

Explicit Data Structures and Automatic Induction Proofs

Abstract

We first present a specification language for modelling memory explicitly as an array, and models pointers and data elements uniformly as integers, used here as an abstraction of a machine word. There are three distinguishing features: the use of recursive definitions, the use of set variables representing *explicit footprints* in order to implement the concept of *separation*, and finally, the language can express the *strongest postcondition* of a loop-free program. This provides a basis for implementing a Floyd-Hoare-style program verifier which generates verification conditions for an external theorem prover.

We then present a prover in the form of inference rules that are systematically applied. We argue that a large class of verification conditions can thus be automatically proven. The main novelty is the ability to employ induction hypotheses that are *automatically generated* in order to reason about the interplay between recursive definitions, arrays, sets and integers. Finally, recursive definitions are reduced into purely integer formulas, which we can finally dispense with using a standard system.

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 need 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. For example, the recent advance in Separation Logic [20] provided a succinct way to specify the separation of “footprints” of data structures, a concept which is frequently useful. The development of Separation Logic inference rules then suggested that proving certain verification conditions involved intuitively appealing steps. In our language, we model the notion of footprint explicitly as a set expression, and will show that this methodology is as intuitive in both the specification and proving phases.

This paper comprises two main parts. We first define 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.

The second and main part is a proof method which is based on the standard concept of fold/unfold for recursive definitions. Traditional approaches perform fold/unfold operations until the proof obligation is either trivial (e.g. a tautology), or it is subsumed by a previous proof obligation, i.e. a form of “loop-checking”. What is novel here is the use of *automatic induction*, a method which permits a dynamically generated but yet unproved obligation to be *assumed true* within the proof process itself. We thus can obtain automatic proofs of important classes of properties which traditionally required proofs by induction where the induction hypotheses is manually provided.

The unfolding process ultimately reduces the proof obligation to a basic formula or “constraint” that no longer contains recursive predicates. The algorithm then arithmetizes the remaining constraints, involving array, multiset and integer constraints, into an integer formula. Thus, at this point, the remaining proof obligation can be dispensed with standard constraint solvers.

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 [10, 5, 17, 23]. For example, the “negation as failure” inference in [10] is akin to our left unfold rule, while the “definite clause inference” step in [10, 11] is akin to our right-unfold step. In [23], fold/unfold transformations are performed toward the objective of transforming two programs into syntactically identical ones. These do not explicitly

use induction hypotheses. Recent work [3] provides a method for proving the equivalence of general CLP programs that makes use of a coinduction rule. However, all these works do not address the problem of mutable data structures.

In contrast, our method for reducing recursive definitions is based on [8] in which the central step is the use of certain existing proof obligations as *inductive hypotheses*. This induction-based method does not require a base case. Therefore the key advantage is that our method is automatic.

With regard to 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 fix-point iteration algorithm. As argued in [4], 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 [21, 22]. Also, [9] 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, 18], which is based on Hoare logic, and PALE [18] verifier can be efficiently run when loop invariant is given. The paper [16] 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. These works on shape analysis and customized reasoning about data structures do not model memory explicitly.

A significant advance is Separation Logic [20], an extension of Hoare logic for reasoning about programs that use shared mutable data structures. At its core there are separation constructs that allow the specification of program properties that hold in separate parts of the underlying heap. This logic is demonstrated in the context of user-defined recursive definitions.

A recent work [19] provides some level of automation to Separation Logic. They consider 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 an induction rule. They do not consider arrays.

Finally, the advantages of explicit footprints and their amenability to automated reasoning is folklore. Some recent works are [12][25] which deal with footprints of procedures and methods, but not of recursive data structures. We however will elaborate in section 3 to focus on our particular context of data structures and recursive definitions.

2. The Specification Language

2.1 Basic Specifications - Constraints

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 [15] axioms:

$$\begin{aligned} i = k &\rightarrow \langle a, i, j \rangle[k] = j \\ i \neq k &\rightarrow \langle a, i, j \rangle[k] = a[k] \end{aligned}$$

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 $\{a[i], a[i+1], \dots, a[j]\}$.

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:

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

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 -ARC 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 Recursive Specifications

We present some preliminary definitions about CLP [6]. An *atom* is of the form $p(\vec{t})$ where p is a user-defined predicate symbol and \vec{t} a tuple of terms, as defined above. A *rule* is of the form $A : -\Psi, \vec{B}$ where the atom A is the *head* of the rule, and the sequence of atoms \vec{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 $p(\vec{t})$ is an object of the form $p(\vec{t}\theta)$. A grounding θ of a goal $G \equiv (p(\vec{t}), \Psi)$ is a grounding θ of $p(\vec{t})$ where $\Psi\theta$ is *true*. We write $\llbracket G \rrbracket$ to denote the set of groundings of G .

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

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

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

A *derivation sequence* for a goal G_0 is a possibly infinite sequence of goals G_0, G_1, \dots where $G_i, i > 0$ is a reduct of G_{i-1} . If the last goal 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 G , $\text{UNFOLD}(G)$ is $\{G' \mid \exists R \in P : G' = \text{REDUCT}(G, R)\}$. \square*

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

We describe here the use of constraints and (recursive) predicates as assertions, i.e. as specifications of program variables at given program points.

We assume all variables in a user program are integer variables except one special array variable h , called the *heap*, representing the entire memory. We write $*x$ to denote $h[x]$, and where there is an underlying structure for a cell that is pointed to, we write $x \rightarrow f$ to denote the field of the structure pointed to by x (in the tradition of C). Thus, for example, when dealing with structures for a binary tree of integers, $*x$ denotes the value of the node while $x \rightarrow \text{left}$ and $x \rightarrow \text{right}$ can be used to denote the values of the cells pointed to by the left and right fields of the cell pointed to by x . Thus assuming that the left and right pointers of a structure are positioned immediately after the value of the structure, $x \rightarrow \text{left}$ and $x \rightarrow \text{right}$ are equivalently, $*(x+1)$ and $*(x+2)$ respectively.

A basic assertion is expressed directly in CLP as a constraint, using the original program variable names. A recursive assertion is one that uses CLP predicates whose arguments may contain some or all of the original program variable names, in addition to new variables. An example of a new variable is a multiset variable which appears in the predicate but not in the program.

A typical predicate is one which describes a property, say positiveness, of the elements a singly-linked list, e.g. the following predicate b says that the elements in the list x are bigger than 999. Note that we assume that the offset “nxt” here refers to the pointer field.

```
biglist(H, X) :- X = 0.
biglist(H, X) :-
  X > 0, H[X] > 999, biglist(H, H[X + nxt]).
```

Such a predicate is a “one-heap” predicate because it mentions just one array variable h . Another typical kind of predicate is a “two-heap” predicate mentioning two array expressions. This is used to relate the current heap with the value of the heap at another program point. For example, the following predicate p can be used to say that the list x in the current heap h is the same as the list x in another heap h_0 except that all the elements have now been set to zero. One important point to note here is that the heaps h and h_0 are *identical* outside the footprint of x .

```
allzero(H, H, X) :- X = 0.
allzero(H0, H, X) :-
  X > 0, H[X] = 0, allzero(H0, H, H0[X + nxt]).
```

Note that in these two definitions, the use of the variables such as h and h_0 are the same as the program variables by coincidence only. We could have used any other names. However, when these predicates are used as *assertions*, it is then important to match the names of the variables in the predicates with the names of the program variables to which they correspond.

Abbreviation of Predicates

It is almost universal that the variable h is used in predicates (for it represents the memory) and so we shall omit writing h when defining predicates, when no confusion is possible. Correspondingly, when we refer to the field of a structure in the context of the (current) heap h , we shall use the more recognizable syntax mentioned above. More precisely, we write $*x$ to abbreviate $h[x]$, and instead of writing $h[x+1]$ and $h[x+2]$ to denote the left and right pointers of a node in a binary tree residing in the heap h , we shall write $x \rightarrow \text{left}$ and $x \rightarrow \text{right}$ respectively. Thus for example, we could rewrite the definition of the predicate $biglist$ into:

```
biglist(X) :- X = 0.
biglist(X) :- X > 0, *X > 999, biglist(X→nxt).
```

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)
```

Predicates:

```
sorted(I, N) :- I = N.
sorted(I, N) :- I < N, *I ≤ *(I+1), sorted(I+1, N).
max(Y, _) :- 0 > Y.
max(Y+1, U) :- 0 ≤ Y+1, *(Y+1) ≤ U, max(Y, U).
```

Figure 1. Bubble Sort

2.4 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 1. The CLP definition of $sorted(i, n)$ is a one-heap predicate that specifies that the sequence of cells $*i, *(i+1), \dots, *(i+n)$, if $i < n$, is an ordered sequence. The predicate $max(y, u)$ is *true* if u is an upper bound of the values $*0, *(1), \dots, *y$.

We will later exemplify two proofs of the inner loop B (between (2) and (5) of the bubble sort program in Figure 1). The “Hoare triples” are:

$$\{j = 0, 0 \leq i < n-1, max(n-i-1, n-i), sorted(h, n-i, n)\} \quad B \quad (1)$$

$$\{0 \leq i < n-1, max(n-i-2, n-i-1), sorted(h, n-i-1, n)\},$$

and

$$\{j = 0, 0 \leq i < n-1, h = h_0\} \quad B \quad (2)$$

$$\{0 \leq i < n-1, h_0\{0 \dots n-1\} = h\{0 \dots n-1\}\}$$

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.

2.5 An List Example: Reverse

The CLP program for $reverse(h_1, h_2, i_1, i_2, j)$ in Figure 2 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 are used to specify that the list j_2 is an update of the list h_1 , hence the reverse operation is *in-situ*. Just as importantly, note that the predicate also says that the two heaps are *identical* for addresses outside the list in question.

The CLP program for $alist(h, l, s)$ defines an acyclic list whose set of node addresses is s .

3. Separation and Explicit Footprints

Recent work on verification of programs with shared mutable data structures [20] introduced the concept of *Separation Logic* 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

Program:

```

{alist(h0, i0, m0)}
j:=0
(0) while (i>0) do
(1)   k, [i+1], j:= [i+1], j, i
      i:=k end (2)
{reverse(h0, h, i0, 0, j), alist(h, j, m0)}

```

Predicates:

```

rev(H, H, I, I, 0).
rev(H1, (H2, J+1, New), I, Old, J) :-
  H2[J+1] = Old, rev(H1, H2, I, J, New).
alist(H, L, {}).
alist(H, L, S ∪ {L, L+1}) :-
  L > 0, {L, L+1} ⊗ S, alist(H, H[L+1], S).

```

Figure 2. List Reverse

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 $\Psi_1 \star \Psi_2 \star \dots \star \Psi_n$, $n \geq 2$ by a *separation* operator \star . The interpretation of a heap predicate is just a heap in which the predicate is true. The interpretation of $\Psi_1 \star \Psi_2 \star \dots \star \Psi_n$ is a heap which can be decomposed into n disjoint heaps H_i in which Ψ_i holds, $2 \leq i \leq n$.

Some important advantages of Separation Logic are:

- *Succinctness*

The separation operator \star concisely specifies that given formulas hold and simultaneously, that their "footprints" are disjoint.

- *Ease of Proof*

Separation Logic formulas, when applied to by a generator of verification conditions, produces entailment formulas whose proof depends on steps that are intuitive.

- *Modularity*

There is a key rule, the "Frame Rule", allowing the preservation of a property across a program fragment provided that the property's footprint is disjoint from that of the program fragment.

We now demonstrate our language, focusing on these aspects, in order to compare with Separation Logic.

3.1 Succinctness

In Separation Logic, the expression $\Psi \star \Phi$ states not only that the formulas Ψ and Φ hold, but that they hold in separate heaps, that is, in separate parts of overall memory. Using an explicit modelling of memory, it is folklore that to achieve this, one would have to specify that each relevant heap location in one formula is not equal to *all* the relevant heap locations in the other formula. This if the two formulas describe two lists of length n , the explicit specification would involve $O(n^2)$ disequalities. This is exemplified by using a recursive definition as follows, which unrolls into $O(n^2)$ invocations of the formula $x \neq y$:

```

list(0).
list(X) :- X > 0, list(X→nxt).
separate_list(0, Y).
separate_list(X, Y) :-
  X > 0, X != Y, list(X→nxt),
  separate_list(X→nxt, Y).

```

However, in our framework, we achieve the specification of separation by first creating an assertion predicate p_i which defines the heap predicate Ψ_i , and explicitly mentions its heap locations as a

multiset¹ variable \mathcal{M}_i . We shall informally call \mathcal{M}_i the *footprint* of the predicate. Then, we simply add the constraint

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

For the example immediately above, our proposed encoding of separation would be to use a set variable associated with the recursive definition of a list:

```

list(0, {}).
list(X, {X} ∪ S) :- X > 0, list(X→nxt, S).

```

so that the separation of two lists, say $\text{list}(x_1, S_1)$ and $\text{list}(x_1, S_2)$, can be simply specified by the (non-recursive) formula $S_1 \otimes S_2$.

3.2 Ease of Proof

We next deal with the more important aspect of whether the *proof process* of formulas (arising from the generation of verification conditions) is in fact simplified by this encoding using set variables. Here we revisit the canonical "reverse" example in [20]. The program is displayed in Figure 2. Let us examine the proof of the loop invariant, summarized as follows. Note that the symbol \star denotes the separation operator in Separation Logic, α and β denote sequences and the notation α^R denotes the reverse sequence α .

$$\begin{aligned}
& \text{list } a.\alpha(i, \text{nil}) \star \text{list } \beta(j, \text{nil}), \alpha_0^R = (a.\alpha)^R.\beta \\
& i \mapsto a, k \star \text{list } \alpha(k, \text{nil}) \star \text{list } \beta(j, \text{nil}), \alpha_0^R = (a.\alpha)^R.\beta \\
& i \mapsto a, k \star \text{list } \alpha(k, \text{nil}) \star \text{list } \beta(j, \text{nil}), \alpha_0^R = (a.\alpha)^R.\beta \\
& i \mapsto a, j \star \text{list } \alpha(k, \text{nil}) \star \text{list } \beta(j, \text{nil}), \alpha_0^R = (a.\alpha)^R.\beta \\
& \text{list } \alpha(k, \text{nil}) \star \text{list } a.\beta(i, \text{nil}), \alpha_0^R = (a.\alpha)^R.\beta
\end{aligned}$$

The key steps in this proof in fact reasons as follows:

- If l is a list, then its head h and tail t are separate, i.e. $h \star t$ holds.
- If l and l_2 are separate lists, then the head h of l may be appended to l_2 form a new list, i.e. $\text{list } h.l_2$ holds.

In our framework, this step essentially translates into the straightforward proof of the following, where the predicate list has been defined above:

$$\text{list}(i, s), j \rightarrow \text{nxt} = i, j \otimes s \models \text{list}(j, -)$$

Note that in this example, it was not proven that the reversal took place in-situ. That is, there was no specification about how the final heap relates to the initial heap. In contrast, we proved both these properties in the example on Figure 2 above.

3.3 A Frame Rule for Modular Reasoning

Perhaps the main strength of Separation Logic is that it facilitates modular or compositional reasoning. This is realized in the form of a *frame rule*:

$$\frac{\{\Psi\} P \{\Phi\}}{\{\Psi \star \Psi\} P \{\Phi \star \Psi\}}$$

where no occurrence of a variable free in Ψ is modified by the program fragment P . This allows a (previously established) proof that the footprint of the program P is contained in the heap indicated by Ψ to infer that any property separated from Ψ remains unchanged by P .

In our framework, the notion of separation is not *built-in*, but rather *user-defined* in the form of set variables, representing, amongst other things, sets of memory addresses. Thus, in order to use a frame rule (whether in a manual or automated setting), it is necessary to first verify that the relevant set variable in fact behaves like a footprint.

This is tantamount to proving a Hoare triple of the form:

$$\{p(x, s), y \otimes s\} \ *y := e \ \{p(x, s)\}$$

¹ In this paper, we use set and multiset synonymously.

which in turn means to prove a formula of the form:

$$p(h, x, s), y \otimes s \models p(\langle h, y, e \rangle, x, s) \quad (1)$$

where p is the predicate of interest, h and $\langle h, y, e \rangle$ denote respectively the heap immediately before and after the assignment, and e is any expression. Thus the formula (1) says that the property p wrt the "handle" x is unaffected by the assignment to the heap location y . We now informally show, assuming a "typical" formulation of p , that the proof can easily be done automatically. Later sections of the paper makes provides for the formal proof.

A most typical formulation of p is a tail-recursive specification of a list, say that all its elements are zero:

$$\begin{aligned} & p(H, 0, \{\}) . \\ & p(H, X, \{X\} \cup S) :- H[X] = 0, p(H, H[X+1], S) . \end{aligned}$$

Returning to formula (1), let us "unfold" the definition of p on the lhs, and obtain two subobligations corresponding to each of the two rules defining p . The first case, using the first rule for p , results in:

$$x = 0, s = \{\}, y \otimes s \models p(\langle h, y, e \rangle, x, s) \quad (2)$$

and this is straightforward because both the predicates $p(h, x, s)$ and $p(\langle h, y, e \rangle, x, s)$ reduce to *true* in the context $x = 0, s = \{\}$. The second case, obtained from unfolding using the recursive rule defining p , results in:

$$h[x] = 0, p(h, h[x+1], s - \{x\}), y \otimes s \models p(\langle h, y, e \rangle, h[x+1], s - \{x\}) \quad (3)$$

Now the crucial step: we note that the lhs contains an instance of (1), and replacing this instance with the rhs of (1), we get

$$*x = 0, p(\langle h, y, e \rangle, h[x+1], s - \{x\}), y \otimes s \models p(\langle h, y, e \rangle, h[x+1], s - \{x\}) \quad (4)$$

This is an example of a step we call "induction application", which is formalized later. Intuitively, what we have done is simply to *assume* that the original proof obligation (1) is true when we are trying to prove antecedents of the obligation. We do not use explicitly resort to any notion of a level of induction, an induction hypothesis, nor a base case. Instead, the consideration of a base case is *inherited* from the definition of the predicate.

The proof of (4) is obvious and we are done.

We have just described how the proof system can be used to "certify" that the use of the distinguished multiset variable s in the definition of p indeed corresponds to s being a footprint of p . We did this by proving that the definition of $p(h, x, s)$ is such that when the heap h is changed at a location outside s , the property p continues to hold. Since it is almost always the case that the predicate definition is such that the multiset variable is *faithful* to the definition, it is very useful to perform such certification offline. We show concretely how we use these certifications in the FLATTEN rule in Section 5 later.

In order to use such a certified predicate with program fragments in a modular way, we also need to *certify the program*. The process here is essentially an extension of the proof of (1) above. The idea is to attach, given a program P , an assertion just before each assignment in P . More precisely, at every assignment $*y = e$ in P , the assertion states that y is disjoint from the set variable in question. For example, if the predicate of interest were $p(x, s)$, the assertion required just before an assignment of the form $*y = e$ would be $y \otimes s$. The proof process then disposes of the proof of the entire program in the usual way. We omit the details in this informal section. In the next section, we formalize the generation of verification conditions from an annotated program. If the annotated program were proved, we may then say that the program P has been certified to have the footprint *away from* s .

We can now state our Frame Rule as follows. Let P be a program with certified footprint s and p a predicate with certified footprint s_2 . Then the following Hoare triple holds.

$$\{p(x, s_2)\} P \{p(x, s_2)\}$$

Note that this rule is not formally required in our proof system (since its result can directly be proven from first principles). However, it can be useful for compositional reasoning in both an automated as well as a manual setting. In fact, we use this in our algorithm, detailed in section 5.4.

3.4 Explicit Footprints go beyond Separation

We conclude this section by exemplifying the use of set variables beyond that of specifying footprints. Rather, we show here, that a multiset variable can usefully specify subsets of a footprint. Consider for example the specification of a list of integers and the sum of these integers which are positive:

$$\begin{aligned} & \text{sumpos}(0, 0, \{\}) . \\ & \text{sumpos}(X, \text{Sum}, S) :- \\ & \quad *X \leq 0, \text{sumpos}(X \rightarrow \text{nxt}, \text{Sum}, S) . \\ & \text{sumpos}(X, \text{Sum} + *X, S \cup \{X\}) :- \\ & \quad *X > 0, \text{sumpos}(X \rightarrow \text{nxt}, \text{Sum}, S) . \end{aligned}$$

Now suppose we assign address y with the value 0. Clearly the predicate $\text{sumpos}(x, \text{sum}, s)$ continues to hold, regardless of whether y is an address within the list x , or not. That is, we can prove both

$$\begin{aligned} & \{\text{sumpos}(l, s, \text{sum}), y \otimes s\} *y = 0 \{\text{sumpos}(l, s, \text{sum})\} \\ & \{\text{sumpos}(l, s, \text{sum}), *y < 0\} *y = 0 \{\text{sumpos}(l, s, \text{sum})\} \end{aligned}$$

although the proof in each case is different. We omit further details.

In summary for this subsection: because our multisets are not built-in to represent footprints, this provides for their use with arbitrary flexibility. The subsequent dispensing of the verification conditions that arise then is entrusted to the general inference rules presented in this paper.

4. Generation of Verification Conditions

In this brief section, we provide an algorithm for the generation of verification conditions given an annotated program. In the usual Floyd-Hoare tradition, it essentially suffices to have a rule for the assignment statement. We note that in contrast, Separation Logic does not provide such an algorithm².

Define a function *post* representing the *strongest postcondition* of an assignment statement as follows. Let x' denote a fresh collection of variables representing the program variables.

$$\begin{aligned} \text{post}(\Psi, x = e) & \stackrel{\text{def}}{=} x = e[x/x'], \Psi[x/x'] \\ \text{post}(\Psi, *x = e) & \stackrel{\text{def}}{=} h = \langle h', x, e \rangle, \Psi[h/h'] \end{aligned}$$

Extending the function *post* to if-statements is obvious:

$$\text{post}(\Psi, \text{if}(b) s_1 \text{ else } s_2) \stackrel{\text{def}}{=} \text{post}(\Psi \wedge b, s_1) \vee \text{post}(\Psi \wedge \neg b, s_2)$$

Finally we deal with loops: assuming that each loop is provided with a loop invariant I , we have:

$$\text{post}(\Psi, \text{while } b \text{ do } s) \stackrel{\text{def}}{=} I \wedge \neg b$$

where there is a side condition: $\{I \wedge b\} s \{I\}$. We note that the function *post*, now including the consideration of loops, does not necessarily compute the strongest postcondition. It does, however, compute the strongest postcondition wrt the invariant I .

² See eg: [1] which provide an algorithm for a subset of Separation Logic.

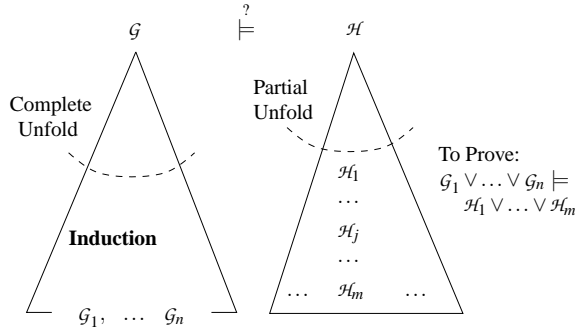


Figure 3. Informal Structure of Proof Process

Note that the generation process is now *complete* in the sense of Cook [2].

Finally, the verification process is now simply stated: in order to prove the Hoare triple $\{\Psi\}P\{\Phi\}$, we prove the verification condition $post(\Psi, P) \models \Phi$, and all side conditions (which in general will generate more verification conditions).

5. Proof of Verification Conditions

In this key section, we consider proof obligations of the form $G \models H$ where G and H are CLP goals and $var(H) \subseteq var(G)$. The validity of this formula expresses the fact that $H\theta$ succeeds w.r.t. the CLP program at hand whenever $G\theta$ succeeds, for any grounding θ of 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.

5.1 Unfolding Recursive Assertions

Intuitively, we proceed as follows: unfold G completely a finite number of steps in order to obtain a “frontier” containing the goals G_1, \dots, G_n . Then unfold H , but this time not necessarily completely, that is, not necessarily obtaining *all* the reducts each time, obtain goals H_1, \dots, H_m . This situation is depicted in Figure 3. Then, the proof holds if

$$G_1 \vee \dots \vee G_n \models H_1 \vee \dots \vee H_m$$

or alternatively, $G_i \models H_1 \vee \dots \vee H_m$ for all $1 \leq i \leq n$. This follows easily from the fact that $G \models G_1 \vee \dots \vee G_n$, and $H_j \models H$ for all j such that $1 \leq j \leq m$. More specifically, but with some loss of generality, the proof holds if

$$\forall i: 1 \leq i \leq n, \exists j: 1 \leq j \leq m: G_i \models H_j$$

and for this reason, our *proof obligation* shall be defined below to be simply a pair of goals, written $G_i \models H_j$.

5.2 Proof Rules

We now present a formal calculus for the proof of $G \models H$. To handle the possibly infinite unfoldings of G and H , we shall depend on the use of a key concept: *induction*. Proof by induction 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 $\tilde{A} \vdash G \models H$ where the G and H are goals and \tilde{A} is a set of assumption goals. \square

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

Our proof rules³ are presented in Figure 4. 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 induction hypothesis* (LU+IH) is key rule. It 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 induction 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 inductively or directly, using the IA and CP rules, respectively.

The *rule induction application* (IA) 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+IH rule, the IA rule realizes the coinduction principle. The underlying principle behind the (IA) rule is that a “similar” assertion $G' \models H'$ has been previously encountered in the proof process, and assumed as true⁴.

Note that this test for coinduction applicability is itself of the form $G \models H$. 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.

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 $p(\bar{x})$ 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. We do not discuss an algorithm for such direct proofs; instead we resort to standard methods, eg [14].

Given a proof obligation $G \models H$, a proof shall start with $\Pi = \{\tilde{A} \vdash G \models H\}$, and proceed by repeatedly applying the rules in Figure 4 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 $G \models H$ holds if, starting with the proof obligation $\emptyset \vdash G \models H$, there exists a sequence of applications of proof rules that results in proof obligations $\tilde{A} \vdash G' \models H'$ such that (a) H' contains only constraints, and (b) $G' \models H'$ can be discharged by the constraint solver. \square

³ We place the obligation on top, and its reduced form at the bottom.

⁴ In fact, the repeating pattern corresponds to a loop in the original programming language, and H acts as an invariant.

(LU+IH)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^n \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}}$	UNFOLD(\mathcal{G}) = $\{\mathcal{G}_1, \dots, \mathcal{G}_n\}$
(RU)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{1 \leq i \leq k} \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}}$	$\mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$
(IA)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{H}' \theta \models \mathcal{H}\}}$	$\mathcal{G}' \models \mathcal{H}' \in \tilde{A}$ and there exists a substitution θ s.t. $\mathcal{G} \models \mathcal{G}' \theta$
(CP)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\bar{x}) \models \mathcal{H} \wedge p(\bar{y})\}}{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \bar{x} = \bar{y}\}}$	

Figure 4. General Proof Rules

5.3 Automatic Induction

Here we elaborate on the key LU+IH rule, and exemplify the automatic proof of an inductive property. As mentioned above, a standard approach to disposing of entailments of recursively defined predicates is a process of folding/unfolding until the proof obligation becomes trivial. This means that the proof obligation becomes a tautology, or it is subsumed by a previous proof obligation. We call the latter situation a “loop”.

This process is akin to a process of computing a transitive closure. It is also akin to the proof of equations defined by term-rewriting systems, where the folding/unfolding serves to arrive at a *normal form* so that equality reduces to the problem of having the same normal form⁵.

In contrast, our left unfold rule automatically tables the parent goal as an induction hypothesis, which can be used in the scope of the subsequent proof process. We consider with a classic example:

```
fib(0, 0).
fib(1, 1).
fib(X, Y1+Y2) :-
  X >= 2, fib(X - 1, Y1), fib(X - 2, Y2).
```

and outline a proof of $\text{fib}(x, y) \models x \leq y$. Consider just the recursive rule above (the other two cases are straightforward) and unfold the formula into:

$$x \geq 2, \text{fib}(x-1, y_1), \text{fib}(x-2, y_2) \models x \leq y \quad (5)$$

At this point, the original obligation is *tabled* as an induction hypothesis. Next, we can *apply* this hypothesis to reduce the expression $\text{fib}(x-1, y_1)$ in (5) into $x-1 \leq y_1$. Similarly doing this with the other expression in (5) results in

$$x \geq 2, x-1 \leq y_1, x-2 \leq y_2 \models x \leq y$$

which clearly can be finally dispensed with by a standard theorem-prover.

Note that we have not explicitly used an induction hypothesis nor a base case, and hence the automation. As mentioned above, the consideration of a base case is *inherited* from the definition of the predicate. In this example, had the two base cases of the fib predicate been missing, the proof obligation (5) would still be provable. Of course, since the definition of fib is vacuous, this proof is hardly interesting.

There are some subtle points over this seemingly trivial use of induction. Consider another example:

```
double(0, 0).
double(X+1, Y+2) :- double(X, Y).
```

⁵Strictly speaking, we also require that the system is *confluent*.

We can prove $\text{double}(x, y) \models y = 2x$ in much the same way as we proved the *fib* formula above. However, we can *not* prove $y = 2x \models \text{double}(x, y)$. Technically, this is because the use of induction is preconditioned by a previous use of the left-unfold rule. In fact, a standard approach to conducting the proof here would involve well-founded induction, and thus be harder to automate.

We conclude this subsection with a final example showing the use of induction on a programming example. Imagine a loop in which the elements of a list are being assigned to zero. The predicate $\text{zlistseg}(x, y, s)$, stating that the list starting at x and ending at y contains only zero elements, is naturally suited as a loop invariant. However, it is more natural to use the simpler predicate zlist to define simply a list of all zero elements.

```
zlistseg(X, Y, {X}) :- X = Y, *X = 0.
zlistseg(X, Y, {X} \cup S) :-
  X != Y, *X = 0, X \otimes S, zlistseg(X \to \text{nxt}, Y, S).
```

```
zlist(0, {}).
zlist(X, {X} \cup S) :-
  X > 0, *X = 0, zlist(X \to \text{nxt}, S).
```

Thus a verification condition that arises is of the form

$\text{zlistseg}(x, y, s), y \rightarrow \text{nxt} = 0 \models \text{zlist}(x, s)$. However, as with the *fib* example above, we cannot prove this example by a simple process of folding/unfolding (using loop-checking), but can do so with the use of our induction rule. We omit the details.

5.4 A Systematic Strategy for Applying Rules

We now describe a systematic strategy for applying certain rules in order to dispense a proof obligation $\mathcal{G} \models \mathcal{H}$ mechanically. In Figure 5, the function `consproof` performs basic constraint solving as prescribed by the CP rule in Figure 4. It acts as the base case of our recursive algorithm.

Note that the algorithm is nondeterministic because of a **choose** construct. The first and outermost one chooses one strategy out of three. The second chooses one of the reducts of \mathcal{H} . Overall termination, which is not guaranteed, is obtained when one proof is found. Clearly this nondeterminism is the main challenge, for it can produce a large search tree of proof obligations. However, recursive definitions are typically small (they are intended to be understandable specifications, not programs). We demonstrate the algorithm later in section 6.

The main features of the algorithm are

- a MONOCUT rule to select just one of several atoms in the lhs \mathcal{G}
- a MONOSPLIT rule for disposing each atom in the rhs \mathcal{H} in *sequence*
- tabling only *monogoals* as induction hypothesis, and
- flattening array expressions within certified predicates.

We begin with the rules:

$$\text{(MONOCUT)} \quad \frac{\Pi \uplus \{\tilde{A} \vdash g, \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash g \models \mathcal{H}\}} \quad \text{where } g \text{ is a monogoal}$$

$$\text{(MONOSPLIT)} \quad \frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models h, \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \ \& \ \theta \models \mathcal{H}\}} \quad \tilde{A} \vdash \mathcal{G} \ \& \ \theta \models h$$

The first rule MONOCUT is key: it chooses exactly one atom on the lhs \mathcal{G} in order to attempt the proof. This is a specialized version of what is commonly known as a “cut” rule, and is clearly not completely general. Cut rules have the problem of automation, for it is not always clear in what way to generalize the hypothesis goal

\mathcal{G} . Our formulation solves this problem, and so it provides a basis for a tractable algorithm.

The important thing is that this particular rule, in conjunction with the other steps of the algorithm, is sufficient for a large class of proof obligations, including all the examples in this paper. The general intuition behind this sufficiency is that it works *in tandem* with the right unfold rule (which reduces one atom on the rhs into possibly several atoms) and the MONOSPLIT rule which in turn focuses on each of these atoms individually. Ultimately, the resulting monogoals, goals which contain at most one atom, can typically be proven by induction.

The second rule (MONOSPLIT) simply breaks the problem down by proving one atom h in the rhs at a time. Upon each application of this rule, a “binding” θ indicating values for existential variables (not appearing in \mathcal{G}) that the constructive proof process produced. When this rule is used in conjunction with the (MONOCUT) rule, the prover, after a search process, has ultimately to deal with proof obligations of monogoals, ie: goals with at most one atom.

These two rules, in turn, means that the the LU+IH rule, which tables new induction hypotheses on-the-fly, to monogoals. This is important because the application of induction requires a test that a proposed goal is an instance of some induction hypothesis. With general goals, this raises a combinatorial problem of choosing which atoms in the proof obligation match the atoms in the induction hypothesis. In short, the tabling of monogoals is a major factor toward tractable implementation.

The last, but not least, main feature of the algorithm is to use *certified* predicates in order to reduce or flatten array expressions appearing in the predicate. Recall that in subsection 3.3, we described how a definition of a predicate p with a distinguished set variable s can be certified in the sense that s is proved to be a faithful footprint of p . That is, we (in an initial and separate phase) prove that the definition of $p(h, x, s)$ is such that when the heap h is changed at a location outside s , the property p continues to hold. We then say $p(h, x, s)$ is certified wrt to h and s . We also indicated in subsection 3.3 that such a certification proof can be done using induction. We exemplify this more precisely below.

Now, our algorithm will perform simplification of expressions in the proof obligation $\mathcal{G} \models \mathcal{H}$ by using the following rule, whose informal syntax simply means to *flatten* any array expression $\langle h, i, j \rangle$ appearing in a context such as $p(\langle h, i, j \rangle, x, s)$ into $p(h, x, s)$ whenever it is established that $i \otimes s$

$$\text{(FLATTEN)} \quad \frac{\Pi \uplus \{\tilde{A} \vdash \dots p(\langle h, i, j \rangle, x, s) \dots\}}{\Pi \cup \{\tilde{A} \vdash \dots p(h, x, s) \dots\}} \quad p(h, x, s) \text{ certified for } h, s \text{ and } i \otimes s$$

Recall the main algorithm in Figure 5. We finally remark that side condition for applying induction uses the `consproof` function, ie a constraint solver. It is in fact possible to generalize the side condition by a (recursive) use of the *solve* process itself (with an empty set of induction hypothesis as a base). This would clearly make the search space larger. We have chosen the simpler use at this time because it is sufficient for all of the examples we have investigated.

5.5 Examples of Proofs

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:

`solve($\mathcal{G} \models \mathcal{H}, I$) returns θ`

```

let  $\mathcal{G} = \Psi, g_1, \dots, g_m$  and  $\mathcal{H} = \Phi, h_1, \dots, h_n$ 
foreach  $i = 0$  to  $n$ 
  foreach  $j = 1$  to  $m$ 
    if  $(i = 0)$   $h = \Phi$ ;  $\theta = \text{consproof}(\Psi, h)$ 
    else  $h = h_j$ ;  $\theta = \text{consproof}((\Psi, g_j), h)$ 
    if  $\theta$  return  $\theta$ 
    choose
      (induction)
        suppose  $\mathcal{G}' \models h' \in I$  and  $\beta = \text{consproof}(\mathcal{G}', g_j)$ 
        if  $(\theta = \text{solve}(h' \beta \models h, I))$  continue
        else return  $\perp$ 
      (right unfold)
        choose one reduct  $h'$  of  $h$ 
        if  $(i > 0 \ \& \ \theta = \text{solve}(g_j \models h', I))$  continue
        else return  $\perp$ 
      (left unfold)
        forall reduct  $g'$  of  $g_j$ 
          if  $(\theta = \text{solve}(g' \models h, I \cup \{\mathcal{G} \models \mathcal{H}\}))$  continue
          else return  $\perp$ 
        endchoose
    endfor
   $\mathcal{G} := \mathcal{G} \ \& \ \theta$ 
endfor
if  $(\theta = \text{solve}(\mathcal{G} \models \Phi, I))$  return  $\theta$  else return  $\perp$ 

```

Figure 5. The Algorithm

$$\begin{aligned} i_f = i, n_f = n, 0 \leq i_f < n_f - 1, \\ \max(n_f - i_f - 2, *(n_f - i_f - 1)), \\ \text{sorted}(n_f - i_f - 1, n_f - 1) \models \\ i_f = i, n_f = n, \text{sorted}(n_f - i_f - 2, n_f - 1). \end{aligned} \quad (S.1)$$

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

$$\begin{aligned} i_f = i, n_f = n, 0 \leq i_f < n_f - 1, \\ 0 \leq n_f - i_f - 2, *(n_f - i_f - 2) \leq *(n_f - i_f - 1), \\ \max(n_f - i_f - 3, *(n_f - i_f - 1)), \\ \text{sorted}(n_f - i_f - 1, n_f - 1) \models \\ i_f = i, n_f = n, n_f - i_f - 2 < n_f - 1, \\ *(n_f - i_f - 2) \leq *(n_f - i_f - 1), \\ \text{sorted}(n_f - i_f - 1, n_f - 1) \end{aligned} \quad (S.2)$$

Next we replace both array references $*(n_f - i_f - 2)$ and $*(n_f - i_f - 1)$ with simple integer variables.

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

Reverse

Consider the list reverse example in Figure 2. 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.

In what follows we present one particular path that the algorithm in Figure 5 would traverse, and succeed.

We prove the loop invariant $\Psi \equiv \exists t, u. \text{reverse}(h_0, h, i_0, i, j), \text{alist}(h, j, t), \text{alist}(h, i, u)$. which amounts to proving:establishing the following obligation:

$$\begin{aligned} \text{reverse}(h_0, h, i_0, i, j), \text{alist}(h, j, t), \text{alist}(h, i, u), \\ t \otimes u, s = t \cup u, i > 0 \models \\ \text{reverse}(h_0, \langle h, i + 1, j \rangle, i_0, h[i + 1], i), \\ \text{alist}(\langle h, i + 1, j \rangle, i, ?t'), \\ \text{alist}(\langle h, i + 1, j \rangle, h[i + 1], ?u'), \\ ?t' \otimes ?u', ?t' \cup ?u' = s \end{aligned} \quad (R.1)$$

First left unfold the atom $alist(h, i, u)$ in the lhs resulting in two obligations which must be proved separately. One of them is:

$$\begin{aligned}
& reverse(h_0, h, i_0, i, j), alist(h, j, t), alist(h, h[i+1], u_1), \\
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& reverse(h_0, \langle h, i+1, j \rangle, i_0, h[i+1], i), \\
& alist(\langle h, i+1, j \rangle, i, ?t'), \\
& alist(\langle h, i+1, j \rangle, h[i+1], ?u'), \\
& ?t' \otimes ?u', ?t' \cup ?u' = s
\end{aligned} \tag{R.2b}$$

Perform a right unfolding step on the rhs atom $reverse$ results in:

$$\begin{aligned}
& reverse(h_0, h, i_0, i, j), alist(h, j, t), alist(h, h[i+1], u_1), \\
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& reverse(h_0, h, i_0, i, j), \\
& alist(\langle h, i+1, j \rangle, i, ?t'), \\
& alist(\langle h, i+1, j \rangle, h[i+1], ?u'), \\
& ?t' \otimes ?u', ?t' \cup ?u' = s
\end{aligned} \tag{R.3}$$

Another right unfolding step on the first occurrence of the $alist$ predicate on the rhs results in the following:

$$\begin{aligned}
& reverse(h_0, h, i_0, i, j), alist(h, j, t), alist(h, h[i+1], u_1), \\
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& reverse(h_0, h, i_0, i, j), \\
& alist(\langle h, i+1, j \rangle, \langle h, i+1, j \rangle[i+1], ?t_1), \\
& alist(\langle h, i+1, j \rangle, h[i+1], ?u'), \\
& i > 0, i \in ?t, ?t = ?t_1 \cup \{i, i+1\}, \{i, i+1\} \otimes ?t_1, \\
& ?t' \otimes ?u', ?t' \cup ?u' = s
\end{aligned} \tag{R.4}$$

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

At this point, we will apply the FLATTEN rule using the certification (which we prove separately below) that:

$$alist(h, j, s), \{i\} \otimes s \models alist(\langle h, i, e \rangle, j, s) \tag{L.1}$$

Flattening the rhs of R.3 results in the following:

$$\begin{aligned}
& reverse(h_0, h, i_0, i, j), alist(h, j, t), alist(h, h[i+1], u_1), \\
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& reverse(h_0, h, i_0, i, j), \\
& alist(h, \langle h, i+1, j \rangle[i+1], ?t_1), \{i+1\} \otimes ?t_1, \\
& alist(h, h[i+1], ?u'), \{i+1\} \otimes ?u', \\
& i > 0, i \in ?t, ?t = ?t_1 \cup \{i, i+1\}, \{i, i+1\} \otimes ?t_1, \\
& ?t' \otimes ?u', ?t' \cup ?u' = s
\end{aligned} \tag{R.3}$$

The algorithm then proves each rhs atoms by matching it with a lhs atom (by the MONOSPLIT rule). Firstly we deal with the three atoms in the rhs, and lastly, deal the constraints in the rhs. Note that the last step here requires only constraint solving.

$$\begin{aligned}
& reverse(h_0, h, i_0, i, j), \quad i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& reverse(h_0, h, i_0, i, j)
\end{aligned} \tag{R.3a}$$

$$\begin{aligned}
& alist(h, j, t), \\
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& alist(h, \langle h, i+1, j \rangle[i+1], ?t_1)
\end{aligned} \tag{R.3b}$$

$$\begin{aligned}
& alist(h, h[i+1], u_1), \\
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& alist(h, h[i+1], ?u')
\end{aligned} \tag{R.3c}$$

$$\begin{aligned}
& i > 0, u = u_1 \cup \{i, i+1\}, \{i, i+1\} \otimes u_1, \\
& t \otimes u, s = t \cup u, i > 0 \models \\
& \{i+1\} \otimes ?t_1, \{i+1\} \otimes ?u', \\
& i > 0, i \in ?t, ?t = ?t_1 \cup \{i, i+1\}, \{i, i+1\} \otimes ?t_1, \\
& ?t' \otimes ?u', ?t' \cup ?u' = s
\end{aligned} \tag{R.3d}$$

Each of the above obligations can be established by reasoning on the constraints. The proof of R.3a does not produce any binding on the existentially-quantified variables in the rhs. The proof of R.3b and R.3c, however produces some bindings that are ultimately passed on for further use.

Certifying $alist$

We finally demonstrate the proof of L.1., thus certifying the predicate $alist$ and justifying the use of flattening above. For this proof, left unfolding is performed on the only atom resulting in two obligations:

$$j = 0, s = \{j\}, \{i\} \otimes s \models alist(\langle h, i, e \rangle, j, s) \tag{L.2a}$$

$$\begin{aligned}
& alist(h, h[j+1], t), s = t \cup \{j\}, t \otimes \{j\}, \{i\} \otimes s \models \\
& alist(\langle h, i, e \rangle, j, s)
\end{aligned} \tag{L.2b}$$

On L.2a the algorithm performs right unfolding resulting in

$$j = 0, s = \{j\}, \{i\} \otimes s \models j = 0, s = \{j\} \tag{L.3}$$

which holds by constraint reasoning. Similarly, the algorithm performs right unfolding on L.2b resulting in the following obligation:

$$\begin{aligned}
& alist(h, h[j+1], t), s = t \cup \{j\}, t \otimes \{j\}, \{i\} \otimes s \models \\
& alist(\langle h, i, e \rangle, \langle h, i, e \rangle[j+1], ?t'), s = ?t' \cup \{j\}, ?t' \otimes \{j\}
\end{aligned} \tag{L.4}$$

On this the algorithm performs induction step using L.1 as hypothesis resulting in the following obligation:

$$\begin{aligned}
& alist(\langle h, i, e \rangle, h[j+1], t), s = t \cup \{j\}, t \otimes \{j\}, \{i\} \otimes s \models \\
& alist(\langle h, i, e \rangle, \langle h, i, e \rangle[j+1], ?t'), s = ?t' \cup \{j\}, ?t' \otimes \{j\}
\end{aligned} \tag{L.5}$$

Since $\{i\} \otimes \{j\}$ is implied by the lhs, we can simplify the rhs expression $\langle h, i, e \rangle[j+1]$ into $h[j+1]$. By also matching the existential variable t' in the rhs with the variable t , we get the following obligation, which can be discharged by constraint reasoning.

$$\begin{aligned}
& alist(\langle h, i, e \rangle, h[j+1], t), s = t \cup \{j\}, t \otimes \{j\}, \{i\} \otimes s \models \\
& alist(\langle h, i, e \rangle, h[j+1], t), s = t \cup \{j\}, t \otimes \{j\}
\end{aligned} \tag{L.6}$$

6. Experiments

We implemented our algorithm using the CLP(\mathcal{R}) programming language [7], and present some results for various example problems in Table 1. We chose a couple of number theoretic functions fib and $idempotent$, as well as a number of classic programs dealing with list resetting, list reverse, array bubblesort and AVL tree rebalancing.

The ‘‘Obligations’’ column displays the total number of proof obligations in the proof tree. The ‘‘Constraint Proof’’ is the number of successful constraint proofs, the ‘‘Induction’’ column displays the number of successful induction application, the ‘‘Left Unfold’’ displays the number of left unfoldings which are not necessarily results in successful proof, and finally, the ‘‘Right Unfold’’ column displays the number of right unfolds attempted not necessarily resulting in successful proofs.

Running times are insignificant, less than a second, and hence not displayed.

We implemented an iterative deepening strategy in order to cover the nondeterministic choices in the algorithm. That is, we specify a depth bound for some of the problems that we experimented with, increasing this bound successively. For the first three problems the proofs are completed by induction, hence it is not necessary to specify a depth bound. The last three proofs are completed by a number of unfoldings. We note that the ‘‘Reverse’’ row reports the result for the proof obligation R.1 in Section 5.5. The ‘‘Bubble Sort’’ row displays the results for the proof of obligation S.1. Finally, the ‘‘AVL Tree’’ considers the standard program (not listed here) and a proof of rebalancing.

We believe these experiments would extrapolate to many larger examples because the nature of the proof obligations, for example

Problem	Depth Bound	Obligations	Constraint Proof	Induction	Left Unfold	Right Unfold
Fibonacci	∞	8	5	2	1	0
Idempotent	∞	25	5	1	5	12
List Reset	∞	17	1	1	2	6
Reverse	2	23	2	0	4	13
Bubble Sort	2	11	5	0	3	2
AVL Tree	1	13	4	0	0	2

Table 1. Experimental Results

the reversing and marking operations in the Schor-Waite program on graphs, have essentially a similar structure to our experimental programs.

7. Conclusion

We presented a specification language which describes memory explicitly as an array, and describes pointers and data elements as integers. In conjunction with user-defined recursive definitions, the language provides for the explicit modelling of data structures. Furthermore, with the inclusion of set expressions, we showed how the language expressed “footprints” of both user-defined predicates and program fragments, thus proving the ability to specify separation properties. Verification conditions can then be automatically generated from a program annotated with specifications.

The second part of the paper presented an algorithm for proving verification conditions. The distinctive feature of the algorithm is the ability to generate and then use induction hypotheses. Induction then allowed us to *certify* predicates so that their multiset arguments are faithful representations of the predicate footprint. We then used a monocus rule as a main heuristic a cut rule to provides a basis for a practical implementation. Finally, we demonstrated the algorithm on a collection of classic problems.

References

- [1] J. Berdine, C. Calcagno, and P. W. O’Hearn. Using Datalog with binary decision diagrams for program analysis. In K. Yi, editor, *3rd APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
- [2] S. A. Cook. Soundness and completeness of axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
- [3] S. Craciunescu. Proving equivalence of CLP programs. In *18th ICLP*, volume 2401 of *LNCS*. Springer, 2002.
- [4] 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.
- [5] L. Fribourg. Automatic generation of simplification lemmas for inductive proofs. In *ISLP 1991*, pages 103–116. MIT Press, 1991.
- [6] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
- [7] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(\mathcal{R}) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
- [8] J. Jaffar, A. E. Santosa, and R. Voicu. A coinduction rule for entailment of recursively defined properties. In P. J. Stuckey, editor, *14th CP*, volume 5202 of *LNCS*, pages 493–508. Springer, 2008.
- [9] B. Jeannot, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *11th SAS*, volume 3148 of *LNCS*, pages 246–264. Springer, 2004.
- [10] T. Kanamori and H. Fujita. Formulation of induction formulas in verification of Prolog programs. In *8th CADE*, volume 230 of *LNCS*, pages 281–299. Springer, 1986.
- [11] 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.
- [12] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.
- [13] N. Klarlund and M. I. Schwartzbach. Graph types. In *20th POPL*, pages 196–205. ACM Press, 1993.
- [14] P. Maier. Deciding extensions of the theories of vectors and bags. In *VMCAI*, pages 245–259, 2009.
- [15] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *IFIP Congress 1962*. North-Holland, 1983.
- [16] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *17th CAV*, volume 3576 of *LNCS*, pages 476–490. Springer, 2005.
- [17] 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.
- [18] 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).
- [19] 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.
- [20] J. C. Reynolds. Separation logic: A logic for shared mutable data objects. In *17th LICS*, pages 55–74. IEEE Computer Society Press, 2002.
- [21] 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.
- [22] 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.
- [23] A. Roychoudhury, K. N. Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM TOPLAS*, 26(3):464–509, 2004.
- [24] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, May 2002.
- [25] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In J. L. Fiadeiro and P. Inverardi, editors, *FASE 2008*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.