (labeled P1 in fig. 5.1d). Note the use of $\mathtt{Coptstr}^{\mathtt{u8[]}}_{\mathtt{m}}$ in the postcondition (labeled E in fig. 5.1d).

Figure 5.1c shows the product-CFG showing the path correlations between $\mathcal{S}$ and $\mathcal{C}$. Consider the product-CFG edge $(\mathtt{S1:C2}) \rightarrow (\mathtt{SE:CE})$ correlating the pathsets: $\mathtt{S1}\rightarrow\mathtt{S3}\rightarrow(\mathtt{S4} + (\mathtt{S4}\rightarrow\mathtt{S5}) + \mathtt{S7})\rightarrow\mathtt{SE}$ (in $\mathcal{S}$) and $\mathtt{C2}\rightarrow((\mathtt{C3}\rightarrow\mathtt{C4}) + \mathtt{C6})\rightarrow\mathtt{CE}$ (in $\mathcal{C}$). The $\rightarrow$ and $+$ operators are used to represent 'series' and 'parallel' path combinations. The above two pathsets represent the following two sets $\{\mathtt{S1}\rightarrow\mathtt{S3}\rightarrow\mathtt{S4}\rightarrow\mathtt{SE}, \mathtt{S1}\rightarrow\mathtt{S3}\rightarrow\mathtt{S4}\rightarrow\mathtt{S5}\rightarrow\mathtt{SE}, \mathtt{S1}\rightarrow\mathtt{S3}\rightarrow\mathtt{S7}\rightarrow\mathtt{SE}\}$ and $\{\mathtt{C2}\rightarrow\mathtt{C3}\rightarrow\mathtt{C4}\rightarrow\mathtt{CE}, \mathtt{C2}\rightarrow\mathtt{C6}\rightarrow\mathtt{CE}\}$ respectively. In $\mathcal{S}$, the case of $\mathtt{c}_{\mathcal{S}}$ being the null character is handled explicitly in `S4` while `S7` handles the case where the string $\mathtt{s}_{\mathcal{S}}$ contains the (non-null) character $\mathtt{c}_{\mathcal{S}}$. However in $\mathcal{C}$, the above two cases are taken care of by the singular exit edge in `C6`. For a successful bisimulation proof, we are required to correlate the $\mathcal{C}$ path $\mathtt{C2}\rightarrow\mathtt{C6}\rightarrow\mathtt{CE}$ with the $\mathcal{S}$ pathset $\{\mathtt{S1}\rightarrow\mathtt{S3}\rightarrow\mathtt{S4}\rightarrow\mathtt{SE}, \mathtt{S1}\rightarrow\mathtt{S3}\rightarrow\mathtt{S7}\rightarrow\mathtt{SE}\}$. Such cases are rather frequent because the strongly-typed specification has to handle each case explicitly while its C implementation may take advantage of the underlying representation to generalize multiple explicit cases into one.

Another Example : strlen

Figure 5.2 shows the `strlen` specification and two vastly different C implementations. Figure 5.2b is a generic implementation using a null character terminated array to represent a string similar

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

(a) Strlen Specification

```
size_t strlen(char* s);

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

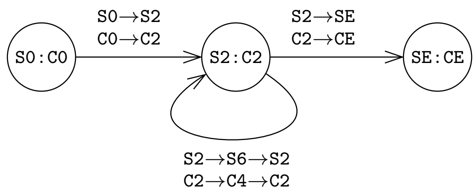
(b) Strlen Implementation using Array

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

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

(c) Optimized Strlen Implementation using Chunked Linked List

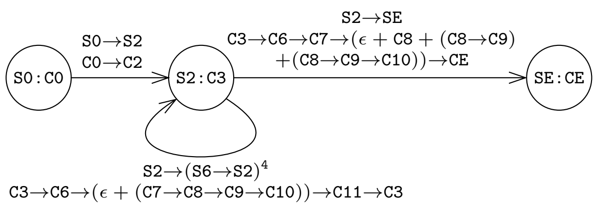
Figure 5.2: Specification of Strlen along with two possible C implementations. Figure 5.2b is a generic implementation using a null-terminated array for `String`. Figure 5.2c is an optimized implementation using a chunked linked list for `String`.



(a) Product CFG for programs figs. 5.2a and 5.2b

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

(b) Invariants Table for fig. 5.3a



(c) Product CFG for programs figs. 5.2a and 5.2c

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

(d) Invariants Table fig. 5.3c

Figure 5.3: Product CFGs and Invariants Tables showing bisimulation between Strlen specification in fig. 5.2a and two C implementations in figs. 5.2b and 5.2c

to a C-style string. The second implementation in fig. 5.2c differs from fig. 5.2b in the following: (a) it uses a chunked linked list data layout for the input string and (b) it uses specialized bit manipulations to identify a null character in a chunk at a time. S2C is able to automatically find a bisimulation relation for both implementations against the unaltered specification. Figure 5.3 shows the product-CFG and invariants for each implementation.

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

Table 5.2: List lifting constructors and their definitions.

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

5.1.2 List

We wrote a Spec program specification that creates a list, a program that traverses a list to compute the sum of its elements and a program that computes the dot product of two lists. We use three different data layouts for a list in C: array ($\texttt{Clist}_{\mathbb{m}}^{\texttt{u32[]}}$), linked list ($\texttt{Clist}_{\mathbb{m}}^{\texttt{lnode(u32)}}$), and a chunked linked list ($\texttt{Clist}_{\mathbb{m}}^{\texttt{clnode(u32)}}$). The lifting constructors are shown in table 5.2. Although similar to the String lifting constructors, these lifting constructors differ widely in their data encoding. For example, $\texttt{Clist}_{\mathbb{m}}^{\texttt{u32[]}}(p, i, n)$ represents a `List` value constructed from a C array p of size n starting at the i^{th} index. The list becomes empty when we are at the end of the array. ($\texttt{Clist}_{\mathbb{m}}^{\texttt{lnode(u32)}}$) and ($\texttt{Clist}_{\mathbb{m}}^{\texttt{clnode(u32)}}$), on the other hand, encodes empty lists (`LNil`) using *null pointers*. These layouts are in contrast to the `String` layouts, all of which uses a *null character* to indicate the empty string.

Table 5.3: Tree lifting constructors and their definitions.

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

5.1.3 Tree

We wrote a Spec program that sums all the nodes in a tree through an inorder traversal using recursion. We use two different data layouts for a tree: (1) a flat array where a *complete* binary tree is laid out in breadth-first search order commonly used for heaps ($\mathtt{Ctree}_{\mathbb{m}}^{\mathtt{u32[]}}$), and (2) a linked tree node with two pointers for the left and right children ($\mathtt{Ctree}_{\mathbb{m}}^{\mathtt{tnode(u32)}}$) (shown in table 5.3). Both Spec and C programs contain non-tail recursive procedure calls for left and right children. S2C is able to correlate these recursive calls using user-provided *Pre* and *Post* as discussed in section 4.2.3. At the entry of the recursive calls, S2C is required to prove that *Pre* holds for the arguments and at the exit of the recursive calls, S2C assumes *Post* on the returned states.

Table 5.4: Matrix and auxiliary List lifting constructors and their definitions.

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

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 5.4). The C implementations for a `Matrix` object include (a) a two-dimensional array ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[][]}}$), (b) a flattened row-major array ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[r]}}$), (c) a flattened column-major array ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[c]}}$), (d) a linked list of 1D arrays ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{lnode(u32[])}}$), (e) a 1D array of linked lists ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{lnode(u32)[]}}$) and (f) a 1D array of chunked linked list ($\texttt{Cmat}_{\mathbb{m}}^{\texttt{clnode(u32)[]}}$) data layouts. Note that both `T[r]` and `T[c]` represent a 1D array of type `T`. The r and c simply emphasizes that these arrays are used to represent matrices in row-major and column-major encodings respectively. We also introduce two auxiliary lifting constructors, $\texttt{Clist}_{\mathbb{m}}^{\texttt{u32[r]}}$ and $\texttt{Clist}_{\mathbb{m}}^{\texttt{u32[c]}}$ for lifting each row of matrices lifted using the corresponding $\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[r]}}$ and $\texttt{Cmat}_{\mathbb{m}}^{\texttt{u32[c]}}$ `Matrix` lifting constructors. These constructors are listed in table 5.4.

Table 5.5: Equivalence checking times and minimum under- and over-approximation depth values at which equivalence checks succeeded.

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

5.2 Results

Table 5.5 lists the various C implementations and the time it took to compute equivalence with their specifications. For functions that take two or more data structures as arguments, we show results for different combinations of data layouts for each argument. We also show the minimum under-approximation (d_u) and over-approximation (d_o) depths at which the equivalence proof completed (keeping all other parameters to their default values).

During the verification of `strchr` and `strpbrk` implementations, we identified an interesting subtlety. Since `strchr` and `strpbrk` return null pointers to signify absence of the required character(s) in the input string, we additionally need to model the UB assumption that the zero address does not belong to the null character terminated array representing the string. We use an explicit constructor `SInvalid` to expose this well-formedness property in a Spec `String`. Furthermore, we relate `SInvalid` to the condition of C character pointer being null using the lifting constructors $\mathtt{Cstr}^{\mathtt{T}}_{\mathbb{m}}(p\,{:}\,\mathtt{i32}, \ldots)$ (as defined in table 5.2). These lifting constructors are used as part of *Pre* to equate S and C input strings. Finally in S, we model the absence of `SInvalid` in the input string as a UB assumption using the `assuming-do` statement introduced in section 2.1. Due to the ($\mathcal{S}$ `def`) assumption, this constraints the inputs to S as well as C to well-formed strings only. This is an example where ($\mathcal{S}$ `def`) and *Pre* can be used to model wellformedness of values in C.

5.3 Limitations

S2C is not without limitations. Since S2C is only interested in finding a bisimulation relation, a whole class of non-bisimilar but equivalent program pairs is beyond our scope. S2C currently only supports bitvector affine and inequality relations, in addition to recursive relations based on the lifting constructors provided as part of *Pre* and *Post*. Consequently, non-linear bitvector invariants (such as polynomial invariants) are not supported. More importantly, S2C does not attempt to infer lifting constructors and merely uses the lifting constructors (with different arguments) provided as part of the input-output characteristics. While our correlation and invariant inference algorithms are based on the Counter tool [26] designed primarily for translation validation between (C-like) unoptimized IR and assembly, we found them to be quite effective for Spec to (C-like) IR as well. However, S2C inherits many of the limitations of the Counter tool. For example, S2C supports path specializations from Spec to C, but does not search for path merging correlations.

In case of our proof discharge algorithm, we have identified a source of inefficiency for type II proof obligations. For a recursive relation relating values of a non-linear ADT such as `Tree`, its d-depth approximation results in $\mathcal{O}(2^d)$ scalar equalities. This is a source of major inefficiency due to generation of large SMT queries. The completely of type III proof obligations is highly contingent on the precision of our points-to analysis on $\mathcal{C}$ as well as the deconstruction programs being checked for equivanece as part of the nested bisimulation check. We found our coarse-grained `{1,2+}` categorization of allocation recency combined with allocation-site based points-to analysis to be quite good at identifying required points-to invariants. However, such an abstraction is far from complete.

Chapter 6

Conclusion

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

Prior work on push-button verification of specific systems [16, 41, 39, 40] involves a combination of careful system design and automatic verification tools like SMT solvers. Constrained Horn Clause (CHC) Solvers [20] encode verification conditions of programs containing loops and recursion, and raise the level of abstraction for automatic proofs. Comparatively, S2C further raises the level of abstraction for automatic verification from SMT queries and CHC queries to automatic discharge of proof obligations involving recursive relations.

A key idea in S2C is the conversion of proof obligations involving recursive relations to bisimulation checks. Thus, S2C performs *nested* bisimulation checks as part of a 'higher-level' bisimulation search. This approach of identifying recursive relations as invariants and using bisimulation to discharge the associated proof obligations may have applications beyond equivalence checking.

Bibliography

[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] Lars Ole Andersen. Program analysis and specialization for the C programming language. Technical report, 1994.

[11] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, page 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. Tvoc: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K.

Rajamani, editors, *Computer Aided Verification*, pages 291–295, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[13] Nick 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.

[14] Benjamin C. *Types and Programming Languages*. MIT Press, 2002.

[15] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 296–310, New York, NY, USA, 1990. Association for Computing Machinery.

[16] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai 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.

[17] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex 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.

[18] Jeffrey Considine. Efficient hash-consing of recursive types. Technical report, USA, 2000.

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

[20] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through horn clause transformation. In Xavier Rival, editor, *Static Analysis*, pages 147–169, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[21] Leonardo De Moura and Nikolaj 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.

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

[23] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias 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.

[24] Cormac Flanagan and K. Rustan 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.

[25] Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, page 150–161, New York, NY, USA, 2004. Association for Computing Machinery.

[26] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[28] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 66–74, New York, NY, USA, 1982. Association for Computing Machinery.

[29] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Validation of gcc optimizers through trace generation. *Softw. Pract. Exper.*, 39(6):611–639, April 2009.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon 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.

[31] Sudipta Kundu, Zachary Tatlock, and Sorin 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.

[32] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[33] 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.

[34] Nuno P. Lopes and José Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *Int. J. Softw. Tools Technol. Transf.*, 18(4):359–374, August 2016.

[35] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[36] Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 46–60, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[37] KedarS. Namjoshi and LenoreD. Zuck. Witnessing program transformations. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 304–323. Springer Berlin Heidelberg, 2013.

[38] George 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.

[39] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In Tim Brecht and Carey 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.

[40] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 41–61. USENIX Association, 2020.

[41] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi 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.

[42] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

[43] Arnd Poetzsch-Heffter and Marek Gawkowski. Towards proof generating compilers. *Electron. Notes Theor. Comput. Sci.*, 132(1):37–51, May 2005.

[44] Robert Harper Robin Milner, Mads Tofte and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[45] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin 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.

[46] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex 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.

[47] Michael Stepp, Ross Tate, and Sorin 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.

[48] Ofer Strichman and Benny Godlin. Regression verification - a practical way to verify programs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 496–501. Springer Berlin Heidelberg, 2008.

[49] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin 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.

[50] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.

[51] Anna Zaks and Amir 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.

[52] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A methodology for the translation validation of optimizing compilers. 9(3):223–247, mar 2003.

[53] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27(3):335–360, November 2005.

Biography

Indrajit Banerjee is a MS(R) student of Computer Science & Engineering Department at IIT Delhi. He couldn't think of anything interesting to add here.