Diagrammatic Reasoning in Separation Logic

M. Ridsdale[1], M. Jamnik[1], N. Benton[2], and J. Berdine[2]

[1] University of Cambridge
[2] Microsoft Research Cambridge

Abstract. A new method of reasoning about simple imperative programs in separation logic is proposed. Rather than proving program specifications symbolically, the hope is to model more closely human diagrammatic reasoning, and to perform automated diagrammatic reasoning in separation logic.

1 Introduction

Separation logic is used for reasoning about low-level imperative programs that manipulate pointer data structures. It enables the writing of concise proofs of correctness of the specifications of simple programs, and such proofs have been successfully automated.

When reasoning informally about separation logic, it is often useful to draw diagrams representing program states, with memory locations represented by boxes, pointers represented by arrows, etc. We aim to formalise these diagrams and implement an automated theorem prover (ATP) which makes use of this formalism. This proposal outlines a promising direction for research. The ideas on diagrammatic reasoning are drawn from [1], which is on diagrammatic theorem proving in arithmetic; we also draw on ideas from [2], which is on symbolic automated theorem proving in separation logic. An example of the kind of problem we want to be able to solve is shown in Fig. 1.

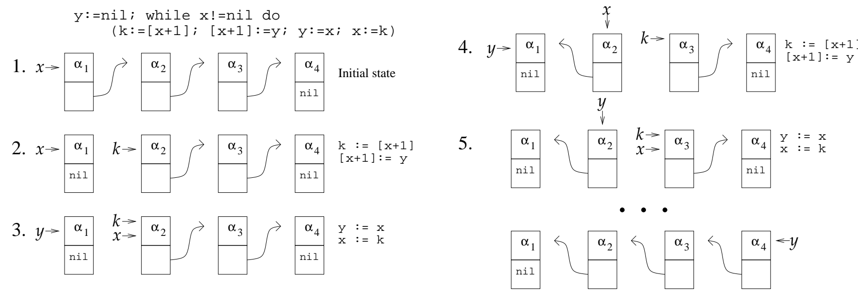

Fig. 1. An informal diagrammatic proof, that the program shown reverses a list

Each square represents a memory location, which can hold values (α_i) or pointers to other memory locations. x, y, k outside of the squares represent program variables, which store memory locations. The program shown reverses a linked list; this can be proved symbolically using separation logic. An informal diagrammatic proof is given in Fig. 1. It shows a partial execution trace of the list-reversal program, through two iterations of the `while` loop. On each iteration, the direction of one pointer in the list is reversed. A person following this trace can see intuitively that the entire list will be reversed when the program is complete.

Following [1], we intend to turn this into a formal proof using an ATP which makes use of *schematic proofs*. This approach allows us to avoid including abstractions such as ellipses in diagrams, and doing inductive proofs over diagrams. Informally, schematic proofs are intended to capture the notion of a 'general proof': they are functions from some set of parameters to a set of proofs, for some more specific notion of a proof. In the example in Fig. 1, the schematic proof *sch-pf* would be a function of the length of the list, such that *sch-pf* (n) is a proof that lists specifically of length n are reversed by the program.

A disadvantage of this approach is that it requires at least one example proof from the user in every case. Possible future work would be to look at an alternative diagrammatic procedure which is fully automated.

In order to implement the above, we first need to define the following:

1. Diagram syntax: the arrangements of shapes on a page or screen which constitute a well-formed diagram.
2. Diagram semantics: a function mapping each diagram to a set of memory states represented by the diagram.
3. Diagram operations: a set of operations on diagrams that will permit us to perform automated reasoning. These operations must be sound with respect to the semantics, and preferably complete, particularly since the symbolic system in [2] is complete.
4. Theorems and proofs: a schematic proof will output a concrete proof for each value of parameter n. A definition is required of what would constitute such a concrete proof.

Once the appropriate definitions are in place, the ATP would work in the following stages:

1. User provides a few example proofs for a specific instance of a theorem.
2. Program generalises from the examples, using a heuristic which suggests a schematic proof.
3. Program verifies the schematic proof using a sound verification procedure.

2 Syntax and Semantics

We can define diagrammatic objects corresponding to predicates in separation logic. These predicates represent data structures and statements about data structures. For example, the two diagrams in Fig. 2 represent memory states in which the predicates `list` and `list segment` hold. The `list` is terminated by `nil`, while the `list segment` is terminated by a program variable.

Fig. 2. Diagrams representing a list segment (left) and a list (right)

3 Theorems and Proofs

Before being able to reason about programs using the schematic proofs mentioned previously, we need to be able to reason about static memory states, computing whether two diagrams are equivalent, or whether one diagram entails

another, more general one. For example, a single list might be drawn as the concatenation of any possible decomposition into sublists, and we need a way to automatically recognise these equivalences. An example from [2] is shown in Fig. 3. There is a `list segment` from x to t, and a separate `list` beginning at y. The two are connected by a single element, indexed by t. It follows that there is in fact a `list` from x terminated by `nil`.

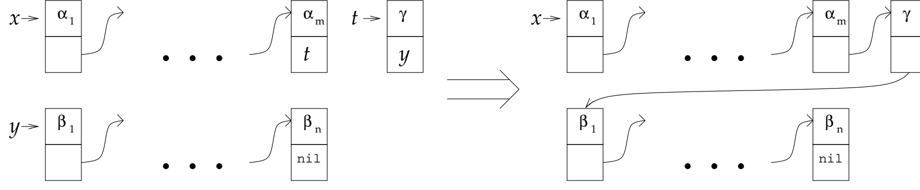

Fig. 3. A *list segment* and a *list*, which are implicitly connected.

In order to perform automated reasoning, we define a set of operations on diagrams, which will be picked to allow for sound reasoning with respect to the semantics. These can be of two types: static operations which preserve the diagrams' meaning, for reasoning about static memory states, and dynamic operations corresponding to program commands. An example of the former is shown in Fig 3. It constitutes a simple diagrammatic proof of the problem described in the previous paragraph: two instances of a `replacement` operation. This operation simply replaces both instances of program variables y and t with pointers between the relevant cells. For comparison, here is the symbolic proof that the procedure in [2] would give:

$$\cfrac{\cfrac{\cfrac{\cfrac{t \neq \mathsf{nil} \mid \mathsf{ls}(y, \mathsf{nil}) \ \vdash \ \mathsf{ls}(y, \mathsf{nil})}{t \neq \mathsf{nil} \mid t \mapsto [n\!:\!y] \ * \ \mathsf{ls}(y, \mathsf{nil}) \ \vdash \ t \mapsto [n\!:\!y] \ * \ \mathsf{ls}(y, \mathsf{nil})}}{t \neq \mathsf{nil} \mid t \mapsto [n\!:\!y] \ * \ \mathsf{ls}(y, \mathsf{nil}) \ \vdash \ \mathsf{ls}(t, \mathsf{nil})}}{t \neq \mathsf{nil} \mid \mathsf{ls}(x, t) \ * \ t \mapsto [n\!:\!y] \ * \ \mathsf{ls}(y, \mathsf{nil}) \ \vdash \ \mathsf{ls}(x, \mathsf{nil})}}{\mathsf{ls}(x, t) \ * \ t \mapsto [n\!:\!y] \ * \ \mathsf{ls}(y, \mathsf{nil}) \ \vdash \ \mathsf{ls}(x, \mathsf{nil})}$$

We believe the diagrammatic proof is more human-readable.

4 Conclusion

Developers frequently use diagrams informally when discussing separation logic problems with one another. Our aim is to formalise these diagrams and implement an automated diagrammatic theorem prover for separation logic. A necessary first step to creating an ATP which can reason about entire programs is an ATP which can reason about static memory states, and this is our initial direction of research.

References

1. Jamnik, M., Bundy, A., Green, I.: On automating diagrammatic proofs of arithmetic arguments. Journal of Logic, Language and Information **8(3)** (1999) 297–321
2. Berdine, J., Calcagno, C., O'Hearn, P.: Symbolic execution with separation logic. APLAS (2005)