Diagrammatic Reasoning for Software Verification

Matt Ridsdale, Mateja Jamnik*
*University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD, U.K.
mer39@cam.ac.uk, mj201@cl.cam.ac.uk

Nick Benton, Josh Berdine†
†Microsoft Research Cambridge
Roger Needham Building
7 JJ Thomson Avenue
Cambridge CB3 0FB, U.K.
nick@microsoft.com, jjb@microsoft.com

1 Introduction

Diagrams have historically been seen as informal aids to understanding, rather than rigorous mathematical tools. However they are widely used in mathematics teaching and informal communication. Recent work has challenged the historical view, with formal diagrammatic reasoning systems implemented for geometry, arithmetic, and other areas of mathematics. (Mumma, 2009; Jamnik, 2001; Barker-Plummer and Bailin, 1997).

This work is about formalising diagrams for program verification. Program verification using symbolic logic is a well-developed field, but anecdotes suggest that diagrams are widely used by researchers in informal reasoning and communication about the subject. Typically, a "boxes and arrows" diagram is used to indicate the input state to the program, and modifications to this diagram trace the program's execution. Humans can informally "verify" an algorithm this way by satisfying themselves that it works correctly for some typical input. We therefore specify a formal semantics of box-and-arrow diagrams and define operations corresponding to the program commands of our language.

1.1 Why diagrams?

As compared to symbolic logic, diagrammatic languages can be seen as domain-specific, while symbolic logics such as predicate logic are generic. This specificity enables domain knowledge to be encoded in the representation, which allows diagrams to concisely capture information which would require a verbose symbolic description, and can support inference by decreasing the proof search space. Diagrams offer other advantages too: diagrammatic proofs can be easier for humans to understand, and they can support the inference of general conclusions by reasoning about specific examples.

Our system will be used to investigate whether the above advantages can be exploited in this problem domain.

2 Reasoning With Diagrams

We intend to make use of diagrams in at least three aspects of program verification: writing the program specifications, deciding entailment problems between program states, and reasoning about programs. We briefly describe an approach to each of these.

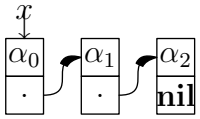

Figure 1: Examples of what our diagrams look like diagrams. This diagram represents a linked list.

Fig. 1 shows some examples of the kinds of diagrams we are using. We define a syntax and semantics of diagrams such that each square represents a memory location, which can hold values (α_i or nil) or pointers to other memory locations. Adjacent squares represent adjacent cells in memory, and the roman letters outside of the squares represent program variables, which store memory locations or values. Thus each diagram represents a set of memory states.

The semantics of programs can be specified as a partial function from sets of program states to sets of program states, as in the well-known notion of *Hoare triples*. The Hoare triple $\{P\}$ prog $\{Q\}$ asserts that if prog is run in a state in which the predicate P holds, and if prog terminates, then the predicate Q holds in the resulting state. In our system, P and Q will be replaced with diagrams describing sets of states, enabling us to write program specifications using diagrams.

In software verification, there are two levels of inference that can be performed: entailments between static program states, and reasoning about the effects of program commands. For example in the Hoare triple

$$\{x = 4 \wedge y = x\}\, z := y\, \{x = 4 \wedge y = x \wedge z = 4\}$$

we must first deduce that $(x = 4 \wedge y = x \Rightarrow y = 4)$ before we can deduce that the triple holds. This can be accomplished with diagrams by defining operations which strengthen, weaken or preserve a diagram's meaning. An example was given in Ridsdale et al. (2008) of a diagrammatic proof of entailments between program states (Fig. 2). Fig. 2 shows a pair of lists which are implicitly connected, as the value y at the end of the first list equals the address of the head of the second list. A diagrammatic proof that the lists are connected need consist of only a single operation, replacing both instances of y with a pointer. We argue this is more intuitive than the corresponding symbolic proof in separation logic.

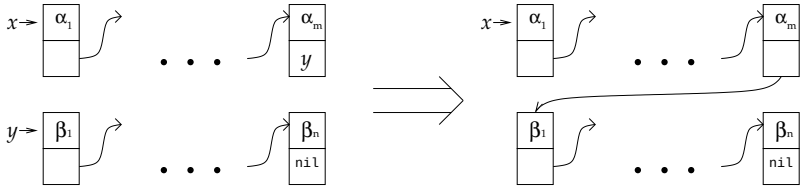

Figure 2: A *list segment* and a *list*, which are implicitly connected.

3 Reasoning About Programs

Our approach to reasoning about programs is based on Jamnik (2001), on diagrammatic proofs in arithmetic, in which a diagrammatic proof consists of a program which acts on diagrams. We have a set of diagrammatic operations corresponding to commands in our programming language, and use those to model the program to be verified with a program on diagrams. The hope is that a way can be found to do this such that the resulting diagrammatic program is easier to verify than the original program, but we have not yet been able to determine that this will be the case.

References

Dave Barker-Plummer and Sidney C. Bailin. The role of diagrams in mathematical proofs. *Machine Graphics and Vision*, 6(1):25–56, 1997.

Mateja Jamnik. *Mathematical Reasoning with Diagrams: From Intuition to Automation*. CSLI Press, Stanford, CA, USA, 2001.

John Mumma. Proofs, pictures, and euclid. http://www.contrib.andrew.cmu.edu/ jmumma/list.html. *Synthese (To appear)*, 2009. URL `http://www.contrib.andrew.cmu.edu/ jmumma/list.html`.

Matt Ridsdale, Mateja Jamnik, Nick Benton, and Josh Berdine. Diagrammatic reasoning in separation logic. In *Diagrammatic Representation and Inference, 5th International Conference, Herrsching, Germany. Proceedings*, volume 5223 of *Lecture Notes in Computer Science*, pages 408–411. Springer, 2008.