Ribbon Proofs for Separation Logic

John Wickerson
University of Cambridge
john.wickerson@cl.cam.ac.uk

Mike Dodds
University of Cambridge
mike.dodds@cl.cam.ac.uk

Matthew Parkinson
Microsoft Research Cambridge
mattpark@microsoft.com

A program proof should not merely certify *that* a program is correct; it should explain *why* it is correct. A proof should be more than 'true': it should be informative, and it should be intelligible. Extending work by Bean [1], we introduce a system that produces readable program proofs that are highly scalable and easily modified.

The de facto standard for presenting program proofs in Hoare logic [2] is the *proof outline*, in which the program's instructions are interspersed with 'enough' assertions to allow the reader to reconstruct the derivation tree. As an example, Fig. 1a presents a proof outline for a program that performs in-place list reversal. A key asset of the proof outline is what we shall call *instruction locality*: that one can verify each instruction in isolation (by confirming that the assertions immediately above and below it form a valid Hoare triple) and immediately deduce that the entire proof is correct.

The proof outline suffers several drawbacks, however. First, there is much repetition: '$\mathit{list}\,\alpha\,\mathrm{x}$' appears redundantly in six consecutive assertions before it is used on line 25. Second, there is no distinction between those parts of an assertion that are affected by an instruction and those that are merely in what separation logic calls the *frame*. For instance, line 19 affects only the second and fourth conjuncts of its preceding assertion, but it is difficult to deduce its effect because two unchanged conjuncts are interspersed. (Had we followed common practice and reduced the size of the proof outline by combining this line with the assignment on line 17, the effect would be even harder to deduce.) Third, the use of logical variables is unclear. For instance, spotting that the β in line 20 differs from the one in line 22 requires careful examination, or else, as we have done, an explicit textual comment. These minor problems in our illustrative example quickly become devastating when scaled to large programs.

Separation logic [3], [4] provides a mechanism for handling a second dimension of locality: *resource locality*. One can use separation logic's *small axioms* to reason about an instruction operating only on the resources (i.e. memory cells) that it needs, and immediately deduce its effect on the entire state using the *frame rule*. To depict this mechanism in a proof outline, one must show applications of the frame rule explicitly. But this is tedious; moreover, it is difficult to know when and what to frame. Meanwhile, the ribbon proof inherently supports resource locality. Its primitive steps correspond exactly to the small axioms. It is thus an ideal representation for exploiting both forms of locality that separation logic provides.

Figure 1b recasts our proof as a ribbon proof. The state is distributed across several *ribbons* (thick borders). Horizontally separated ribbons describe disjoint parts of the state. The instructions are in grey bars, and the scope of each logical variable is delimited by an *existential box* (thin borders). We are free to stretch ribbons as required by the layout, and, because $*$ is commutative, we can 'twist' them too. A temporarily inactive ribbon slides discreetly down the side of the proof. This effect is achieved by invoking the frame rule at each instruction; but crucially in a ribbon proof, these invocations are implicit and do not increase the diagram's complexity. Observe that the repetition has disappeared, and that each instruction's effect is clear: it affects exactly those assertions directly above and below it, while framed assertions (which must not mention variables written by the instruction) bypass to the left or right. Existential boxes extend vertically to indicate the range of steps over which the same witness is used, thus making the usage of logical variables visually clear.

In our full paper [5]:

- we present an Isabelle-checked graph-based formalisation of our proof system;
- we showcase, with a ribbon proof of the memory manager from Version 7 Unix, the ability of our diagrams to present readable proofs of large, complex programs; and
- we describe a prototype tool for mechanically checking ribbon proofs in Isabelle. Provided with a small proof script for each primitive step, our tool assembles a script that verifies the entire diagram. The tool handles tediums such as the associativity and commutativity of $*$ automatically, leaving the user to concentrate on the interesting parts of the proof.

This work lays the foundations for a new way to use logic to understand programs. Where a proof outline essentially flattens a proof to a list of assertions, our system produces geometric objects that illuminate the structure of proofs, and which can be navigated, modified and simplified by leveraging human visual intuition.

References

[1] J. Bean, "Ribbon proofs," in *MFPS*, 2003.

[2] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, October 1969.

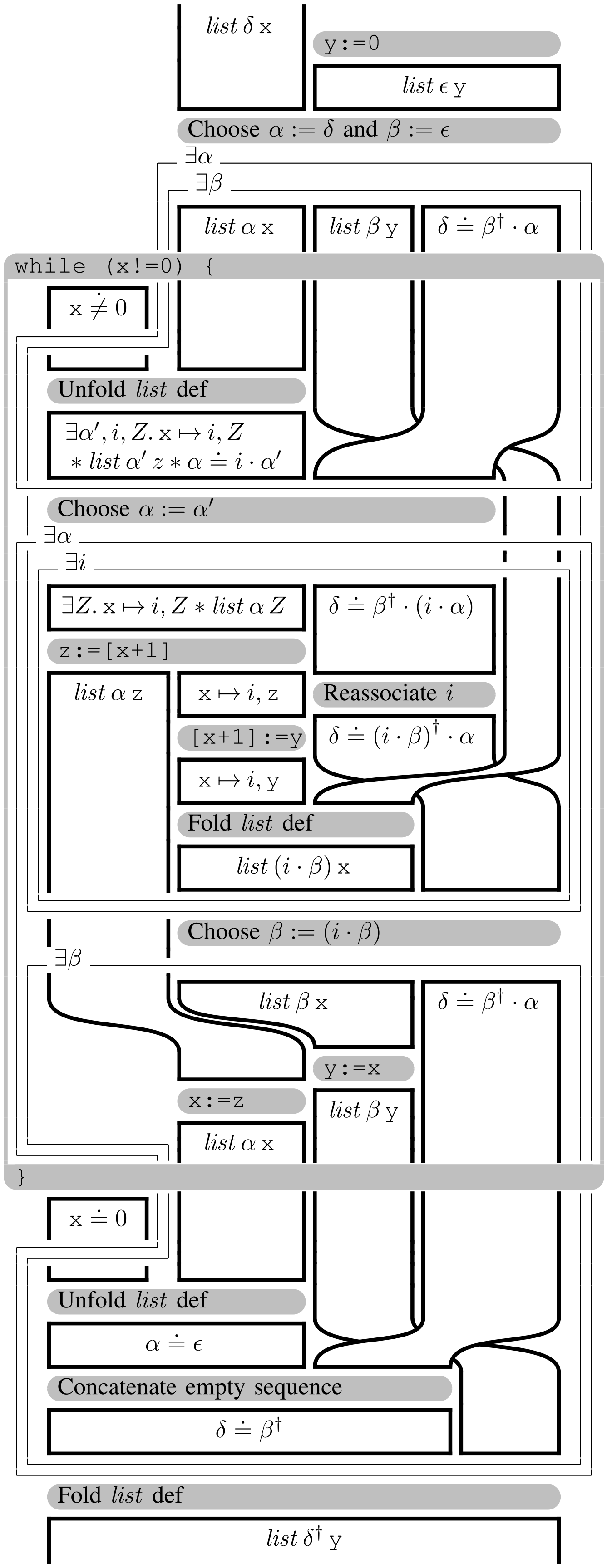
[3] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*, 2002.

[4] S. Ishtiaq and P. W. O'Hearn, "BI as an assertion language for mutable data structures," in *POPL*, 2001.

[5] J. Wickerson, M. Dodds, and M. J. Parkinson, "Ribbon proofs for separation logic," May 2012, http://www.cl.cam.ac.uk/~jpw48/ribbons.html.

```
$\{ \mathit{list}\,\delta\,\mathrm{x} \}$
y:=0;
$\{ \mathit{list}\,\delta\,\mathrm{x} * \mathit{list}\,\epsilon\,\mathrm{y} \}$
// Choose $\alpha := \delta$ and $\beta := \epsilon$
while $\{ \exists\alpha,\beta.\, \mathit{list}\,\alpha\,\mathrm{x} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \}$
(x!=0) {
  $\{ \mathrm{x} \mathrel{\dot{\neq}} 0 * (\exists\alpha,\beta.\, \mathit{list}\,\alpha\,\mathrm{x} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha) \}$
  $\{ \exists\alpha,\beta.\, \mathrm{x} \mathrel{\dot{\neq}} 0 * \mathit{list}\,\alpha\,\mathrm{x} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \}$
  // Unfold list def
  $\left\{ \begin{array}{l} \exists\alpha,\beta.\, (\exists\alpha',i,Z.\, \mathrm{x} \mapsto i,Z * \mathit{list}\,\alpha'\,\mathrm{z} * \alpha \doteq i \cdot \alpha') \\ * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \end{array} \right\}$
  // Choose $\alpha := \alpha'$
  $\left\{ \begin{array}{l} \exists\alpha,\beta,i,Z.\, \mathrm{x} \mapsto i,Z * \mathit{list}\,\alpha\,Z * \delta \doteq \beta^\dagger \cdot (i \cdot \alpha) \\ * \mathit{list}\,\beta\,\mathrm{y} \end{array} \right\}$
  z:=[x+1];
  $\{ \exists\alpha,\beta,i.\, \mathit{list}\,\alpha\,\mathrm{z} * \mathrm{x} \mapsto i,\mathrm{z} * \delta \doteq \beta^\dagger \cdot (i \cdot \alpha) * \mathit{list}\,\beta\,\mathrm{y} \}$
  // Reassociate $i$
  $\{ \exists\alpha,\beta,i.\, \mathit{list}\,\alpha\,\mathrm{z} * \mathrm{x} \mapsto i,\mathrm{z} * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha * \mathit{list}\,\beta\,\mathrm{y} \}$
  [x+1]:=y;
  $\{ \exists\alpha,\beta,i.\, \mathit{list}\,\alpha\,\mathrm{z} * \mathrm{x} \mapsto i,\mathrm{y} * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha * \mathit{list}\,\beta\,\mathrm{y} \}$
  // Fold list def
  $\{ \exists\alpha,\beta,i.\, \mathit{list}\,\alpha\,\mathrm{z} * \mathit{list}\,(i \cdot \beta)\,\mathrm{x} * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha \}$
  // Choose $\beta := (i \cdot \beta)$
  $\{ \exists\alpha,\beta.\, \mathit{list}\,\alpha\,\mathrm{z} * \mathit{list}\,\beta\,\mathrm{x} * \delta \doteq \beta^\dagger \cdot \alpha \}$
  y:=x;
  $\{ \exists\alpha,\beta.\, \mathit{list}\,\alpha\,\mathrm{z} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \}$
  x:=z;
  $\{ \exists\alpha,\beta.\, \mathit{list}\,\alpha\,\mathrm{x} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \}$
}
$\{ \mathrm{x} \doteq 0 * (\exists\alpha,\beta.\, \mathit{list}\,\alpha\,\mathrm{x} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha) \}$
$\{ \exists\alpha,\beta.\, \mathrm{x} \doteq 0 * \mathit{list}\,\alpha\,\mathrm{x} * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \}$
// Unfold list def
$\{ \exists\alpha,\beta.\, \alpha \doteq \epsilon * \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \cdot \alpha \}$
// Concatenate empty sequence
$\{ \exists\beta.\, \mathit{list}\,\beta\,\mathrm{y} * \delta \doteq \beta^\dagger \}$
// Fold list def
$\{ \mathit{list}\,\delta^\dagger\,\mathrm{y} \}$
```

(a) A proof outline



$\mathit{list}\,\delta\,\mathrm{x}$
y:=0
$\mathit{list}\,\epsilon\,\mathrm{y}$
Choose $\alpha := \delta$ and $\beta := \epsilon$
$\exists\alpha$
$\exists\beta$
$\mathit{list}\,\alpha\,\mathrm{x}$
$\mathit{list}\,\beta\,\mathrm{y}$
$\delta \doteq \beta^\dagger \cdot \alpha$
while (x!=0) {
$\mathrm{x} \mathrel{\dot{\neq}} 0$
Unfold list def
$\exists\alpha',i,Z.\, \mathrm{x} \mapsto i,Z$
$* \mathit{list}\,\alpha'\,z * \alpha \doteq i \cdot \alpha'$
Choose $\alpha := \alpha'$
$\exists\alpha$
$\exists i$
$\exists Z.\, \mathrm{x} \mapsto i, Z * \mathit{list}\,\alpha\,Z$
$\delta \doteq \beta^\dagger \cdot (i \cdot \alpha)$
z:=[x+1]
$\mathit{list}\,\alpha\,\mathrm{z}$
$\mathrm{x} \mapsto i, \mathrm{z}$
Reassociate i
[x+1]:=y
$\delta \doteq (i \cdot \beta)^\dagger \cdot \alpha$
$\mathrm{x} \mapsto i, \mathrm{y}$
Fold list def
$\mathit{list}\,(i \cdot \beta)\,\mathrm{x}$
Choose $\beta := (i \cdot \beta)$
$\exists\beta$
$\mathit{list}\,\beta\,\mathrm{x}$
$\delta \doteq \beta^\dagger \cdot \alpha$
y:=x
x:=z
$\mathit{list}\,\beta\,\mathrm{y}$
$\mathit{list}\,\alpha\,\mathrm{x}$
}
$\mathrm{x} \doteq 0$
Unfold list def
$\alpha \doteq \epsilon$
Concatenate empty sequence
$\delta \doteq \beta^\dagger$
Fold list def
$\mathit{list}\,\delta^\dagger\,\mathrm{y}$

(b) A ribbon proof

Fig. 1. Two proofs of list reverse. For a binary relation r, we write $x\ \dot{r}\ y$ for $x\ r\ y \wedge emp$. We write $\cdot$ for sequence concatenation, $(-)^\dagger$ for sequence reversal and ϵ for the empty sequence, and define *list* as the smallest predicate satisfying $\mathit{list}\,\alpha\,x \Leftrightarrow x \doteq 0 * \alpha \doteq \epsilon \vee x \mathrel{\dot{\neq}} 0 * \exists\alpha',i,x'.\, x \mapsto i, x' * \alpha \doteq i \cdot \alpha' * \mathit{list}\,\alpha'\,x'$.