Recursive Assertions for Data Structures

Joxan Jaffar Andrew E. Santosa Răzvan Voicu
School of Computing, National University of Singapore
joxan,andrews,razvan@comp.nus.edu.sg

Abstract

We present an assertion language for expressing properties of data structures. Its key features are *constraints* over arrays, multisets and integers which allow the specification of basic assertions, and *rules*, which allow the recursive specification of assertions. This language can thus be used to define assertions to an arbitrary level of expressiveness, ranging from low-level properties of memory allocation, for example, to abstract properties of complex data structures such as AVL trees.

The main result is a proof method for verification conditions arising from a program annotated with assertions. The method has two main components. First and foremost is an unfolding algorithm which works by reducing the recursive definitions so that a constraint proof may now be applied. Here we introduce a notion of *coinduction* which forms the basis for termination of the unfolding process. The second step is to reduce the constraints, which in general contain expressions involving all the three data types of integers, arrays and multisets, into a base constraint involving only integer constraints. Base constraints can then be dispensed with available solvers.

We finally show via a small benchmark of classic examples that our proof method is practical.

1. Introduction

Reasoning about programs which construct and manipulate mutable data structures remains an open problem in the sense that present methods are limited in applicability, and that they do not scale well to large programs.

A traditional challenge is how to implement a notion of *closure*, such as transitive closure. Typically, there is no closed form to describe a typical class of data structures, for example, the acyclic singly-linked lists. Therefore, in order to specify that a variable points to such a structure, one would require an inductive or recursive formulation. Indeed, such a class of data structures is often called "recursive" in the literature. For example, to prove that a cell q is reachable from a cell p one would some formulation of the reachability closure of p.

Another traditional challenge concerns *aliasing*, the problem of reasoning about two pointers which may, or definitely do not, point to the same data structure. For example, one specific challenge is to determine, when a data structure pointed to by one particular pointer is changed, what the effect is on all other pointers. Some approaches focus on maintaining non-aliasing information. Thus, for example, after operations are performed on a data structure pointed to by p, we may reason that no change has taken place on the structure of another pointer q. Conversely, there is also need to consider explicit aliasing information. For example, if q points to the third cell of an acyclic list p, and if a three-step traversal of p results in r, we would require that $q = r$.

The most important challenge of all, however, is to capture *abstract* properties of data structures in such a way that the formal techniques are in tandem with the intuitive reasoning embodied in the user program.

In this paper, we address these issues, amongst others, by first defining a language of array, multiset and integer expressions. The class of integer expressions includes both array elements and array indices. These basic formulas can describe basic and detailed properties about mutable heaps and pointers. We then embed this formalism in Constraint Logic Programming (CLP) so that CLP predicates can be used to describe recursive properties of data structures. This formulation of recursion then provides for the specification of basic closure properties, amongst other properties. Further, because the CLP formalism has a well-understood logical reading, assertion predicates can be designed to represent abstract properties of data structures. At the same time, low-level specifications, such as pointer arithmetic or memory management operations, can be represented by the rich constraint language. In particular, our formalism supports a notion of *separation* [19] by simply using multiset constraints to specify that certain heaps do not intersect.

The main contribution is a proof method for CLP assertion predicates. We present an algorithm which is based on a standard notion of unfolding definitions. The main novelty is the use of "left and right" unfolding, augmented with a principle of *coinduction* which forms the basis for terminating the unfolding process. This unfolding process ultimately reduces the proof obligation to another that no longer contains (recursive) assertion predicates. That is, what remains is to prove a constraint.

The secondary contribution is an algorithm for proving a constraint. The algorithm reduces the proof of a constraint involving array, multiset and integer constraints, into a proof involving only integer constraints. The novelty in our approach is in how the array and multiset constraints are arithmetized, that is, converted into equivalent integer constraints. At this point, the remaining proof obligation can be dispensed with standard constraint solvers.

We finally argue, via examples, that our proof methodology is intuitive and expressive, and amenable to a practical implementation. We show via a small benchmark of classic examples that our automatic proof method is in fact practical.

1.1 Related Work

The use of proof rules for proving properties of user-defined predicates in a CLP-based setting has been widely explored [11, 4, 16, 22]. For example, the "negation as failure" inference in [11] is akin to our left unfold rule, while the "definite clause inference" step in [11, 12] is akin to our right-unfold step below. In [22], fold/unfold transformations are performed toward the objective of transforming two programs into syntactically identical ones. All these approaches are based on some form of structural and/or computational induction.

In this respect, one major difference of our algorithm is that it is based on a coinduction principle, which does not require a base case. Recent work [2] provides a method for proving the equiva-

lence of general CLP programs that makes use of a coinduction rule. However, this work does not address data structures.

A main difference of our work from all these is that our proof method is systematic. Another difference is that our domain of discourse is a *constraint* domain of arrays, multisets and integers. In contrast, these other works are based on traditional logic programming, or do not directly accommodate properties of data structures.

The area of *shape analysis* adopts an abstract interpretation-based approach, and is surveyed in [24]. Here the focus is on the accuracy and efficiency trade-off involving the abstract domain (which is constructed of predicates that define the "shape" of the data structure), and the fixpoint iteration algorithm. As argued in [3], it is rather difficult to construct modular, interprocedural shape analyses, since after every memory update, all reachability relations have to be recomputed. Attempts to introduce local reasoning into shape analysis are presented in [20, 21]. Also, [10] propose an interprocedural shape analysis that represents each procedure as a rather coarse abstraction of its input-output relation.

Next we mention some other works specialized on reasoning about data structures in customized ways.

Other approaches to data structure verification include the approaches based on *graph types* [13, 17], which is based on Hoare logic, and PALE [17] verifier can be efficiently run when loop invariant is given. The paper [15] presents an algorithm for specification and verification of data structure using equality axioms. It has a support for scalar values as compared to most works on shape analysis.

None of the above works on shape analysis and customized reasoning about data structures allow recursive definitions provided by the user. A recent exception is [18] which considers a class of pointer operations augmented with a separation construct, and allows user-defined shape properties. They employ folding and unfolding rules, whereas we employ unfolding alone, augmented with a coinduction rule. They do not consider arrays.

Finally, we comment on our design decision on choosing arrays and multisets over integers. The case for arrays is clear, because a heap is essentially an array. We chose the domain of multisets for two main purposes: to specify separation [19], and permutation. Our algorithm extends the early works [6, 8]. In addition to multisets, one could also consider modeling array segments as sets or sequences. However, there are no known efficient algorithms for these. For sets, the problem at hand would essentially be that faced by ACI unification, see eg. [1]. For sequences, the problem is even harder than reasoning about word equations, see eg. [7].

2. The Language

2.1 Basic Assertions

We consider three kinds of terms: integer, array and multiset terms. Integer terms are constructed in the usual way, with one addition: the array element. The latter is defined recursively to be of the form $a[i]$ where a is an *array expression* and i an integer term. An array expression is either an array variable or of the form $\langle a,i,j\rangle$ where a is an array expression and i,j are integer terms. A multiset term is either a singleton multiset $\{i\}$ where i is an integer variable, a multiset variable, or it is constructed from an array "segment": $a\{i..j\}$ where a is an array expression and i,j integer variables.

The meaning of an array expression is simply a map from integers into integers, and the meaning of an array expression $a' = \langle a,i,j\rangle$ is a map just like a except that $a'[i] = j$. The meaning of array elements is governed by the classic McCarthy [14] axioms:

$$i = k \;\rightarrow \langle a,i,j\rangle[k] = j$$
$$i \neq k \;\rightarrow \langle a,i,j\rangle[k] = a[k]$$

The meaning of a singleton multiset is obvious, and the meaning of a multiset term of the form $a\{i..j\}$ is the multiset of array elements $\{a[i],a[i+1],\cdots,a[j]\}$.

A *constraint* is either an integer equality or inequality, an equation between array expressions, or a *multiset constraint*. The latter is of one of the forms:

- $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$
- $\mathcal{M}_1 \otimes \mathcal{M}_2 \otimes \cdots \otimes \mathcal{M}_n, \quad n \geq 2$

The purpose of the first form is clear, to allow the propagation of equational reasoning between multiset terms constructed naturally via multiset union. The latter form, which in fact defines a family of n-ary constraints $\otimes$, specifies that the multisets $\mathcal{M}_i, 1 \leq i \leq n$, are *disjoint*, That is, each element appearing in one multiset does not appear in the other. As we shall see later, this constraint is specially introduced in order to capture the notion of *separation* between the cells of two different data structures.

The meaning of a constraint is defined in the obvious way.

In what follows, we use constraint to mean either an atomic constraint or a conjunction of constraints. We shall use the symbol ψ or Ψ, with or without subscripts, to denote a constraint.

2.2 Constraint Logic Programs

We present some preliminary definitions about CLP [9]. An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ a tuple of terms, as defined above. A *rule* is of the form A: -$\Psi,\tilde{B}$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and constraint Ψ constitute the *body* of the rule. A *program* is a finite set of rules. A *goal* has exactly the same format as the body of a rule. A goal that contains only constraints and no atoms is called *final*.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. A *grounding* is a substitution which maps each array, multiset or integer variable into its intended universe of discourse: an array, a multiset or an integer. Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$. A grounding θ of a goal $\mathcal{G} \equiv (p(\tilde{t}),\Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $[\![\mathcal{G}]\!]$ to denote the set of groundings of $\mathcal{G}$.

Let $\mathcal{G} \equiv (B_1,\cdots,B_n,\Psi)$ and P denote a non-final goal and program respectively. Let $R \equiv A$: -$\Psi_1,C_1,\cdots,C_m$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of $\mathcal{G}$ using a rule R, denoted $reduct(\mathcal{G},R)$, is of the form

$(B_1,\cdots,B_{i-1},C_1,\cdots,C_m,B_{i+1},\cdots,B_n,B_i = A,\Psi,\Psi_1)$

provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable.

A *derivation sequence* for a goal $\mathcal{G}_0$ is a possibly infinite sequence of goals $\mathcal{G}_0,\mathcal{G}_1,\cdots$ where $\mathcal{G}_i, i > 0$ is a reduct of $\mathcal{G}_{i-1}$. If the last goal $\mathcal{G}_n$ is a final goal, we say that the derivation is *successful*. A *derivation tree* for a goal is defined in the obvious way.

DEFINITION 1 (Unfold). *Given a program P and a goal $\mathcal{G}$,* UNFOLD$(\mathcal{G})$ *is* $\{\mathcal{G}' | \exists R \in P : \mathcal{G}' = reduct(\mathcal{G},R)\}$. □

In the formal treatment below , we shall assume, without losing generality, that goals are written so that atoms contain only distinct variables as arguments.

2.3 Assertions in CLP

A basic assertion is expressed directly in CLP as a constraint. We shall adopt the following convention: if the structure at hand is a list and I is a pointer to a structure, then heap location $I+1$ represents the "next" pointer of the list. Similarly, if the structure I has two "next" pointers, such as in a binary tree, the heap locations $I+1$ and $I+2$ shall represent these pointers. In general, if the structure of interest has multiple pointers, we use $I+1, I+2, \cdots$, etc.

An assertion can, more generally, be a *recursive* assertion. This can be represented as any CLP program. However, for this paper, we shall limit ourselves to two slightly specialized classes. First consider *one-heap* assertions whose rules are of the form:

$$p(H,\tilde{X}) \;\text{ :- }\; \Psi_A,\; p(H',\tilde{X}').$$
$$p(H,\tilde{X}) \;\text{ :- }\; \Psi_B.$$

Program:

```
        {l = l0 ≥ 0}
        i,x,y:=2,1,0
⟨0⟩     while (i≤l) do
⟨1⟩         tmp:=x ⟨2⟩ x:=x+y
⟨3⟩         y:=tmp ⟨4⟩ i:=i+1 end ⟨5⟩
        {fib(l0,x)}
```

Assertion Predicate:

$fib(0,0).$
$fib(1,1).$
$fib(I,X+Y)$:- $I \geq 2, fib(I-1,X), fib(I-2,Y).$

Figure 1: Fibonacci Number Generator

Program:

```
    i=0;
⟨0⟩ while (i<N-1) do
⟨1⟩     j:=0
⟨2⟩     while (j < N-1-i) do
⟨3⟩         if ([j+1]<[j]) then
                t:=[j+1] [j+1]:=[j] [j]:=t endif
⟨4⟩         j:=j+1 end
⟨5⟩     i:=i+1 end ⟨6⟩
```

Assertion Predicates:

$sorted(H,I,N)$:- $I = N.$
$sorted(H,I,N)$:- $I < N, H[I] \leq H[I+1], sorted(H,I+1,N).$

$max(H,Y,U)$:- $0 > Y.$
$max(H,Y+1,U)$:- $0 \leq Y+1, H[Y+1] \leq U, max(H,Y,U).$

Figure 2: Bubble Sort

where H, H' are array expressions representing the previous and next versions of the global heap, and $\tilde{X}, \tilde{X}'$ are sequences of variables, some of which represent the program variables. The assertion $p(H,\tilde{X})$ specifies that the variables $\tilde{X}$ satisfy some property a in the heap H.

Similarly, a *two-heap* assertion is one whose rules are of the form:

$p(H_1,H_2,\tilde{X})$:- Ψ_A, $p(H_1',H_2',\tilde{X}')$.
$p(H_1,H_2,\tilde{X})$:- Ψ_B.

where $p(H_1,H_2,\tilde{X})$ specifies a relationship between heaps H_1 and H_2, typically that H_2 is an update of H_1.

Predicates such as p are called *assertion predicates* and correspondingly, atoms $p(H,\tilde{X})$ are called *assertion atoms*. We will provide a few examples in Section 3.

With some loss of generality, but no substantial loss, we shall assume that any *local* variable in a rule, one that appears in the body but not the head, appears as an argument of one of its body atoms.

3. Example Assertion Predicates

3.1 An Integer Example: Fibonacci

The annotated program in Figure 1 computes in x the l-th Fibonacci number. Fibonacciness is defined by the assertion predicate $fib(a,b)$. We prove the correctness of the program in Section 6.1.

3.2 An Array Example: Bubblesort

Here we consider array segments and multisets. Consider the bubble sort program and definitions of the predicates *max* and *sorted* in Figure 2. The CLP definition of $sorted(H,I,N)$ is a one-heap predicate that specifies that the sequence of cells $H[I], H[I+1], \cdots, H[N]$, if $I < N$, is an ordered sequence. The predicate $max(H,Y,U)$ is *true* if U is an upper bound of the values $H[0], \cdots, H[Y]$.

Program:

```
    {h = h0, p = p0 > 0}
⟨0⟩ while (p>0) do
        [p] := 0 ⟨1⟩
        p := [p+1] ⟨2⟩ end ⟨3⟩
    {∃y.allz(h0,h,p0,y), h[y+1] = 0}
```

Assertion Predicates:

"Tail Recursive"
$allz(H,\langle H,L,0\rangle,L,L)$:- $L > 0.$
$allz(H_1,\langle H_2,L,0\rangle,L,R)$:- $L > 0, allz(H_1,H_2,H_1[L+1],R).$

"Sublist Recursive"
$allz(H,\langle H,L,0\rangle,L,L)$:- $L > 0.$
$allz(H_1,\langle H_2,R,0\rangle,L,R)$:- $R > 0, R = H_2[T+1],$
$allz(H_1,H_2,L,T).$

Figure 3: List Reset

We will later exemplify two proofs of the inner loop B (between ⟨2⟩ and ⟨5⟩ of the bubble sort program in Figure 2). The "Hoare triples" are:

$$\begin{array}{l}\{\mathtt{j} = 0, 0 \leq \mathtt{i} < \mathtt{n}-1, max(\mathtt{n}-\mathtt{i}-1, \mathtt{n}-\mathtt{i}),\\ \ sorted(\mathtt{h}, \mathtt{n}-\mathtt{i}, \mathtt{n})\}\\ \qquad B\\ \{0 \leq \mathtt{i} < \mathtt{n}-1, max(\mathtt{n}-\mathtt{i}-2, \mathtt{n}-\mathtt{i}-1),\\ \ sorted(\mathtt{h}, \mathtt{n}-\mathtt{i}-1, \mathtt{n})\},\end{array} \quad (1)$$

and

$$\begin{array}{l}\{\mathtt{j} = 0, 0 \leq \mathtt{i} < \mathtt{n}-1, \mathtt{h} = h_0\}\\ \qquad B\\ \{0 \leq \mathtt{i} < \mathtt{n}-1, h_0\{\mathtt{0}\ldots\mathtt{n\text{-}1}\} = \mathtt{h}\{\mathtt{0}\ldots\mathtt{n\text{-}1}\}\}\end{array} \quad (2)$$

The condition (1) states that given the array elements from n-i to n is sorted before B, the execution results in a sorted array from n-i-1 to n. It also specifies the upper bounds of certain array segments. The condition (2) states that at the end B's execution, the values in the array is a permutation of the original array. We note here that equality between array segments above is the multiset equality, that is, the equality holds iff the multiset of the elements of the lhs array segment is the same as those of the rhs array segment. We outline the proofs in Section 6.2.

3.3 Examples using Pointers

List Reset

Figure 3 shows a program which "zeroes" all elements of a given linked list with head p. The correctness assertion states that given a nonempty list, the program produces a nonempty null-terminating list upon its termination, with all values in the nodes set to zero. Note that in Figure 3, h is a program variable denoting the current heap. The assertions use the predicate $allz(H,H',L,R)$ which states that the heap H' differs from H only by having zero elements in the non-empty sublist from L to R.

In Figure 3 we provide two different definitions of *allz*. The *tail-recursive* version defines a zeroed list segment (L,R) as one whose head contains zero, and its tail is, recursively, the zeroed list segment $(H[L+1],R)$. In the *sublist-recursive* specification, a zeroed list segment (L,R) is defined to be a zeroed list segment (L,T) appended by one extra zero element R. Note that we have not required that $L \neq R$ in either of the definitions of *allz*., because we do not require that the list is acyclic.

Clearly the program behaves in consistency with the latter definition, and not the former. Despite this, we see later in Section 3.3 that we can provide a proof using *either* definition.

Program:

$\{alist(h_0, i_0, m_0)\}$
```
j:=0
⟨0⟩ while (i>0) do
⟨1⟩     k,[i+1],j:=[i+1],j,i
        i:=k end ⟨2⟩
```
$\{reverse(h_0, \mathtt{h}, i_0, 0, \mathtt{j}), alist(\mathtt{h}, \mathtt{j}, m_0)\}$

Assertion Predicates:

$reverse(H, H, I, I, 0).$
$reverse(H_1, \langle H_2, J+1, NewNext\rangle, I, OldNext, J)$:-
$H_2[J+1] = OldNext,$
$reverse(H_1, H_2, I, J, NewNext).$

$alist(H, L, \emptyset)$:- $L = 0.$
$alist(H, L, S \cup \{L, L+1\})$:- $L > 0,$
$\{L, L+1\} \otimes S, alist(H, H[L+1], S).$

Figure 4: List Reverse

Program:

$\{avl(\mathtt{h}, [\mathtt{x+2}], dl_0 - 2, \mathtt{s2}),$
$avl(\mathtt{h}, [[\mathtt{x+1}]+1], dl_0 - 1, \mathtt{s11}),$
$avl(\mathtt{h}, [[\mathtt{x+1}]+2], dl_0 - 2, \mathtt{s12}),$
$m_0 = \{\mathtt{x}, \mathtt{x+1}, \mathtt{x+2}\} \cup \{[\mathtt{x+1}], [\mathtt{x+1}]+1, [\mathtt{x+1}]+2\}$
$\cup \mathtt{s2} \cup \mathtt{s11} \cup \mathtt{s12}\},$
$\{\mathtt{x}, \mathtt{x+1}, \mathtt{x+2}\} \otimes \{[\mathtt{x+1}], [\mathtt{x+1}]+1, [\mathtt{x+1}]+2\}$
$\otimes \mathtt{s2} \otimes \mathtt{s11} \otimes \mathtt{s12}\}$
```
⟨0⟩ y := [x+1]
⟨1⟩ if ([y] = 1) then
⟨2⟩     [x] := 0
⟨3⟩     [y] := 0
⟨4⟩     z := [y+2]
⟨5⟩     [y+2] := x
⟨6⟩     [x+1] := z ⟨7⟩
        {avl(h,y,dl0,m0)}
    endif
```

Assertion Predicate:

$avl(H, 0, 0, \emptyset).$
$avl(H, X, D_1 + 1, \{X, X+1, X+2\} \cup S_1 \cup S_2)$:-
$H[X] = D_1 - D_2, 0 \leq D_1 - D_2, D_1 - D_2 \leq 1,$
$\{X, X+1, X+2\} \otimes S_1 \otimes S_2,$
$avl(H, H[X+1], D_1, S_1), avl(H, H[X+2], D_2, S_2).$
$avl(H, X, D_2 + 1, \{X, X+1, X+2\} \cup S_1 \cup S_2)$:-
$H[X] = D_1 - D_2, D_1 - D_2 = -1,$
$\{X, X+1, X+2\} \otimes S_1 \otimes S_2,$
$avl(H, H[X+1], D_1, S_1), avl(H, H[X+2], D_2, S_2).$

Figure 5: AVL Tree

List Reverse

The CLP program for $reverse(H_1, H_2, I_1, I_2, J)$ in Figure 4 describes a two-heap predicate. It states that the linked list in heap H_1 starting with I_1 up to but not including I_2 is the reverse of that of the null-terminated list in the heap H_2 which starts from cell J. The array updates in the specification is used to specify that the list H_2 is an update of the list H_1, hence the reverse operation is *in-situ*. The CLP program for $alist(H, L, S)$ defines an acyclic list whose set of node addresses is S.

In section 6.4 we prove that given an acyclic list with head `i`, we obtain a list with head `j` which is a reverse of the original list.

AVL Tree

This example concerns Figure 5 which is a re-balancing routine of an AVL tree after node insertion. AVL is a balanced binary tree, where for each node, the depth of its left and right subtrees differs by only one. Here we demonstrate quantitative reasoning on an abstract data structure. Thus our proof method can be compared with that of [23], who introduced a specialized abstraction to reason about scalar values in data structures.

The rebalancing routine is given an unbalanced subtree rooted at `x`, where its left subtree is two deeper than its right subtree, and at its left child, the left subtree is one deeper than its right subtree. At point ⟨7⟩ we expect to obtain as output a balanced AVL tree.

The CLP program for $avl(H, X, D, S)$ describes a one-heap assertion predicate. It states that the binary tree X in H is an AVL tree of height D, with S as the set of all node addresses in the tree. We discuss a proof that the program preserves AVL structure in Section 6.5.

3.4 On Separation Logic

Recent work on verification of programs with shared mutable data structures [19] introduced the concept of *separation* as a means to simplify the reasoning process and make program correctness proofs less tedious. The separating connectives provide elegant and concise means of specifying that a set of data structures are not shared, or that the elements of a data structure are not reachable from within another data structure.

The general idea is that heap predicates Ψ_i may be combined, in pairs or in a tuple, in the form $\Psi_1 \star \Psi_2 \star \cdots \star \Psi_n$, $n \geq 2$ by a *separation* operator $\star$. The interpretation of a heap predicate is just a heap in which the predicate is true. The interpretation of $\Psi_1 \star \Psi_2 \star \cdots \star \Psi_n$ is a collection of n *disjoint* heaps H_i in which Ψ_i holds, $2 \leq i \leq n$.

In our framework, we achieve this by creating an assertion predicate p_i which defines the heap predicate Ψ_i, and explicitly mentions its heap locations as a multiset variable $\mathcal{M}_i$. Then, we simply add the constraint

$$\mathcal{M}_1 \otimes \mathcal{M}_2 \otimes \cdots \otimes \mathcal{M}_n$$

We shall exemplify this below.

4. Proof Method for Recursive Assertions

In this key section, we consider proof obligations of the form $\mathcal{G} \models \mathcal{H}$ where $var(\mathcal{H}) \subseteq var(\mathcal{G})$. The validity of this formula expresses the fact that $\mathcal{H}\theta$ succeeds w.r.t. the CLP program at hand whenever $\mathcal{G}\theta$ succeeds, for any grounding θ of $\mathcal{G}$. They are the central concept of our proof system, by being expressive enough to capture interesting properties of data structures, and yet amenable to automatic proof process.

The general idea is to reduce the proof obligation into one that can be proven by using the constraint solver alone. Essentially, this involves removing all occurrences of assertion predicates in the obligation. In general, however, this method is not always applicable to the obligation at hand. That is, upon predicate removal, the constraint proof fails. Then it is necessary to reduce the obligation to another obligation upon which the constraint proof can be attempted again. This reduction process, which constitutes a search process, is based on a standard notion of unfolding the definitions of assertion predicates contained in the obligation.

This section describes the unfolding rules. In the following section, we describe the remaining part of the overall proof method, that which proves constraints.

4.1 Unfolding Recursive Assertions

Intuitively, we proceed as follows: unfold $\mathcal{G}$ completely a finite number of steps in order to obtain a "frontier" containing the goals $\mathcal{G}_1, \ldots, \mathcal{G}_m$. Then unfold $\mathcal{H}$, but this time not necessarily completely, that is, not necessarily obtaining *all* the reducts each time, obtain goals $\mathcal{H}_1, \ldots, \mathcal{H}_m$. This situation is depicted in Figure 6. Then, the proof holds if

$$\mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n \models \mathcal{H}_1 \vee \ldots \vee \mathcal{H}_m$$

or alternatively, $\mathcal{G}_i \models \mathcal{H}_1 \vee \ldots \vee \mathcal{H}_m$ for all $1 \leq i \leq n$. This follows easily from the fact that $\mathcal{G} \models \mathcal{G}_1 \vee \ldots \vee \mathcal{G}_n$, and $\mathcal{H}_j \models \mathcal{H}$ for all

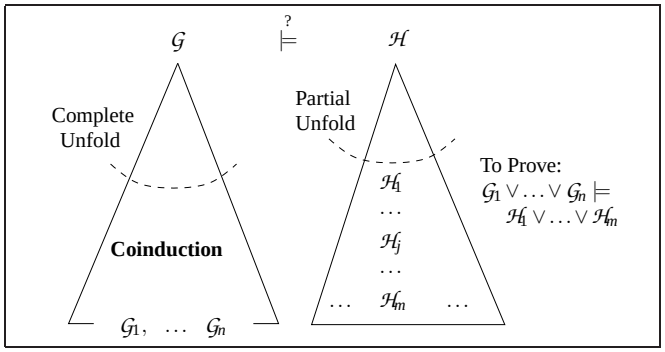

Figure 6: Informal Structure of Proof Process

j such that $1 \leq j \leq m$. More specifically, but with some loss of generality, the proof holds if

$$\forall i : 1 \leq i \leq n, \exists j : 1 \leq j \leq m : \mathcal{G}_i \models \mathcal{H}_j$$

and for this reason, our *proof obligation* shall be defined below to be simply a pair of goals, written $\mathcal{G}_i \models \mathcal{H}_j$.

4.2 Proof Rules

We now present a formal calculus for the proof of $\mathcal{G} \models \mathcal{H}$. To handle the possibly infinite unfoldings of $\mathcal{G}$ and $\mathcal{H}$, we shall depend on the use of a key concept: *coinduction*. Proof by coinduction allows us to assume the truth of a *previous* obligation.

The proof process starts with a set of proof obligations and attempts to discharge them one by one (although at times the set may in fact become larger).

DEFINITION 2 (Proof Obligation). *A* proof obligation *is of the form $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$ where the $\mathcal{G}$ and $\mathcal{H}$ are goals and $\tilde{A}$ is a set of* assumption *goals.* □

The role of proof obligations is to capture the state of a proof. The set $\tilde{A}$ contains goals whose truth can be assumed coinductively to discharge the proof obligation at hand.

Our proof rules are presented in Figure 7. The $\uplus$ symbol represents the disjoint union of two sets, and emphasizes the fact that in an expression of the form $A \uplus B$, we have that $A \cap B = \emptyset$. Each rule operates on the (possibly empty) set of proof obligations Π, by selecting one of its proof obligations and attempting to discharge it. In this process, new proof obligations may be produced.

The *left unfold with coinduction* (LU+C) rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π, is added as an assumption to every newly produced proof obligation, opening the door to using coinduction in the proof. The rule *right unfold* (RU) performs an unfold operation on the rhs of a proof obligation. In general, the two unfold rules will be systematically interleaved. The resulting proof obligations are then discharged either coinductively or directly, using the (CO) and (CP) rules, respectively.

The rule *coinduction application* (CO) transforms an obligation by using an assumption, and thus opens the door to discharging that obligation via the direct proof (CP) rule. Since assumptions can only be created using the (LU+C) rule, the (CO) rule realizes the coinduction principle. The underlying principle behind the (CO) rule is that a "similar" assertion $\mathcal{G}' \models \mathcal{H}'$ has been previously encountered in the proof process, and assumed as true[1].

Note that this test for coinduction applicability is itself of the form $\mathcal{G} \models \mathcal{H}$. However, the important point here is that this test can only be carried out using constraints, in the manner prescribed for the CP rule described below. In other words, this test does not use the definitions of assertion predicates.

[1] In fact, the repeating pattern corresponds to a loop in the original programming language, and $\mathcal{H}$ acts as an invariant.

$$\text{(LU+C)}\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}} \quad \text{UNFOLD}(\mathcal{G}) = \{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$$

$$\text{(RU)}\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{1 \leq i \leq k} \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}} \quad \mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$$

$$\text{(CO)}\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{H}'\theta \models \mathcal{H}\}} \quad \mathcal{G}' \models \mathcal{H}' \in \tilde{A} \text{ and there exists a substitution } \theta \text{ s.t. } \mathcal{G} \models \mathcal{G}'\theta$$

$$\text{(CUT)}\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G}' \models \mathcal{H},\ \tilde{A} \vdash \mathcal{G} \models \mathcal{G}'\}}$$

$$\text{(SPL)}\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^{k} \{\tilde{A} \vdash \mathcal{G} \wedge \psi_i \models \mathcal{H}\}} \quad \psi_1 \vee \ldots \vee \psi_k \text{ is true.}$$

$$\text{(CP)}\quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})\}}{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}\}}$$

Figure 7: Proof Rules for Reduction into Constraints

The (CUT) rule is manual and hence not used by our automatic algorithm. It is included here because it is particularly useful for strengthening an obligation. Indeed, given a proof obligation $\mathcal{G} \models \mathcal{H}$, it is often the case that $\mathcal{H}$ is too weak to result in applications of the (CO) and (CP) rules that would lead to a successful proof. To address this, the (CUT) rule introduces a new goal $\mathcal{G}'$ and the new proof obligations $\mathcal{G} \models \mathcal{G}'$ and $\mathcal{G}' \models \mathcal{H}$.

The rule *split* (SPL) rule is also manual and hence not used by our automatic algorithm. It is particularly useful for converting a proof obligation into several, more specialized ones.

Finally, the rule *constraint proof* (CP), when used repeatedly, discharges a proof obligation by reducing it to a form which contains no assertion predicates. Note that one application of this removes one occurrence of a predicate $p(\tilde{y})$ appearing in the rhs of an obligation. Once a proof obligation has no predicate in the rhs, a constraint proof may be attempted by simply removing any predicates in the corresponding lhs. Such a constraint is handled by the proof method for constraints in the following section.

Given a proof obligation $\mathcal{G} \models \mathcal{H}$, a proof shall start with $\Pi = \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}$, and proceed by repeatedly applying the rules in Figure 7 to it. The conditions in which a proof can be completed are stated in the following theorem.

THEOREM 1 (Soundness of Unfolding). *A proof obligation $\mathcal{G} \models \mathcal{H}$ holds if, starting with the proof obligation $\emptyset \vdash \mathcal{G} \models \mathcal{H}$, there exists a sequence of applications of proof rules that results in proof obligations $\tilde{\mathcal{A}} \vdash \mathcal{G}' \models \mathcal{H}'$ such that (a) $\mathcal{H}'$ contains only constraints, and (b) $\mathcal{G}' \models \mathcal{H}'$ can be discharged by the constraint solver.* □

We have now presented proof rules to explain how to reduce a proof obligation into others, and how, eventually, a constraint proof may be attempted. We next describe a strategy so as to make the application of the rules automated.

4.3 A Systematic Strategy for Unfolding

First, we remark that we do not make use of the cut (CUT) and split rules (SPL) rules.

This subsection simply describes a systematic interleaving of the left-unfold and right-unfold rules, attempting a constraint proof along the way. We present our algorithm in pseudocode in Figure 8. Note that the presentation is in the form of a nondeterministic algorithm, and thus each of the nondeterministic operator **choose** needs to be implemented by some form of systematic search.

By a *constraint proof* of a obligation, we mean to repeatedly apply the CP rule in order to remove all occurrences of assertion

REDUCE($\mathcal{G} \models \mathcal{H}$) **returns boolean**

- Constraint Proof:
  Apply a constraint proof to $\mathcal{G} \models \mathcal{H}$.
  If successful, **return** *true*, otherwise **return** *false*
- memoize ($\mathcal{G} \models \mathcal{H}$) as an assumption
- Coinduction :
  choose to attempt coinduction or not
  case: yes
  Check if the coinduction rule CO applies, that is, there is an assumption $\mathcal{G}' \models \mathcal{H}'$ such that $\mathcal{G} \models \mathcal{G}'\theta$ has a constraint proof.
  If so **return** REDUCE($\mathcal{H}'\theta \models \mathcal{H}$);
  otherwise **continue**
  case: no
  continue
- Unfold:
  choose left or right
  case: left
  choose an atom A in $\mathcal{G}$ to reduce
  for all reducts $\mathcal{G}_L$ of $\mathcal{G}$ using A:
  if REDUCE($\mathcal{G}_L \models \mathcal{H}$) = *false* **return** *false*
  return *true*
  case: right
  choose an atom A in $\mathcal{H}$ to reduce, obtaining $\mathcal{G}_R$
  return REDUCE($\mathcal{G} \models \mathcal{G}_R$)

Figure 8: Systematic Reduction into Constraints

predicates in the obligation, in an obvious way. Then the constraint solver is applied to the resulting obligation.

5. Proof Method for Constraints

We now present an algorithm which, given a constraint obligation $\Psi_L \models \Psi_R$ where Ψ_L and Ψ_R are constraints, reduces this to one or more obligations $\Psi'_L \models \Psi'_R$ which contains only integer constraints such that $\Psi_L \models \Psi_R$ iff all of the obligations $\Psi'_L \models \Psi'_R$ hold.

5.1 Remove Existential Variables

First consider *existential* variables y that appear in Ψ_R but not in Ψ_L. Since we started off with a proof obligation $\mathcal{G} \models \mathcal{H}$ where $var(\mathcal{H}) \subset var(\mathcal{G})$, any existential variable in Ψ_R must have emerged from a right unfold. Recall that we have assumed that any local variable in a CLP rule appears as an argument to a body atom. This means that the existential variable y must have appeared as an argument in a predicate in the rhs of a proof obligation. This in turn means that after using the CP rule to eliminate predicates by equating predicate arguments in the lhs with the corresponding arguments on the rhs, there will be an equation of the form $y = e$ where the expression e contains no existential variable. Finally, by renaming all occurrences of such existential variables y by their counterparts e, we may assume hereafter that there are no existential variables in our constraint obligation $\Psi_L \models \Psi_R$.

5.2 Partition Orderings

Consider the set of integer expressions i in both Ψ_L and Ψ_R that appear as an index in

- a composite array expression, ie of the form $\langle a, i, e\rangle$, or
- an array element, ie. of the form $a[i]$, or
- a multiset expression of the form $a\{i..j\}$ where i and j are integer variables.

where a is an array expression. Call these expressions the *array indices*. Without losing generality, we assume that array indices are integer variables.

Define that a *partition ordering* Π is an ordering of the array indices using the relations $\{=, <\}$ which is consistent with the hypothesis constraint Ψ_L. That is, given a partition ordering Π, it is the case that for each pair of array indices i and j, exactly one of $i = j, i < j, j < i$ holds[2].

Clearly there are in general an exponential number of partition orderings. However, as we shall see later for several examples, often it is the case that the number is manageable. The main reason for this is that the array indices in a given program are typically already constrained in a partial order which is nearly a linear order.

In what follows, our constraint obligation is now

$$\Psi_L \wedge \Pi \models \Psi_R$$

where Π is a partition ordering for Ψ_L and Ψ_R.

5.3 Flatten Array Expressions

The purpose of this subsection is to reduce composite array expressions into array variables.

First consider multiset expressions of the form $\langle a, k, e\rangle\{i..j\}$ involving a composite array expression $\langle a, i, e\rangle$. If Π implies $k < i$ or $j < k$, then replace the expression by $a\{i..j\}$. Otherwise, replace the expression by:

- $\{e\} \cup a\{i+1..j\}$ if $\Pi \models i = k$;
- $a\{i..j-1\} \cup \{e\}$ if $\Pi \models j = k$;
- $a\{i..k-1\} \cup \{e\} \cup a\{k+1..j\}$ if $\Pi \models i < k < j$

Repeatedly applying this step results in all multiset expressions being of the form $a\{i..j\}$ where a is an array variable.

Next consider array equations which involve at least one composite array expression, that is, equations of the form $a' = \langle a, i, e\rangle$. We now introduce, temporarily, a new equation called a *bounded array equation* and it is of the form

$$a' =_{[i..j]} a$$

and it means that array equality applies only within the bounds i to j, that is, it means that $a'[k] = a[k]$ for all $i \leq k \leq j$. We shall allow the special bounds $-\infty$ and $+\infty$ so that initially all array equations $a' = a$ can be written as $a' =_{[-\infty..+\infty]} a$.

We now replace each bounded array equation which involves a composite array expression, say $a' =_{[i..j]} \langle a, k, e\rangle$, as follows. If Π implies $k < i$ or $j < k$, then replace the equation by $a' =_{[i..j]} a$. Otherwise, replace the equation by:

- $a' =_{[i+1..j]} a$ if $\Pi \models i = k$;
- $a' =_{[i..j-1]} a$ if $\Pi \models j = k$;
- $a' =_{[i..k-1]} a$ and $a' =_{[k+1..j} a$ if $\Pi \models i < k < j$

and finally, add the constraint $a'[k] = e$ (to the collection of integer constraints).

Repeatedly applying this step this results in all array equations being of the form $a' =_{[i..j]} a$ where both a' and a are array variables.

Finally consider array elements (which appear in integer constraints). Replace any occurrence $\langle a, i, e\rangle[j]$ involving a composite array expression by the simpler expression e in case the partition ordering Π implies $i = j$; otherwise, replace by $a[j]$. Repeatedly applying this step this results in all array elements being of the form $a[j]$ where a is an array variable.

At this stage, our proof obligation is of the form $\Psi_L \wedge \Pi \models \Psi_R$ where Ψ_L and Ψ_L contains bounded array equations, multiset constraints and integer constraints. Further, there are no composite array expressions.

[2] This definition is in fact stronger than needed. But its relaxation is complicated and hence omitted.

5.4 Partition Array Equations and Multiset Expressions

The purpose of this subsection is to ensure that each array equation $a' =_{[i..j]} a$ and multiset expression $a\{i..j\}$ is such that the interval $[i..j]$ is "basic" in the partition ordering Π, that is, there is no array index k such that $i \le k \le j$.

First consider any array equation of the form $a' =_{[i..j]} a$ and suppose there is an array index k such that $i \le k \le j$. Then, replace this equation by

- $a' =_{[i+1..j]} a$ if $\Pi \models i = k$ $(3a)$
- $a' =_{[i..j-1]} a$ if $\Pi \models j = k$ $(3b)$
- $a' =_{[i..k-1]} a$ and $a' =_{[k+1..j]} a$ if $\Pi \models i < k < j$ $(3c)$

and add the constraint $a'[k] = a[k]$ (to the collection of integer constraints).

Next consider any multiset expression $a\{i..j\}$. If there is an array index k such that $i \le k \le j$, then replace the multiset expression by

- $\{a[k]\} \cup a\{i+1..j\}$ if $\Pi \models i = k$ $(4a)$
- $a\{i..j-1\} \cup \{a[k]\}$ if $\Pi \models j = k$; $(4b)$
- $a\{i..k-1\} \cup \{a[k]\} \cup a\{k+1..j\}$ if $\Pi \models i < k < j$ $(4c)$

Repeatedly perform the above steps would lead to all array equations and multiset expressions involving only basic intervals.

Finally, for each array equation or multiset expression created in steps $(3a)$ through $(4c)$ whose size s is such that neither $\Pi \models s = 1$ nor $\Pi \models s > 1$ holds, we need to perform a partition ordering *refinement*. For example, if it were step $(3a)$ where the array equation $a' =_{[i+1..j]}$ was created, refine the partition ordering Π into the two cases $\Pi_1 \stackrel{\text{def}}{=} \Pi \wedge i+1 = j$ and $\Pi_2 \stackrel{\text{def}}{=} \Pi \wedge i+1 < j$.

Similarly, in the case step $(4c)$ where the two multiset expressions $a\{i..k-1\}$ and $a\{k+1..j\}$ were created, refine the partition ordering Π into the four cases $\Pi_1 \stackrel{\text{def}}{=} \Pi \wedge i = k-1$, $\Pi_2 \stackrel{\text{def}}{=} \Pi \wedge i < k-1$, $\Pi_3 \stackrel{\text{def}}{=} \Pi \wedge k+1 = j$, $\Pi_4 \stackrel{\text{def}}{=} \Pi \wedge k+1 < j$.

We now repeat this procedure of partitioning refinement, and it in fact terminates. When it does, we have in general several proof obligations and in each, each array equation and multiset expression refers to a "basic interval" in the sense that each basic interval refers to joint regions of the arrays they are associated to.

5.5 Prove Array Equations

We now have the proof obligation $\Psi_L \wedge \Pi \models \Psi_R$ where all array expressions are simply array variables, all array equations $a' =_{[i..j]} a$ and multiset expressions $a\{i..j\}$ involve only basic intervals $[i..j]$.

It is now easy to dispense with array equations. First remove all occurrences of trivial array equations $a =_{[i..j]} a$. Then each array equation in Ψ_R must appear in Ψ_L or else the proof obligation is *false*.

Next, for each array equation $a' =_{[i..j]} a$ where $i \neq -\infty$ and $j \neq +\infty$, add the multiset constraint $a'\{i..j\} = a\{i..j\}$. Finally, *discard all* array equations.

Having dispensed with proving array equations, we are now left with proving the formula $\Psi_L \wedge \Pi \models \Psi_R$ where Ψ_L and Ψ_L contains only multiset and integer constraints. Further, there are no composite array expressions in array element and multiset expressions.

5.6 Arithmetize Array Elements

Without losing generality, assume that all remaining array elements are of the form $a[i]$ where a is an array variable and i an integer variable. Replace each array element by introducing a fresh integer variable. Then for every pair of array elements $a[i]$ and $a[j]$ where $\Pi \models i = j$, equate the corresponding introduced variables.

At this point, the remaining proof obligation contains only multiset and integer expressions.

5.7 Arithmetize Multiset Constraints

For simplicity, and without losing generality, we assume each multiset term is either a multiset variable or of the form $\{e\}$, where e is an integer variable, Recall that each multiset constraint is of the form $\mathcal{M} = \{e\}$, $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$ or $\mathcal{M}_1 \otimes \mathcal{M}_2 \otimes \cdots \otimes \mathcal{M}_n$ where $\mathcal{M}_i$, $1 \le i \le n$ are multiset variables.

Where $\{e\}$ is a term that appears in a multiset constraint, call the integer variable e an *element-term*. Let $\widetilde{\mathcal{E}} = \{e_1, \cdots, e_k\}$ denote the set of all such terms[3]. Let $\widetilde{\mathcal{M}}$ denote the set of all multiset variables appearing in the multiset constraints.

- Create new integer variables of the form $\#(e, \mathcal{M})$ where e ranges over $\widetilde{\mathcal{E}}$ and $\mathcal{M}$ over the multiset variables $\widetilde{\mathcal{M}}$. Then add to Ψ_L the constraints

  $$\#(e, \mathcal{M}) \ge 0, \text{ for all } e \in \widetilde{\mathcal{E}} \text{ and } \mathcal{M} \in \widetilde{\mathcal{M}}.$$

  If $\mathcal{M}$ were a variable created in Section 5.3 (to represent an array segment $a\{i..j\}$, then also add the constraint

  $$\#(e_1, \mathcal{M}) + \cdots + \#(e_k, \mathcal{M}) = j - i + 1$$

- Replace each multiset constraint $\mathcal{M} = \{e\}$ by the new integer constraints:

  $$\#(e, \mathcal{M}) = 1$$
  $$\#(e', \mathcal{M}) = 0, \text{ for all } e' \in \widetilde{\mathcal{E}} - e$$

- Replace each multiset constraint $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$, by the new integer constraints:

  $$\#(e, \mathcal{M}) = \#(e, \mathcal{M}_1) + \#(e, \mathcal{M}_2), \text{ for all } e \in \widetilde{\mathcal{E}}.$$

- Replace each multiset constraint $\mathcal{M}_1 \otimes \mathcal{M}_2 \otimes \cdots \otimes \mathcal{M}_n$ by the new integer constraints:

  $$\left.\begin{array}{l} \#(e, \mathcal{M}_1) > 0 \rightarrow \#(e, \mathcal{M}_2) = \cdots = \#(e, \mathcal{M}_n) = 0 \\ \cdots \\ \#(e, \mathcal{M}_n) > 0 \rightarrow \#(e, \mathcal{M}_1) = \cdots = \#(e, \mathcal{M}_{n-1}) = 0 \end{array}\right\} \text{ for all } e \in \widetilde{\mathcal{E}}$$

Note that the arithmetization of the disjoint constraint $\otimes$ resulted in a number of implications, which are essentially disjunctions. In practice, however, the number of such disjunctions is small because it is often the case the appearance or non-appearance of a certain element e within a multiset $\mathcal{M}$ is known. We shall see this in our examples below.

The constraint obligation we obtain at end of this process now contains only integer constraints. We are thus at the end of the process now, delegating the remainder of the proof obligation to an integer solver. We do not pursue the matter further in this paper.

THEOREM 2 (Soundness and Completeness of Arithmetization). *Let $\Psi_L \models \Psi_R$ denote a constraint obligation. Let this be transformed by the process described above into several constraint obligations $\Psi'_L \models \Psi'_R$ which contain only integer constraints. Then, $\Psi_L \models \Psi_R$ iff all of the integer proof obligations $\Psi'_L \wedge \Pi \models \Psi'_R$ hold.* □

6. Examples of Proofs

6.1 Fibonacci Number Generator

Consider Figure 1. In the proof of the program, we use the following loop invariant $\Psi \equiv fib(I-1, X), fib(I-2, Y), I \ge 2, I \le L+1$. We confirm that the condition is a genuine invariant by proving the following:

$$\Psi, I \le L, X' = X + Y, Y' = X, I' = I + 1 \models \Psi[I \mapsto I', Y \mapsto Y', X \mapsto X']. \quad (F.1)$$

[3] If there are in fact no element terms, we shall just invent one.

In $F.1$ the lhs goal represents the invariant Ψ and the execution of the loop body. $F.1$ can be simplified into

$$\begin{array}{l} \mathit{fib}(I-1,X),\mathit{fib}(I-2,Y),I\geq 2,I\leq L \\ \quad \models \mathit{fib}(I,X+Y),\mathit{fib}(I-1,X),I\geq 1,I\leq L. \end{array} \quad (F.2)$$

By unfolding the atom $\mathit{fib}(I,X+Y)$ in the rhs of the implication we obtain another rhs goal

$$\begin{array}{c} \mathit{fib}(I-1,?X_1),\mathit{fib}(I-2,?Y_1),\mathit{fib}(I-1,X), \\ X+Y=?X_1+?Y_1,I\geq 1,I\leq L. \end{array}$$

Here we instantiate the existentially quantified variables X_1 and Y_1 as X and Y respectively. The predicates of the lhs and rhs are now identical and can be removed.

Now we are left with a constraint proof obligation of $I\geq 2,I\leq L\models I\geq 1,I\leq L$. Clearly this is trivially *true*.

6.2 Bubble Sort

The program has two loops. Here we shall just prove that the inner loop satisfies a particular input-output relation. The following proof obligation states the correctness of the symbolic execution exiting the inner loop:

$$\begin{array}{l} I_f=I,N_f=N,0\leq I_f<N_f-1, \\ \mathit{max}(H_f,N_f-I_f-2,H_f[N_f-I_f-1]), \\ \mathit{sorted}(H_f,N_f-I_f-1,N_f-1)\models \\ \quad I_f=I,N_f=N,\mathit{sorted}(H_f,N_f-I_f-2,N_f-1). \end{array} \quad (S.1)$$

We now perform one left unfold on *max*, and one right unfold on *sorted* so that we obtain

$$\begin{array}{l} I_f=I,N_f=N,0\leq I_f<N_f-1, \\ 0\leq N_f-I_f-2,H_f[N_f-I_f-2]\leq H_f[N_f-I_f-1], \\ \mathit{max}(H_f,N_f-I_f-3,H_f[N_f-I_f-1]), \\ \mathit{sorted}(H_f,N_f-I_f-1,N_f-1)\models \\ \quad I_f=I,N_f=N,N_f-I_f-2<N_f-1, \\ \quad H_f[N_f-I_f-2]\leq H_f[N_f-I_f-1], \\ \quad \mathit{sorted}(H_f,N_f-I_f-1,N_f-1) \end{array} \quad (S.2)$$

Next we replace both array references $H_f[N_f-I_f-2]$ and $H_f[N_f-I_f-1]$ with simple integer variables.

Now the proof obligation contains only integer constraints, and its validity is easy to verify.

We next prove (2), that is, the inner loop results in a permutation of the original array segment. This problem is reducible to proving a number of obligations, one of which is the following:

$$\begin{array}{l} 0\leq I<N-1,0\leq J<N-1-I,H[J+1]<H[J],I_f=I,N_f=N, \\ \langle\langle H,J,H[J+1]\rangle,J+1,H[J]\rangle\{0\ldots N-1\}= \\ \quad H_f\{0\ldots N-1\}\models \\ I_f=I,N_f=N,0<=I<N-1,H\{0\ldots N-1\}=H_f\{0\ldots N-1\} \end{array} \quad (P.1)$$

Now consider multiset elements. They are: $0, J$, $J+1$, and $N-1$. We shall consider one partition ordering $0<J<J+1<N-1$. (There are in fact a total of 4 orderings.) We next perform multiset partitioning which transforms $(P.1)$ into:

$$\begin{array}{l} 0\leq I<N-1,0\leq J<N-1-I,H[J+1]<H[J], \\ I_f=I,N_f=N,0<J,J<J+1,J+1<N-1, \\ \langle\langle H,J,H[J+1]\rangle,J+1,H[J]\rangle\{0\ldots J-1\}\cup \\ \langle\langle H,J,H[J+1]\rangle,J+1,H[J]\rangle\{J\}\cup \\ \langle\langle H,J,H[J+1]\rangle,J+1,H[J]\rangle\{J+1\}\cup \\ \langle\langle H,J,H[J+1]\rangle,J+1,H[J]\rangle\{J+2\ldots N-1\} \\ \quad =H_f\{0\ldots J-1\}\cup H_f\{J\}\cup H_f\{J+1\}\cup H_f\{J+2\ldots N-1\} \\ \models I_f=I,N_f=N,0\leq I<N-1, \\ \quad H\{0\ldots J-1\}\cup H\{J\}\cup H\{J+1\}\cup H\{J+2\ldots N-1\} \\ \quad =H_f\{0\ldots J-1\}\cup H_f\{J\}\cup H_f\{J+1\}\cup \\ \quad\quad H_f\{J+2\ldots N-1\} \end{array} \quad (P.2)$$

We now apply flattening to the multisets, for example, we can reduce $\langle\langle H,J,H[J+1]\rangle,J+1,H[J]\rangle\{0\ldots J-1\}$ into $\langle H,J,H[J+1]\rangle\{0\ldots J-1\}$ since $J-1<J+1$, and this can be further rewritten into $H\{0\ldots J-1\}$ since $J-1<J$. Repeatedly doing this, we rewrite $(P.2)$ into:

$$\begin{array}{l} 0\leq I<N-1,0\leq J<N-1-I,H[J+1]<H[J], \\ I_f=I,N_f=N,0<J,J<J+1,J+1<N-1, \\ H\{0\ldots J-1\}\cup H\{J\}\cup H\{J+1\}\cup H\{J+2\ldots N-1\} \\ \quad =H_f\{0\ldots J-1\}\cup H_f\{J\}\cup H_f\{J+1\}\cup H_f\{J+2\ldots N-1\} \\ \models I_f=I,N_f=N,0\leq I<N-1, \\ \quad H\{0\ldots J-1\}\cup H\{J\}\cup H\{J+1\}\cup H\{J+2\ldots N-1\} \\ \quad =H_f\{0\ldots J-1\}\cup H_f\{J\}\cup H_f\{J+1\}\cup H_f\{J+2\ldots N-1\} \end{array} \quad (P.3)$$

Next we write $H\{0\ldots J-1\}$ as M_1, $H\{J\}$ as M_2, $H\{J+1\}$ as M_3, $H\{J+2\ldots N-1\}$ as M_4, $H_f\{0\ldots J-1\}$ as N_1, $H_f\{J\}$ as N_2, $H_f\{J+1\}$ as N_3, and $H_f\{J+2\ldots N-1\}$ as N_4. We also rewrite $H[J]$ and $H[J+1]$ into X and Y respectively using separate fresh integer variables. This is because $J\neq J+1$.

The next step is to convert $P.3$ into an integer obligation. In order to do this, we need to define a set of element terms. Here we are unable to do so since there is no constraint of the form $e=M$, for some multiset M, hence we invent an element term Z. We then convert $P.3$ into the following obligation in the way prescribed in Section 5.7.

$$\begin{array}{l} 0\leq I<N-1,0\leq J<N-1-I,Y<X, \\ I_f=I,N_f=N,0<J,J<J+1,J+1<N-1, \\ \#(Z,M_1)\geq 0,\#(Z,M_2)\geq 0,\#(Z,M_3)\geq 0,\#(Z,M_4)\geq 0, \\ \#(Z,N_1)\geq 0,\#(Z,N_2)\geq 0,\#(Z,N_3)\geq 0,\#(Z,N_4)\geq 0, \\ \#(Z,M_1)=J,\#(Z,M_2)=1,\#(Z,M_3)=1,\#(Z,M_4)=N-J-2, \\ \#(Z,N_1)=J,\#(Z,N_2)=1,\#(Z,N_3)=1,\#(Z,N_4)=N-J-2, \\ \#(Z,M_1)+\#(Z,M_2)+\#(Z,M_3)+\#(Z,M_4)= \\ \quad \#(Z,N_1)+\#(Z,N_2)+\#(Z,N_3)+\#(Z,N_4) \\ \models I_f=I,N_f=N,0\leq I<N-1, \\ \#(Z,M_1)\geq 0,\#(Z,M_2)\geq 0,\#(Z,M_3)\geq 0,\#(Z,M_4)\geq 0, \\ \#(Z,N_1)\geq 0,\#(Z,N_2)\geq 0,\#(Z,N_3)\geq 0,\#(Z,N_4)\geq 0, \\ \#(Z,M_1)=J,\#(Z,M_2)=1,\#(Z,M_3)=1,\#(Z,M_4)=N-J-2, \\ \#(Z,N_1)=J,\#(Z,N_2)=1,\#(Z,N_3)=1,\#(Z,N_4)=N-J-2, \\ \#(Z,M_1)+\#(Z,M_2)+\#(Z,M_3)+\#(Z,M_4)= \\ \quad \#(Z,N_1)+\#(Z,N_2)+\#(Z,N_3)+\#(Z,N_4) \end{array} \quad (P.4)$$

It can be easily checked that this integer obligation is valid.

6.3 List Reset

We now consider the list reset example in Section 3.3. This proof shows that the program recursion which is "sublist recursive" need not be the same as the recursion in the assertion predicate, which is tail recursive.

Among the obligations that need to be proven, we have to establish that $\Psi\equiv \mathit{allz}(h_0,\mathsf{h},p_0,\mathsf{p})$ is a loop invariant. This obligation is expressed as

$$\begin{array}{l} \mathit{allz}(H_0,H,P_0,P),H[P+1]>0\models \\ \quad \mathit{allz}(H_0,\langle H,H[P+1],0\rangle,P_0,H[P+1]). \end{array} \quad (Z.1)$$

Here we apply left unfold (LU+C rule) to the only lhs atom using the two rules of *allz* obtaining two new obligations. We shall just display one of them:

$$\begin{array}{l} \mathit{allz}(H_0,H_1,H_0[P_0+1],P),P_0>0,H_1[P+1]>0\models \\ \quad \mathit{allz}(H_0,\langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle,P_0,H_1[P+1]). \end{array} \quad (Z.2)$$

Now we apply coinduction (rule (CO)) using $Z.1$ as the hypothesis. According to the CO rule, we require two conditions. The first is

$$\begin{array}{l} \mathit{allz}(H_0,H_1,H_0[P_0+1],P),P_0>0,H_1[P+1]>0\models \\ \quad \mathit{allz}(H_0,H_1,H_0[P_0+1],P),H_1[P+1]>0 \end{array}$$

This is established by eliminating the predicates, as follows:

$$\begin{array}{l} P_0>0,H_1[P+1]>0\models \\ \quad H_0=H_0,H_1=H_1,H_0[P_0+1]=H_0[P_0+1], \\ \quad P=P,H_1[P+1]>0. \end{array}$$

We do not detail this proof further; it is in fact trivial. The second condition is

$$\begin{array}{l} \mathit{allz}(H_0,\langle H_1,H_1[P+1],0\rangle,H_0[P_0+1],H_1[P+1])\models \\ \quad \mathit{allz}(H_0,\langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle,P_0,H_1[P+1]). \end{array} \quad (Z.3)$$

Here we perform a right unfold using the 2nd rule of *allz* resulting in

$$\begin{array}{l} \mathit{allz}(H_0,\langle H_1,H_1[P+1],0\rangle,H_0[P_0+1],H_1[P+1])\models \\ \quad \mathit{allz}(H_0,?H_2,H_0[P_0+1],H_1[P+1]), \\ \quad \langle\langle H_1,P_0,0\rangle,H_1[P+1],0\rangle=\langle ?H_2,P_0,0\rangle \end{array} \quad (Z.4)$$

Here, H_2 is existentially quantified. We substitute it by the expression $\langle H_1, H_1[P+1], 0\rangle$, because both appear in the same position in the argument of *allz* in the hypothesis and conclusion of the obligation. By an application of CP proof rule, we also equate the rest of the arguments of *allz*, and remove the predicates resulting in the obligation:

$$\begin{array}{l} \mathit{true} \models H_0 = H_0, H_0[P_0+1] = H_0[P_0+1], \\ \quad H_1[P+1] = H_1[P+1], \\ \quad \langle\langle H_1, P_0, 0\rangle, H_1[P+1], 0\rangle = \langle\langle H_1, H_1[P+1], 0\rangle, P_0, 0\rangle. \end{array} \quad (Z.5)$$

We now consider one partition ordering of array indices $P_0 = P < P_0+1 = P+1 < H_1[P+1]$. Note that this full consideration of all the array indices is not really needed. We can in fact ignore P_0+1, P, and $P+1$, because they are not used in an array expression. Hence it is sufficient to consider the simpler ordering $P_0 < H_1[P+1]$. Using this ordering, we now can flatten the array expressions used in array equations obtaining

$$\begin{array}{l} P_0 < H_1[P+1], P_0+1 < H_1[P+1] \models \\ \quad H_0 = H_0, H_0[P_0+1] = H_0[P_0+1], \\ \quad H_1[P+1] = H_1[P+1], H_1 =_{[-\infty..P_0-1]} H_1, \\ \quad H_1 =_{[P_0+1..H_1[P+1]-1]} H_1, H_1 =_{[H_1[P+1]+1..+\infty]} H_1, \\ \quad \langle\langle H_1, P_0, 0\rangle, H_1[P+1], 0\rangle[H_1[P+1]] = 0, \\ \quad \langle\langle H_1, P_0, 0\rangle, H_1[P+1], 0\rangle[P_0] = 0 \end{array} \quad (Z.6)$$

We continue flattening array expressions, this time, for those that appear in integer constraints:

$$\begin{array}{l} P_0 < H_1[P+1], P_0+1 < H_1[P+1] \models \\ \quad H_0 = H_0, H_0[P_0+1] = H_0[P_0+1], \\ \quad H_1[P+1] = H_1[P+1], H_1 =_{[-\infty..P_0-1]} H_1, \\ \quad H_1 =_{[P_0+1..H_1[P+1]-1]} H_1, H_1 =_{[H_1[P+1]+1..+\infty]} H_1, \\ \quad 0 = 0, 0 = 0 \end{array} \quad (Z.7)$$

As dictated in Section 5.5, here we need to introduce a fresh variable for each array equality and test for unsatisfiability, also that we need to replace array elements with fresh variables. However, in each array equality in the conclusion of $Z.7$, both sides are the same, and so it is with array element equalities. Hence we can immediately conclude that the obligation holds.

So far we have used the tail recursive definition of *allz* to complete the proof. We could alternatively use the sublist recursive definition (Figure 3), which demonstrates the robustness of our proof method to handle different definitions. Again, we start with the obligation $Z.1$. We now perform right unfold on $Z.1$ using the 2nd rule of the sublist recursive definition:

$$\begin{array}{l} \mathit{allz}(H_0, H, P_0, P), H[P+1] > 0 \models \\ \quad \mathit{allz}(H_0, H, P_0, T), H[P+1] > 0, H[P+1] = H[T+1]. \end{array} \quad (Z.8)$$

We now substitute the existentially-quantified T with P, and we remove the predicates, obtaining an obligation that holds trivially.

$$\begin{array}{l} H[P+1] > 0 \quad \models H_0 = H_0, H = H, \\ \quad P_0 = P_0, H[P+1] > 0, H[P+1] = H[P+1] \end{array} \quad (Z.9)$$

6.4 List Reverse

.

Consider the list reverse example in Figure 4. Note that the definition of *reverse*, corresponds to an "in-situ" property of the reverse function. In particular, the memory region occupied by the list is unchanged. Moreover, the definition also implies that whenever one node points to another in the input, the latter node points to the former in the output.

Similar to the list reset example, here we also need to prove a loop invariant, which in this case is $\Psi \equiv \exists \mathtt{t}, \mathtt{u}.\mathit{reverse}(h_0, \mathtt{h}, i_0, \mathtt{i}, \mathtt{j})$, $\mathit{alist}(\mathtt{h}, \mathtt{j}, \mathtt{t})$, $\mathit{alist}(\mathtt{h}, \mathtt{i}, \mathtt{u})$. The proof amounts to establishing the following obligation:

$$\begin{array}{l} \mathit{reverse}(H_0, H, I_0, I, J), \mathit{alist}(H, J, T), \mathit{alist}(H, I, U), \\ \quad T \otimes U, S = T \cup U, I > 0 \models \\ \quad\quad \mathit{reverse}(H_0, \langle H, I+1, J\rangle, I_0, H[I+1], I), \\ \quad\quad \mathit{alist}(\langle H, I+1, J\rangle, I, ?T'), \\ \quad\quad \mathit{alist}(\langle H, I+1, J\rangle, H[I+1], ?U'), \\ \quad\quad ?T' \otimes ?U', ?T' \cup ?U' = S \end{array} \quad (R.1)$$

We now perform three unfolds simultaneously, for brevity. We unfold the atom $\mathit{alist}(H, I, U)$ in the lhs and both the atoms $\mathit{reverse}(H_0, \langle H, I+1, J\rangle, I_0, H[I+1], I)$ and $\mathit{alist}(\langle H, I+1, J\rangle, I, ?T')$ in the rhs obtaining the obligation

$$\begin{array}{l} \mathit{reverse}(H_0, H, I_0, I, J), \mathit{alist}(H, J, T), \mathit{alist}(H, H[I+1], U_1), \\ \quad I > 0, U = U_1 \cup \{I, I+1\}, \{I, I+1\} \otimes U_1, \\ \quad T \otimes U, S = T \cup U, I > 0 \models \\ \quad\quad \mathit{reverse}(H_0, H, I_0, I, J), \\ \quad\quad \mathit{alist}(\langle H, I+1, J\rangle, \langle H, I+1, J\rangle[I+1], ?T_1), \\ \quad\quad \mathit{alist}(\langle H, I+1, J\rangle, H[I+1], ?U'), \\ \quad\quad I > 0, I \in ?T, ?T = ?T_1 \cup \{I, I+1\}, \{I, I+1\} \otimes ?T_1, \\ \quad\quad ?T' \otimes ?U', ?T' \cup ?U' = S \end{array} \quad (R.2)$$

We now proceed to remove predicates so reducing the problem to a constraint proof.

At this point, we will temporarily assume a useful lemma:

$$\mathit{alist}(H, J, S), \{I\} \otimes S \models \mathit{alist}(\langle H, I, E\rangle, J, S),$$

Using this, we replace any predicate in the rhs of $R.2$ matching the above rhs predicate, with the corresponding lhs. This process results in the following:

$$\begin{array}{l} \mathit{reverse}(H_0, H, I_0, I, J), \mathit{alist}(H, J, T), \mathit{alist}(H, H[I+1], U_1), \\ \quad I > 0, U = U_1 \cup \{I, I+1\}, \{I, I+1\} \otimes U_1, \\ \quad T \otimes U, S = T \cup U, I > 0 \models \\ \quad\quad \mathit{reverse}(H_0, H, I_0, I, J), \\ \quad\quad \mathit{alist}(H, \langle H, I+1, J\rangle[I+1], ?T_1), \{I+1\} \otimes ?T_1, \\ \quad\quad \mathit{alist}(H, H[I+1], ?U'), \{I+1\} \otimes ?U', \\ \quad\quad I > 0, I \in ?T, ?T = ?T_1 \cup \{I, I+1\}, \{I, I+1\} \otimes ?T_1, \\ \quad\quad ?T' \otimes ?U', ?T' \cup ?U' = S \end{array} \quad (R.3)$$

We now apply the CP proof rule to remove the predicates. We first note that we can substitute the existentially quantified variables T_1 and U' with T and U_1, respectively, resulting in the following (for clarity, we immediately remove trivial equivalences such as $I = I$):

$$\begin{array}{l} I > 0, U = U_1 \cup \{I, I+1\}, \{I, I+1\} \otimes U_1, \\ \quad T \otimes U, S = T \cup U, I > 0 \models \\ \quad\quad J = \langle H, I+1, J\rangle[I+1], \{I+1\} \otimes T, \{I+1\} \otimes U_1, \\ \quad\quad I > 0, I \in T \cup \{I, I+1\}, \{I, I+1\} \otimes T, \\ \quad\quad (T \cup \{I, I+1\}) \otimes U_1, T \cup \{I, I+1\} \cup U_1 = S \end{array} \quad (R.4)$$

We now discuss how we prove the array constraint $(T \cup \{I, I+1\}) \otimes U_1$ of the rhs. Intuitively, this constraint is implied by the constraints $U = U_1 \cup \{I, I+1\}$, $\{I, I+1\} \otimes U_1$, and $T \otimes U$ on the lhs. We will omit proving the other constraints on the rhs.

We next convert $(T \cup \{I, I+1\}) \otimes U_1$ into constraints on integers as prescribed in Section 5.7, resulting in the following obligation. Note that we equate a new variable I_1 to $I+1$ so that we can use I_1 in counting variables.

$$\begin{array}{l} \#(I, U) = \#(I, U_1) + \#(I, \{I, I_1\}), \\ \#(I, U_1) + \#(I, \{I, I_1\}) \leq 1, \\ \#(I, T) + \#(I, U) \leq 1, \\ \#(I_1, U) = \#(I_1, U_1) + \#(I_1, \{I, I_1\}), \\ \#(I_1, U_1) + \#(I_1, \{I, I_1\}) \leq 1, \\ \#(I_1, T) + \#(I_1, U) \leq 1, \\ \#(I, \{I, I+1\}) = \#(I, \{I\}) + \#(I, \{I_1\}), \\ \#(I_1, \{I, I+1\}) = \#(I_1, \{I\}) + \#(I_1, \{I_1\}), \\ \#(I, \{I\}) = 1, \#(I, \{I_1\}) = 0, \#(I_1, \{I\}) = 0, \#(I_1, \{I_1\}) = 1 \models \\ \quad \#(I, T) + \#(I, \{I, I_1\}) + \#(I, U_1) \leq 1, \\ \quad \#(I_1, T) + \#(I_1, \{I, I_1\}) + \#(I_1, U_1) \leq 1 \end{array}$$

Let us now prove the rhs constraint $\#(I, T) + \#(I, \{I, I_1\}) + \#(I, U_1) \leq 1$ by testing the unsatisfiability of the conjunction of its negation with lhs constraints. Notice that we have in the lhs $\#(I, U) = \#(I, U_1) + \#(I, \{I, I_1\})$, and $\#(I, T) + \#(I, U) \leq 1$. The conjunction of these with $\#(I, T) + \#(I, \{I, I_1\}) + \#(I, U_1) > 1$ is clearly unsatisfiable.

6.5 AVL Tree

Consider the program in Figure 5. The proof obligation associated with the program point $\langle 7\rangle$ is

$$
\begin{array}{l}
Y_f = H[X+1], H[Y_f] = 1, \\
H' = \langle H, X, 0\rangle, H'' = \langle H', Y_f, 0\rangle, H''' = \langle H'', Y_f+2, X\rangle, \\
H_f = \langle H''', X+1, H''[Y_f+2]\rangle, \\
avl(H, H[X+2], DL-2, S_2), \\
avl(H, H[H[X+1]+1], DL-1, S_{11}), \\
avl(H, H[H[X+1]+2], DL-2, S_{12}), \qquad (V.1) \\
S = \{X, X+1, X+2\} \cup \\
\quad \{H[X+1], H[X+1]+1, H[X+1]+2\} \cup S_2 \cup S_{11} \cup S_{12}, \\
\{X, X+1, X+2\} \otimes \{H[X+1], H[X+1]+1, H[X+1]+2\} \otimes \\
\quad S_2 \otimes S_{11} \otimes S_{12} \\
\models avl(H_f, Y_f, DL, S).
\end{array}
$$

Here we perform right unfold twice on the recursive definition of *avl*. After these unfolds, we perform predicate eliminations so that we are left with a constraint obligation. The remainder of the proof proceeds much like the list reverse example, and hence is omitted.

6.6 Statistics

The critical statistics for performance are firstly, (a) the size of the unfold search tree, (b) the number of partition orderings, and finally, (c) the number of multiset elements and multisets, which gives rise to counting variables of the form $\#(i, \mathcal{M})$. It is easy to see that for all the above examples, these numbers are trivially small.

First consider unfolds. In the Fibonacci example (obligation *F*.1), we needed only a single unfold. The bubble sort proof for sortedness (obligation *S*.1) used one left and one right unfolds. The proof of permutation of bubble sort (obligation *P*.1) used none. The list reset example (obligation *Z*.1) used two left unfolds, and three right unfolds. The list reverse example (obligation *R*.1) used one left and two right unfolds. Finally the AVL tree example (obligation *V*.1) used two right unfolds.

Next consider partition orderings. In the proof of $F.1, S.1$ and *R*.1 we only required a single partition ordering. In the proof of *P*.1, we have only four possible orderings. In the proof of *Z*.6, we needed five. In the proof of *V*.1 there are only two possible orderings.

Finally consider multiset counting variables. The proof of $F.1$, $Z.1$, and $S.1$ did not employ multiset constraints. For the proof of *P*.1 there was 8 multiset counting variables. For *R*.1 there was also 8, and finally for *V*.1, there were 18.

In general, we believe our algorithm scales mainly because the number of partition orderings is typically very constrained, and is largely independent the size of the program. The unfolding process is typically short because user-supplied predicates representing abstract properties of linked data structures, are typically simple. Predicates representing arithmetic properties can be however be very complex[4]. Finally, multiset elements are typically very few. This is because they represent the few distinguished cells in a data structure, such as the head of a list or root of a tree.

7. Conclusion

We presented a general purpose assertion language for expressing properties of data structures, and a CLP-based proof system for assertions of this language. We showed that the system is expressive because it describes both low-level specifications via constraints, and high-level specifications via CLP rules.

The main contribution was proof method which is based on unfolding CLP definitions of user-specified properties of data structures. We introduced a novel principle of coinduction which is used in conjunction with a set of unfold rules in order to efficiently dispense recursive definitions into constraints involving arrays, multisets and integers. We then provided a sound and complete method for reducing these constraints into integer constraints. Finally we demonstrated the practicality of our algorithm.

While we have used the integer domain as our destination, clearly we could have used another domain consistent with a different modeling of the underlying machine, for example, 32-bit numbers. In our experimental system, we make one more step: we transform the integer problem $\Psi_L{}' \wedge \Pi \models \Psi'$ into a satisfiability problem in the obvious way, and use a *real arithmetic* solver available in order to dispense the proof. Note that this step is sound, ie. a successful result is correct, but not complete, ie. the real solver may not succeed even if a proof exists.

[4] But here we would have few partition orderings.

References

[1] F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Deduction*, chapter 8. Springer, 2001.

[2] S. Craciunescu. Proving equivalence of CLP programs. In P. J. Stuckey, editor, *18th ICLP*, volume 2401 of *LNCS*. Springer, 2002.

[3] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *12th TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[4] L. Fribourg. Automatic generation of simplification lemmas for inductive proofs. In V. A. Saraswat and K. Ueda, editors, *ISLP 1991*, pages 103–116. MIT Press, 1991.

[5] R. Giacobazzi, editor. *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26–28, 2004, Proceedings*, volume 3148 of *LNCS*. Springer, 2004.

[6] J. Jaffar. Presburger arithmetic with array segments. *Information Processing Letters*, 12(2):79–82, 1981.

[7] J. Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.

[8] J. Jaffar and J.-L. Lassez. Reasoning about array segments. In *ECAI 1982*, pages 62–66, 1982.

[9] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[10] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In Giacobazzi [5], pages 246–264.

[11] T. Kanamori and H. Fujita. Formulation of induction formulas in verification of Prolog programs. In J. H. Siekmann, editor, *8th CADE*, volume 230 of *LNCS*, pages 281–299. Springer, 1986.

[12] T. Kanamori and H. Seki. Verification of Prolog programs using an extension of execution. In E. Y. Shapiro, editor, *3rd ICLP*, volume 225 of *LNCS*, pages 475–489. Springer, 1986.

[13] N. Klarlund and M. I. Schwartzbach. Graph types. In *20th POPL*, pages 196–205. ACM Press, 1993.

[14] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *IFIP Congress 1962*. North-Holland, 1983.

[15] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In K. Etessami and S. K. Rajamani, editors, *17th CAV*, volume 3576 of *LNCS*, pages 476–490. Springer, 2005.

[16] F. Mesnard, S. Hoarau, and A. Maillard. CLP($\mathcal{X}$) for automatically proving program properties. In F. Baader and K. U. Schulz, editors, *1st FroCoS*, volume 3 of *Applied Logic Series*. Kluwer Academic Publishers, 1996.

[17] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *15th PLDI*, pages 221–231. ACM Press, May 2001. SIGPLAN Notices 36(5).

[18] H. H. Nguyen, C. David, S. C. Qin, and W. N. Chin. Automated verification of shape and size properties via separation logic. In B. Cook and A. Podelski, editors, *8th VMCAI*, volume 4349 of *LNCS*. Springer, 2007.

[19] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *17th LICS*, pages 55–74. IEEE Computer Society Press, 2002.

[20] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd POPL*, pages 296–309. ACM Press, 2005.

[21] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In C. Hankin and I. Siveroni, editors, *12th SAS*, volume 3672 of *LNCS*, pages 284–302. Springer, 2005.

[22] A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):464–509, 2004.

[23] R. Rugina. Quantitative shape analysis. In Giacobazzi [5], pages 228–245.

[24] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.