Diagrammatic Reasoning in Separation Logic

Matt Ridsdale[1] Mateja Jamnik[1] Nick Benton[2] Josh Berdine[2]

[1]Computer Laboratory, University of Cambridge

[2]Microsoft Research Cambridge

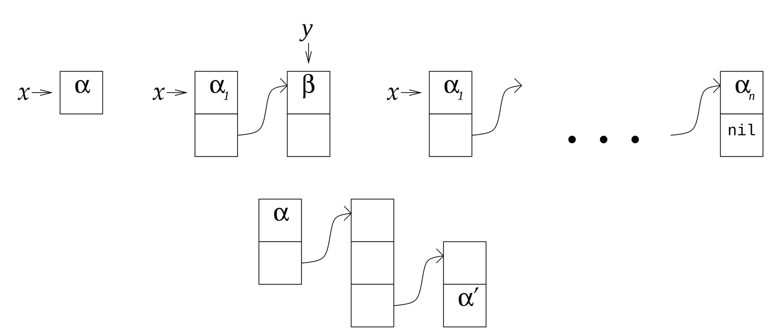
UNIVERSITY OF CAMBRIDGE

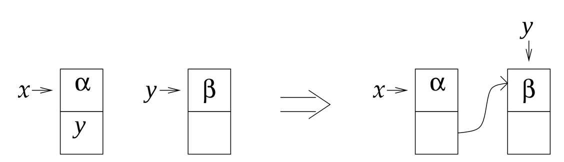
1. Diagrammatic Reasoning

- The majority of work in automated theorem proving is based on symbolic logic.
- Diagrams are seen not as rigorous mathematical tools, but as informal aids to understanding.
- Aim:
  - Formalise a diagrammatic system for a particular problem domain (e.g. program verification using separation logic).
  - Implement an automated theorem prover making use of this formalism.

2. Separation Logic

- Logic for verifying low-level imperative programs.
- Proofs consist of lists of Hoare triples (annotated program statements: see box 3).
- Diagrams are used informally. Boxes represent memory cells; they may contain values and have pointers to other boxes. Program variables are drawn pointing to the corresponding memory cell.
- Operations can re-draw pointers, overwrite values in boxes, etc. Figure on the right shows a *make_pointers_explicit* operation.

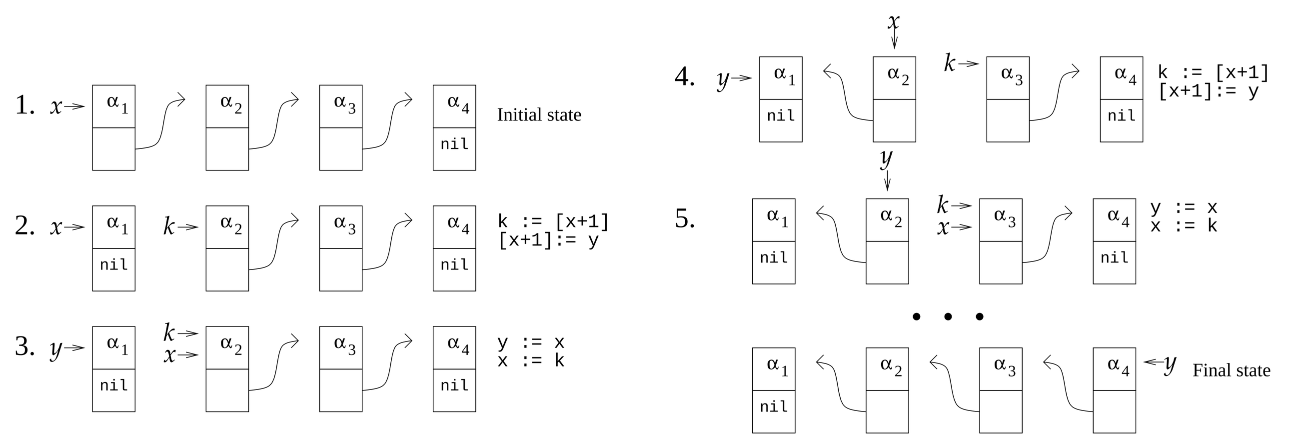




3. Diagrammatic vs Separation Logic Proof

Diagrammatic

```
y:=nil; while x!=nil do
   (k:=[x+1]; [x+1]:=y; y:=x; x:=k)
```



Symbolic

$\{\text{list } \alpha_0 \text{ i}\}$
$\{\text{list } \alpha_0 \text{ i} * (\text{emp} \wedge \text{nil} = \text{nil})\}$
j := nil;
$\{\text{list } \alpha_0 \text{ i} * (\text{emp} \wedge \text{j} = \text{nil})\}$
$\{\text{list } \alpha_0 \text{ i} * \text{list } \epsilon \text{ j}\}$
$\{\exists \alpha, \beta.\ (\text{list } \alpha \text{ i} * \text{list } \beta \text{ j} \wedge \alpha_0^\dagger = \alpha^\dagger . \beta)\}$
while i $\neq$ nil **do**
  $\{\exists \text{a}, \alpha, \beta.\ (\text{list } (\text{a}\cdot\alpha) \text{ i} * \text{list } \beta \text{ j}) \wedge \alpha_0^\dagger = (\text{a} \cdot \alpha)^\dagger \cdot \beta\}$
  $\{\exists \text{a}, \alpha, \beta, \text{k}.\ (\text{i} \mapsto \text{a,k} * \text{list } \alpha \text{ k} * \text{list } \beta \text{ j})$
  $\wedge\ \alpha_0^\dagger = (\text{a} \cdot \alpha)^\dagger \cdot \beta\}$
  k := [i + 1];
  $\{\exists \text{a}, \alpha, \beta.\ (\text{i} \mapsto \text{a,k} * \text{list } \alpha \text{ k} * \text{list } \beta \text{ j})$
  $\wedge\ \alpha_0^\dagger = (\text{a} \cdot \alpha)^\dagger \cdot \beta\}$
  [i + 1] := j;
  $\{\exists \text{a}, \alpha, \beta.\ (\text{i} \mapsto \text{a,j} * \text{list } \alpha \text{ k} * \text{list } \beta \text{ j})$
  $\wedge\ \alpha_0^\dagger = (\text{a} \cdot \alpha)^\dagger \cdot \beta\}$
  $\{\exists \text{a}, \alpha, \beta.\ (\text{list } \alpha \text{ k} * \text{list } (\text{a}\cdot\beta) \text{ i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \text{a} \cdot \beta\}$
  $\{\exists \alpha, \beta.\ (\text{list } \alpha \text{ k} * \text{list } \beta \text{ i}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}$
  j := i; i := k;
  $\{\exists \alpha, \beta.\ (\text{list } \alpha \text{ i} * \text{list } \beta \text{ j}) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta\}$
$\{\exists \alpha, \beta.\ \text{list } \beta \text{ j} \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta \wedge \alpha = \epsilon\}$
$\{\text{list } \alpha_0^\dagger \text{ j}\}$

- By tracing execution of program for a couple of iterations of the while loop, a human can see that the program reverses a linked list.
- Aim to make a formal system of syntax, semantics, operations and inference rules modelling this kind of reasoning.
- Aim to generalise specific proofs like the one above (which is about lists of length 4 only) using *schematic proofs*.

4. Syntax and Semantics

- Can be formally defined for diagrams, just as for symbolic sentences.
- Syntax specifies shapes that can appear in diagrams and the spatial relations which are allowed.
- Semantics given by an interpretive function mapping diagrams to sets of program states.
- Operations: draw or erase operations for pointers, program variables and values.

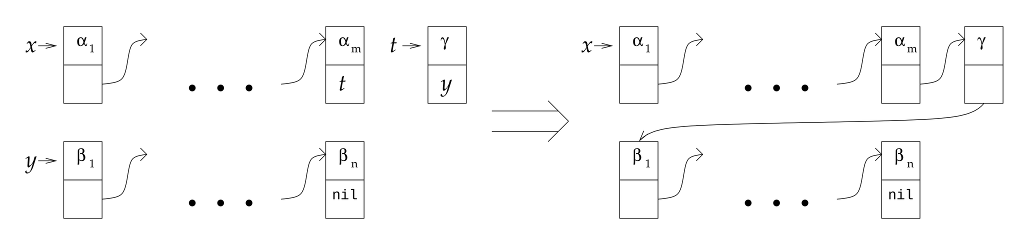
5. Schematic Proofs

- Formalised notion of a general proof derived from specific instances.
- A schematic proof is a program for generating a specific proof for any given problem instance.
- Relevance: diagrams are a way of using the concrete to reason about the general.
- A schematic proof of the theorem in box 3:

sch-pf($\boldsymbol{d_1}, \boldsymbol{d_2}$):
(recursive function on pairs of diagrams. $\boldsymbol{d_1}$ shows a right-to-left list; on its right is $\boldsymbol{d_2}$, showing a left-to-right list. See slide 3)
1: *move_var*($\boldsymbol{k}$, *head*(*tail*($\boldsymbol{d_2}$)))
2: *erase_val*(*head*($\boldsymbol{d_2}$))
3: *draw_pointer*(*head*($\boldsymbol{d_2}$), *last_element*($\boldsymbol{d_1}$))
4: *move_var*($\boldsymbol{y}$, *head*($\boldsymbol{d_2}$))
5: *move_var*($\boldsymbol{x}$, *head*(*tail*($\boldsymbol{d_2}$)))
6: *sch-pf*([$\boldsymbol{d_1}$, *head*($\boldsymbol{d_2}$)], *tail*($\boldsymbol{d_2}$)).

6. Reasoning About Static Program States

- Initially we are investigating how to reason about static program states. This kind of reasoning is necessary at intermediate stages of making proofs about programs.
- Example below: the left-hand diagram entails a nil-terminated list beginning at x.
- The diagrammatic proof proceeds by application of a single operation, *make_pointers_explicit*, 2 times. The symbolic proof is shown on the right.
- The simplicity comes from the similar structure of the problem domain and the diagrammatic system.



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

7. Conclusions and Future Work

- Diagrammatic logic can be formalised, and automated reasoning performed, just as for traditional symbolic logic.
- Diagrammatic proofs in separation logic appear to be more human-readable and “natural” than the corresponding separation logic proofs.
- Diagrammatic reasoning systems are highly tailored to specific problem domains. Future work will look at further case studies and investigate general principles of diagrammatic reasoning.