Ribbon Proofs for Separation Logic

John Wickerson[1], Mike Dodds[2], and Matthew Parkinson[3]

[1] Technische Universität Berlin, Germany
[2] University of York, United Kingdom
[3] Microsoft Research Cambridge, United Kingdom

Abstract. We present *ribbon proofs*, a diagrammatic system for proving program correctness based on separation logic. Ribbon proofs emphasise the structure of a proof, so are intelligible and pedagogical. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they may be more scalable. Where proof outlines are cumbersome to modify, ribbon proofs can be visually manoeuvred to yield proofs of variant programs. This paper introduces the ribbon proof system, proves its soundness and completeness, and outlines a prototype tool for validating the diagrams in *Isabelle*.

1 Introduction

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. This paper does not contribute new methods for proving more properties of more programs, but rather, a new way to present such proofs. Building on work by Bean [2], we present a system that produces program proofs in separation logic that are readable, scalable, and easily modified.

A program proof in Hoare logic [15] is usually presented as a *proof outline*, in which the program's instructions are interspersed with 'enough' assertions to allow the reader to reconstruct the derivation tree. Since emerging circa 1971, the proof outline has become the de facto standard in the literature on both Hoare logic (e.g. [1, 16, 25, 28]) and its recent descendant, separation logic (e.g. [3, 8–11, 14, 17, 18, 20, 27, 31]). Its great triumph is what might be called *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.

Yet proof outlines also suffer several shortcomings, some of which are manifested in Fig. 1a. This proof outline concerns a program that writes to three memory cells, which separation logic's $*$-operator deems distinct. First, it is highly repetitive: '$x \mapsto 1$' appears three times. Second, it is difficult to interpret the effect of each instruction, there being no distinction between those parts of an assertion that are actively involved and those that are merely in what separation logic calls the *frame*. For instance, line 4 affects only the second conjunct of its preceding assertion, but it is difficult to deduce the assignment's effect because two unchanged conjuncts are also present. Of course, these are only minor problems in our toy example, but they quickly become devastating when scaled to serious programs.
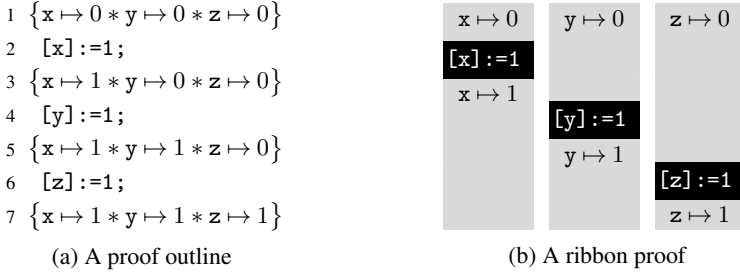
(a) A proof outline   (b) A ribbon proof

Fig. 1. A simple example

The crux of the problem is what might be called *resource locality*. Separation logic [18, 27] specialises in this second dimension of locality. One can use separation logic's *small axioms* to reason about each instruction as if it were executing in a state containing only the resources (i.e. memory cells) that it needs, and immediately deduce its effect on the entire state using the *frame rule*. The proof outline below depicts this mechanism for line 4 of Fig. 1a.

$$
\text{frame} \atop x \mapsto 1 * z \mapsto 0
\left[
\begin{array}{l}
\{x \mapsto 1 * y \mapsto 0 * z \mapsto 0\} \\
\quad \{y \mapsto 0\} \\
\quad\quad [y]:=1; \\
\quad \{y \mapsto 1\} \\
\{x \mapsto 1 * y \mapsto 1 * z \mapsto 0\}
\end{array}
\right]
{\text{small axiom} \atop \text{for heap update}}
$$

Showing such detail throughout a proof outline would clarify the effect of each instruction, but escalate the repetition. Cleverer use of the frame rule can help, but only a little – see Sect. 6. Essentially, we need a new proof representation to harness the new technology separation logic provides, and we propose the **ribbon proof**.

Figure 1b gives an example. The repetition has disappeared, and each instruction's effect is now clear: it affects exactly those assertions directly above and below it, while framed assertions (which must not mention variables written by the instruction) pass unobtrusively to the left or right. Technically, we are still invoking the frame rule at each instruction, but crucially in a ribbon proof, such invocations are implicit and do not complicate the diagram.

A bonus of this particular ribbon proof is that it emphasises that the three assignments update different memory cells. They are thus independent, and amenable to reordering or parallelisation. One can imagine obtaining a proof of the transformed program by simply sliding the left-hand column downward and the right-hand column upward. The corresponding proof outline neither suggests nor supports such manoeuvres.

Where a proof outline essentially flattens a proof to a list of assertions and instructions, our system produces geometric objects that can be navigated and modified by leveraging human visual intuition, and whose basic steps correspond exactly to separation logic's small axioms. A ribbon proof de-emphasises the program's shallow syntax, such as the order of independent instructions, and illuminates instead the deeper structure, such as the flow of resources through the code. Proof outlines focus on Hoare

triples $\{p\}\, c\, \{q\}$, and often neglect the details of entailments between assertions, $p \Rightarrow q$, even though such entailments often encode important insights about the program being verified. Ribbon proofs treat both types of judgement equally, within the same system.

There are many recent extensions of separation logic (e.g. [7–11, 14, 17, 20, 23, 31]) to which our ribbon proof technology can usefully be applied; indeed, ribbons have already aided the development of a separation logic for relaxed memory [5]. All of these program logics are based on increasingly complex reasoning principles, of which clear explanations are increasingly vital. We propose ribbon proofs as the ideal device for providing them.

Comparison with Bean's system Bean [2] introduced ribbon proofs as an extension of Fitch's *box proofs* [12] to handle the propositional fragment of bunched implications logic (BI) [24]. BI being the basis of separation logic's assertion language [18], his system can be used to prove entailments between propositional separation logic assertions. Our system expands Bean's into a full-blown program logic by adding support for commands and existentially-quantified variables. It is further distinguished by its treatment of ribbon proofs as graphs, which gives our diagrams an appealing degree of flexibility.

Contributions and paper outline We describe a diagrammatic proof system that enables a natural presentation of separation logic proofs. We prove it sound and complete with respect to separation logic (Sect. 3). We also give an alternative, graphical formalisation (Sect. 4), which is sound in the absence of the frame rule's side-condition.

We describe a prototype tool (Sect. 5) for mechanically checking ribbon proofs with the *Isabelle* proof assistant. Given a small proof script for each basic step, our tool assembles a script that verifies the entire diagram. Such tediums as the associativity and commutativity of $*$ are handled in the graphical structure, leaving the user to focus on the interesting parts of the proof.

We discuss (Sect. 6) extensions to handle concurrent separation logic, possible applications to parallelisation, and connections to proof nets, bigraphs and string diagrams.

We begin by introducing our ribbon proof system with the aid of an example. Further examples can be found in Wickerson's PhD dissertation [33]. Of those, our ribbon proof of the Version 7 Unix memory manager demonstrates that our system can present readable proofs of more complex programs than those considered in this paper.

2   An example

Let us consider a simple program for in-place reversal of a linked list.

Figure 3a presents a proof of this program as a proof outline (adapted from [27]). For a binary relation r, we write $x \mathbin{\dot{r}} y$ for $x\, r\, y \wedge emp$, where emp describes an empty heap. We write ϵ for the empty sequence, $(-)^\dagger$ for sequence reversal, and $\cdot$ for cons and concatenation. We define the $list\, \alpha\, x$ predicate by induction on the length of the sequence α:

$$list\, \epsilon\, x \stackrel{\text{def}}{=} (x \mathbin{\dot{=}} \texttt{nil}) \qquad list\, (i \cdot \alpha')\, x \stackrel{\text{def}}{=} (\exists x'.\, x \mapsto i, x' * list\, \alpha'\, x'),$$
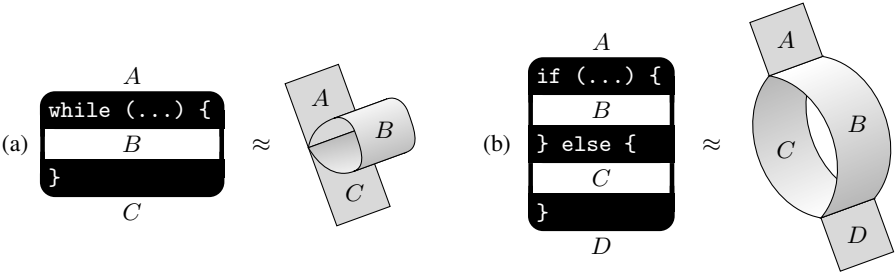
Fig. 2. While-loops and if-statements, pictorially

where $x \mapsto y, z$ abbreviates $(x \mapsto y) * (x + 1 \mapsto z)$.

The invariant (line 5) states that x and y are linked lists representing two sequences α and β such that the initial sequence α_0 is obtained by concatenating the reverse of β onto α. Our proof outline seeks to clarify the proof by making minimal changes between successive assertions, despite this making the proof large and highly redundant. Alternatively, intermediate assertions can be elided, but this can make the proof hard to follow. Either way, proof outlines do not make the structure of the proof clear.

Figure 3b presents a ribbon proof for the same program. It comprises

- *steps*, each labelled with an instruction (black) or a justification of an entailment (dark grey),
- *ribbons* (light grey), each labelled with an assertion, and
- *existential boxes*, which delimit the scope of logical variables.

The ribbon proof advances vertically, and the resources (memory cells) being operated upon are distributed horizontally across the ribbons. Instructions are positioned according to the resources they access, not merely according to the syntax of the program, as in the proof outline. Horizontal separation between ribbons corresponds to the separating conjunction of the assertions on those ribbons; that is, parallel ribbons refer to disjoint sets of memory cells. Because $*$ is commutative, we can 'twist' one ribbon over another. The resource distribution is not only unordered, but also non-uniform, so the width of a ribbon is not proportional to the amount of resource it describes. In particular, the assertion 'x $\neq$ nil' obtained upon entering the while-loop describes no memory cells at all; it merely states that the program variable x is not the null pointer. A gap in the diagram (e.g. above the 'y:=nil' step) corresponds to the 'emp' assertion.

While-loops are special steps that contain further nested steps. The loop invariant is the collection of ribbons and existential boxes entering the top of the loop. This collection must be recreated at the end of the loop body, so that one could roll the proof into the shape drawn in Fig. 2a. If-statements are not depicted in our example, but appear in Wickerson's PhD dissertation [33]. They are treated straightforwardly: the ribbons and boxes entering the then-branch must match those entering the else-branch, and likewise at the two exit points, so that the proof could be cut and folded into the three-dimensional shape suggested in Fig. 2b.
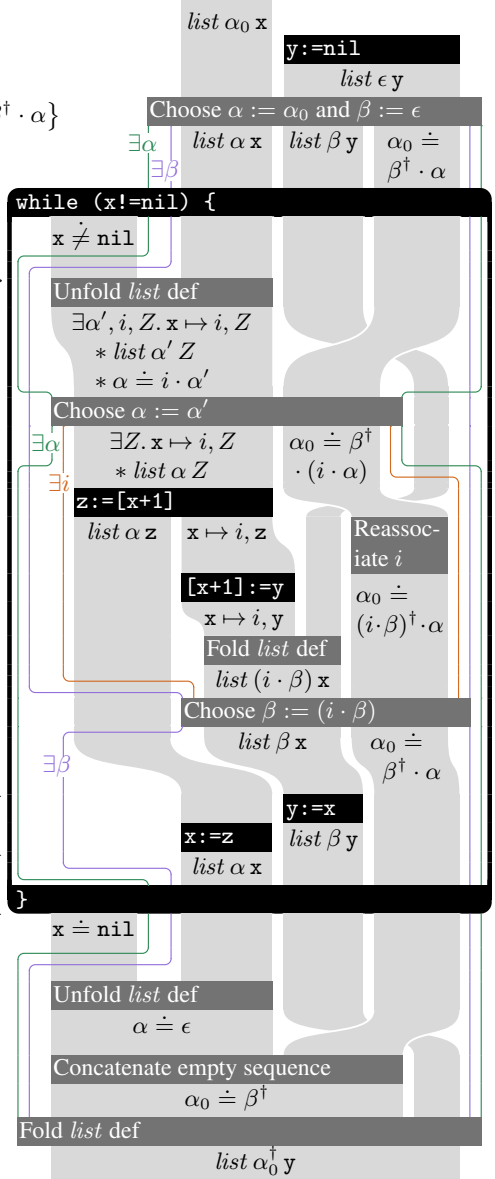
After the 'z:=[x+1]' step, the assertion '$list\ \alpha\ z$' is not needed for a while. In a proof outline, this assertion would either be temporarily removed via an explicit ap-

1 $\left\{ list\ \alpha_0\ \mathtt{x} \right\}$
2 `y:=nil;`
3 $\left\{ list\ \alpha_0\ \mathtt{x} * list\ \epsilon\ \mathtt{y} \right\}$
4 `// Choose` $\alpha := \alpha_0$ `and` $\beta := \epsilon$
5 `while` $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha \right\}$
   `(x!=nil) {`
6 $\left\{ \begin{array}{l} \exists \alpha, \beta.\ \mathtt{x} \ne \mathtt{nil} * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha \end{array} \right\}$
7 `// Unfold` $list$ `def`
8 $\left\{ \begin{array}{l} \exists \alpha, \beta.\ (\exists \alpha', i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha'\ \mathtt{z} \\ * \alpha \doteq i \cdot \alpha') * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha \end{array} \right\}$
9 `// Choose` $\alpha := \alpha'$
10 $\left\{ \begin{array}{l} \exists \alpha, \beta, i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha\ Z \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y} \end{array} \right\}$
11 `z:=[x+1];`
12 $\left\{ \begin{array}{l} \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \alpha_0 \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y} \end{array} \right\}$
13 `// Reassociate` i
14 $\left\{ \begin{array}{l} \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y} \end{array} \right\}$
15 `[x+1]:=y;`
16 $\left\{ \begin{array}{l} \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{y} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y} \end{array} \right\}$
17 `// Fold` $list$ `def`
18 $\left\{ \begin{array}{l} \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * list\ (i \cdot \beta)\ \mathtt{x} \\ * \alpha_0 \doteq (i \cdot \beta)^\dagger \cdot \alpha \end{array} \right\}$
19 `// Choose` $\beta := (i \cdot \beta)$
20 $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{z} * list\ \beta\ \mathtt{x} * \alpha_0 \doteq \beta^\dagger \cdot \alpha \right\}$
21 `y:=x;`
22 $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{z} * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha \right\}$
23 `x:=z;`
24 $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha \right\}$
25 `}`
26 $\left\{ \begin{array}{l} \exists \alpha, \beta.\ \mathtt{x} \doteq \mathtt{nil} * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} \\ * \alpha_0 \doteq \beta^\dagger \cdot \alpha \end{array} \right\}$
27 `// Unfold` $list$ `def`
28 $\left\{ \exists \alpha, \beta.\ \alpha \doteq \epsilon * list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \cdot \alpha \right\}$
29 `// Concatenate empty sequence`
30 $\left\{ \exists \beta.\ list\ \beta\ \mathtt{y} * \alpha_0 \doteq \beta^\dagger \right\}$
31 `// Fold` $list$ `def`
32 $\left\{ list\ \alpha_0^\dagger\ \mathtt{y} \right\}$

(a) A proof outline

(b) A ribbon proof

Fig. 3. Two proofs of list reverse

plication of the frame rule or, as is done in Fig. 3a, redundantly repeated at every intermediate point. In the ribbon proof, it slides discreetly down the left-hand side. This indicates that the assertion is *inactive* without suggesting that it has been *removed*.

The proof outline obscures the usage of the logical variables α and β. The witness for α changes after line 8, then stays the same until line 24; meanwhile, β's witness is constant through lines 5 to 18 before becoming the previous witness prepended with i. This structure can only be spotted through careful examination of the proof outline (aided by the textual hints on lines 9 and 19). The scoping of logical variables in the ribbon proof, through the use of existential boxes, is far more satisfactory. Boxes extend horizontally across several ribbons, but also vertically to indicate the range of steps over which the same witness is used. Horizontally, existential boxes must be well-nested; this corresponds to the static scoping of existential quantifiers in assertions. Vertically, however, boxes may overlap. Figure 4 depicts how the boxes for α and β overlap in Fig. 3b. As explained in Sect. 3.1, such 'overlaps' are formally treated as entailment steps of the form $\exists x.\, \exists y.\, p \Rightarrow \exists y.\, \exists x.\, p$. Similarly, boxes may be stretched horizontally (see, for instance, immediately below the loop in Fig. 3b) in accordance with the entailment $p * (\exists x.\, q) \Rightarrow \exists x.\, p * q$ (for x not in p). We thus obtain an intriguing proof structure – present in neither the proof outline nor the underlying derivation tree – in which the scopes of logical variables do not follow the program's syntactic structure, but are instead *dynamically* scoped. Section 6 contains further discussion.
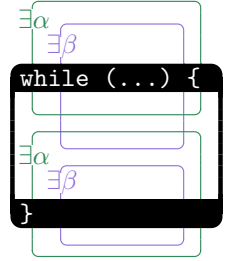

Fig. 4. Existential boxes, vertically overlapping

We close this section by explaining a shortcoming in the proof system as currently presented. One nicety of Fig. 3b is that the 'Reassociate i' entailment, being horizontally separated from its neighbouring proof steps, can clearly be moved a little earlier or later. (Close inspection is necessary to discover this from the proof outline.) But similar reasoning allows the assignments 'y:=x' and 'x:=z' to be swapped, unsoundly. We ensure our proof system is sound either by forbidding such manoeuvres altogether (Sect. 3) or by encoding variable dependencies into the ribbons themselves (Sect. 4).

3   Formalisation

Let us now formalise the concepts introduced in the previous section. We introduce in Sect. 3.1 a two-dimensional syntax for diagrams, and explain how it can generate the pictures we have already seen. We present the rules of our diagrammatic proof system in Sect. 3.2. We relate ribbon proofs to separation logic in Sect. 3.3.

Proofs performed by hand are annotated with □, while those mechanically verified using the *Isabelle* proof assistant are annotated with ⚓, and can be viewed online at: http://www.cl.cam.ac.uk/~jpw48/ribbons.html

Definition 1 (Assertions). *Let p range over a set of ordinary separation logic assertions, containing at least the following constructions:*

$$\mathsf{Assertion} \;\overset{\text{def}}{=}\; \{p ::= emp \mid p * p \mid \exists x.\, p \mid \ldots\}.$$

$$\frac{\vdash_{SL}\{p\}\,c\,\{q\} \qquad wr(c) \cap rd(r) = \emptyset}{\vdash_{SL}\{p*r\}\,c\,\{q*r\}} \qquad \frac{(p,c,q) \in \mathsf{Axioms}}{\vdash_{SL}\{p\}\,c\,\{q\}} \qquad \frac{\vdash_{SL}\{p\}\,c\,\{q\}}{\vdash_{SL}\{\exists x.\,p\}\,c\,\{\exists x.\,q\}}$$

$$\frac{\vdash_{SL}\{p_1\}\,c\,\{q_1\} \qquad \vdash_{SL}\{p_2\}\,c\,\{q_2\}}{\vdash_{SL}\{p_1 \vee p_2\}\,c\,\{q_1 \vee q_2\}} \qquad \frac{\vdash_{SL}\{p'\}\,c\,\{q'\} \qquad p \Rightarrow p' \qquad q' \Rightarrow q}{\vdash_{SL}\{p\}\,c\,\{q\}} \qquad \frac{\vdash_{SL}\{p\}\,c_1\,\{q\} \qquad \vdash_{SL}\{p\}\,c_2\,\{q\}}{\vdash_{SL}\{p\}\,c_1\,\mathtt{or}\,c_2\,\{q\}}$$

$$\frac{\vdash_{SL}\{p\}\,c_1\,\{q\} \qquad \vdash_{SL}\{q\}\,c_2\,\{r\}}{\vdash_{SL}\{p\}\,c_1\,;c_2\,\{r\}} \qquad \frac{}{\vdash_{SL}\{p\}\,\mathtt{skip}\,\{p\}} \qquad \frac{\vdash_{SL}\{p\}\,c\,\{p\}}{\vdash_{SL}\{p\}\,\mathtt{loop}\,c\,\{p\}}$$

Fig. 5. Proof rules for commands

Definition 2 (Commands). *Let c range over the commands of a sequential program-ming language, containing at least sequential composition (which is associative),* skip *(the unit of sequential composition), and non-deterministic choice and looping:*

$$\mathsf{Command} \ \overset{\text{def}}{=} \ \{c ::= c\,;\,c \mid \mathtt{skip} \mid c\,\mathtt{or}\,c \mid \mathtt{loop}\,c \mid \ldots\}.$$

If a primitive '$\mathtt{assume}\,b$' command is available (where b is a *pure* assertion; that is, independent of the heap) then standard if-statements and while-loops can be derived:

$$\mathtt{if}\,b\,\mathtt{then}\,c_1\,\mathtt{else}\,c_2 \ \overset{\text{def}}{=} \ (\mathtt{assume}\,b\,;\,c_1)\,\mathtt{or}\,(\mathtt{assume}\,\neg b\,;\,c_2)$$

$$\mathtt{while}\,b\,\mathtt{do}\,c \ \overset{\text{def}}{=} \ \mathtt{loop}(\mathtt{assume}\,b\,;\,c)\,;\,\mathtt{assume}\,\neg b.$$

We assume a separation logic comprising the rules given in Fig. 5 plus a set of Axioms. In the first rule, the *frame rule*, the rd and wr functions respectively extract the sets of program variables read and written.

Remark 1. We do not consider Hoare logic's conjunction rule in this paper. Conjunc-tion and universal quantification can still appear inside individual ribbon assertions. We could design graphical analogues (which would resemble our treatment of disjunction and existential quantification) but this would complicate our graphical language with constructs that are seldom used in separation logic proofs.

3.1   Syntax of diagrams

We present a syntax that can generate the pictures seen in the preceding section. Each diagram is built up as a sequence of rows, each containing a single proof step. We thus refer to such diagrams as 'stratified'. (Section 4 will present an alternative formalisa-tion that does not impose such strict sequentiality.) We begin by introducing *interfaces*, which are the top and bottom boundaries through which diagrams can be composed.

Definition 3 (Interfaces). *An interface is either a single ribbon labelled with an asser-tion, an empty interface (shown as whitespace in pictures), two interfaces side by side, or an existential box wrapped around an interface:*

$$\mathsf{Interface} \ \overset{\text{def}}{=} \ \{P ::= \ \boxed{p} \ \mid \ \varepsilon \ \mid \ P\,P \ \mid \ \exists^x_i P| \ \}.$$
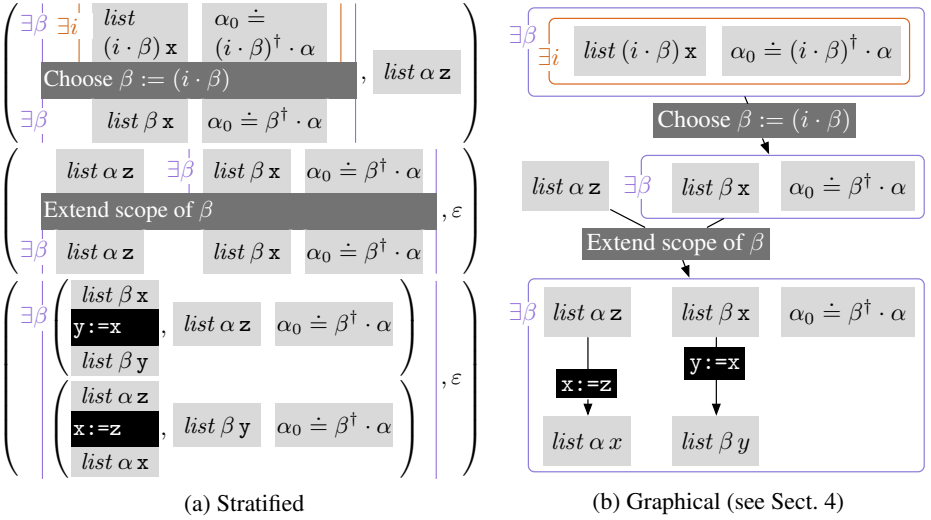
(a) Stratified   (b) Graphical (see Sect. 4)

Fig. 6. Two ways to parse a fragment of Fig. 3b

The asn function maps an interface to the assertion it represents:

$$asn\ \boxed{p} = p \qquad\qquad asn\,(P\,Q) = asn\,P * asn\,Q$$
$$asn\,\varepsilon = emp \qquad\qquad asn\ {}^{\exists x}_{|}P| = \exists x.\ asn\,P.$$

When clarity demands it, we shall write $P \otimes Q$ instead of $P\,Q$, and hence $\otimes_{i \in I} P_i$ for iterated composition. We equate interfaces up to $(P\,Q)\,R = P\,(Q\,R)$, $P\,\varepsilon = \varepsilon\,P = P$ and $P\,Q = Q\,P$. Since $\otimes$ commutes, ribbon 'twisting' is merely a presentational artefact.

A *diagram* can be thought of as a mapping between two interfaces.

Definition 4 (Diagrams). *A diagram $D \in$ Diagram is a non-empty list of* rows $\rho \in$ Row. *When space permits, we align the list elements in a single column without punctuation. A row is a pair (γ, F) comprising a* cell $\gamma \in$ Cell *and a* frame $F \in$ Interface. *The syntax of cells is as follows:*

$$\mathsf{Cell} \stackrel{\text{def}}{=} \{\gamma ::= P \mid \begin{array}{c} P \\ \boxed{\ \ c\ \ } \\ P \end{array} \mid {}^{\exists x}_{|}D| \mid \begin{array}{c} P \\ \boxed{D} \\ \boxed{\text{or}} \\ \boxed{D} \\ P \end{array} \mid \begin{array}{c} P \\ \boxed{\texttt{loop}} \\ \boxed{D} \\ P \end{array} \}.$$

To illustrate how this syntax is used, Fig. 6a shows a term of Diagram that corresponds to a fragment of the picture in Fig. 3b. Note that the cell in each row is always pushed to the left-hand side. In the concrete pictures, it can be moved to allow corresponding ribbons in different rows to be aligned, and hence for redundant labels to be removed. Each entailment $p \Rightarrow q$ is handled as the basic step $\{p\}\ \texttt{skip}\ \{q\}$. Rather than write 'skip', we label such a step with a justification of the entailment, and colour it dark grey to emphasise those steps that actually contain program instructions. Concerning
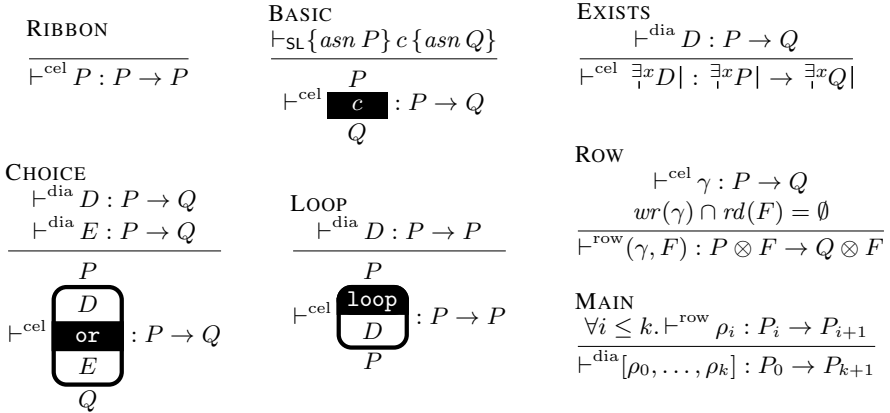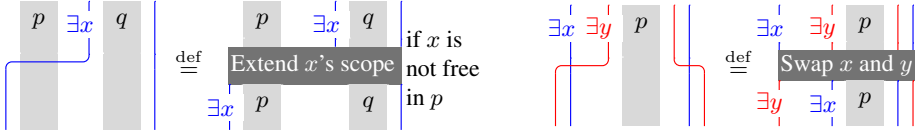
RIBBON

$$\overline{\vdash^{\mathrm{cel}} P : P \to P}$$

BASIC

$$\frac{\vdash_{\mathsf{SL}} \{asn\, P\}\, c\, \{asn\, Q\}}{\vdash^{\mathrm{cel}} \boxed{c} : P \to Q}$$

EXISTS

$$\frac{\vdash^{\mathrm{dia}} D : P \to Q}{\vdash^{\mathrm{cel}} \exists x D | : \exists x P | \to \exists x Q |}$$

CHOICE

$$\frac{\vdash^{\mathrm{dia}} D : P \to Q \quad \vdash^{\mathrm{dia}} E : P \to Q}{\vdash^{\mathrm{cel}} \boxed{\begin{array}{c} D \\ \mathbf{or} \\ E \end{array}} : P \to Q}$$

LOOP

$$\frac{\vdash^{\mathrm{dia}} D : P \to P}{\vdash^{\mathrm{cel}} \boxed{\begin{array}{c} \mathbf{loop} \\ D \end{array}} : P \to P}$$

ROW

$$\frac{\vdash^{\mathrm{cel}} \gamma : P \to Q \quad wr(\gamma) \cap rd(F) = \emptyset}{\vdash^{\mathrm{row}} (\gamma, F) : P \otimes F \to Q \otimes F}$$

MAIN

$$\frac{\forall i \le k.\, \vdash^{\mathrm{row}} \rho_i : P_i \to P_{i+1}}{\vdash^{\mathrm{dia}} [\rho_0, \dots, \rho_k] : P_0 \to P_{k+1}}$$

Fig. 7. Proof rules for stratified ribbon diagrams

existential boxes: the operations of extending, contracting and commuting are really the entailments depicted informally below. Having to show these entailments explicitly would make Fig. 3b much more repetitive. (We are working on an improved formalisation that supports these operations directly – see Sect. 6 for further discussion.)

3.2 Proof rules for diagrams

There are two pertinent questions to be asked of a given ribbon diagram. The first question is: is it a valid proof? This subsection develops a *provability* judgement to answer this. The second question – if this ribbon diagram *is* deemed valid, what does it prove? – is addressed in the next subsection.

The rules given in Fig. 7 define provability judgements for cells ($\vdash^{\mathrm{cel}}$), for rows ($\vdash^{\mathrm{row}}$) and for diagrams ($\vdash^{\mathrm{dia}}$). Each judgement ascribes a type, which comprises the top and bottom interfaces of that object.

The ROW and MAIN rules recall Hoare logic's sequencing rule and separation logic's frame rule. They embody the 'locally checkable' nature of ribbon proofs: that the entire diagram is valid if each row is valid in isolation, and that a row is valid if its active cell is valid and writes no program variable that is read elsewhere in the row.

The BASIC rule corresponds to an ordinary separation logic judgement $\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\}$. This judgement may be arbitrarily complex, so a ribbon diagram may be no easier to check than a traditional proof outline. This is intentional. Our formalisation *allows p* and q to be minimised, by framing common fragments away, but does not *demand* this. The command c *can* be reduced to skip or some primitive command, but this may not be desirable if one requires only a high-level overview proof. A ribbon diagram can

$$com[(\gamma_0, F_0), \ldots, (\gamma_k, F_k)] \qquad com\, P = \text{skip} \qquad com\, \exists^x_1 D| = com\, D$$
$$= com\, \gamma_0 ; \cdots ; com\, \gamma_k$$

Fig. 8. Extracting a command from a stratified diagram

thus be viewed as a flexible combination of diagrammatic and traditional proofs, with the BASIC rule as the interface between these two levels.

We remark that these proof rules provide only limited mechanisms for building new diagrams from old. Diagrams can be wrapped in existential boxes, or put inside choice or loop diagrams, but not stacked vertically or placed side by side. One can define operations for composing elements of Diagram in sequence or in parallel, and hence additional proof rules for diagrams so composed. The process is straightforward, and described in Wickerson's PhD dissertation [33].

3.3 Semantics of diagrams

A stratified ribbon diagram denotes a Hoare triple. The pre- and postconditions of this triple are the assertions represented by the diagram's top and bottom interfaces. The command being proved is extracted by composing the labels on all of the proof steps in top-to-bottom order. Figure 8 defines the function responsible for this extraction. We hence obtain the following soundness result for ribbon proofs.

Theorem 1 (Soundness – stratified diagrams). *Separation logic can encode any provable ribbon diagram.*

$$\vdash^{\text{dia}} D : P \to Q \Longrightarrow \vdash_{\text{SL}} \{asn\, P\}\ com\, D\ \{asn\, Q\}.$$

Proof. By mutual rule induction on $\vdash^{\text{cel}}$, $\vdash^{\text{row}}$, and $\vdash^{\text{dia}}$. ∎

Ribbon diagrams are trivially complete, because the BASIC rule can be invoked right at the root of the proof tree. In fact, ribbon diagrams remain complete even when the BASIC rule can occur only immediately beneath an axiom or the rule of consequence.

Theorem 2 (Completeness – stratified diagrams). *A strengthened ribbon proof system in which the BASIC rule is replaced by*

$$\frac{(asn\, P, c, asn\, Q) \in \text{Axioms}}{\vdash^{\text{cel}}\ \begin{array}{c} P \\ \boxed{c} \\ Q \end{array} : P \to Q} \qquad and \qquad \frac{asn\, P \Rightarrow asn\, Q}{\vdash^{\text{cel}}\ \begin{array}{c} P \\ \boxed{\text{skip}} \\ Q \end{array} : P \to Q}$$

can encode any separation logic proof.

$$\vdash_{\text{SL}} \{p\}\, c\, \{q\} \Longrightarrow \exists D, P, Q.\, c \in com\, D \wedge p = asn\, P \wedge q = asn\, Q \wedge \vdash^{\text{dia}} D : P \to Q$$

Proof. By rule induction on $\vdash_{\mathsf{SL}}$.   □

The main problem with the formalisation given in this section is that it sacrifices much of the flexibility we expect in our ribbon diagrams. It is often sound to tweak the layout of a diagram by sliding steps up or down or reordering ribbons, but by thinking of our diagrams as sliced into a sequence of rows, we rule out *all* such manoeuvres.

4   Graphical formalisation

We now give an alternative formalisation, in which diagrams are represented not as a sequence of rows, but as graphs.

Our 'graphical' diagrams are more flexible than their 'stratified' cousins, but extra precautions must be taken to ensure soundness. The core difficulty is the side-condition on the frame rule: that the command writes no program variable in the frame. With stratification, the frame is clearly delimited, so this condition is easily checked. Without it, this check would become more global: a command may affect a ribbon that appears far above or below itself in a laid-out diagram. Our simple solution is to require henceforth that the frame rule has no side-condition. This requirement could be met by abolishing program variables altogether, leaving only the heap and numerical constants. A more practical alternative, explored later in this section, is to use the *variables-as-resource* paradigm [4].

Our graphs are nested, directed, acyclic hypergraphs. Ribbons correspond to nodes, and basic steps to hyperedges. Existential boxes are represented as single nodes that contain a nested graph. Likewise, choice diagrams and loop diagrams are represented by single hyperedges that contain, respectively, one or two nested graphs.

Definition 5 (Graphical diagrams, assertion-gadgets and command-gadgets). *Let* $\mathcal{V}$ *be an infinite set of node-identifiers. We define a language of* assertion-gadgets, command-gadgets *and* graphical diagrams *as follows.*

$$\mathsf{AsnGadget} = \{A ::= \boxed{p} \mid \boxed{\exists x\ G}\} \qquad \mathsf{ComGadget} = \{C ::= \boxed{c} \mid \boxed{\begin{array}{c} G \\ \mathsf{or} \\ G \end{array}} \mid \boxed{\begin{array}{c} \mathsf{loop} \\ G \end{array}}\}$$

$$\mathsf{GDiagram} = \{G \mid \Lambda_G \in V_G \to \mathsf{AsnGadget},\ E_G \subseteq_{\mathrm{fin}} \mathcal{P}(V_G) \times \mathsf{ComGadget} \times \mathcal{P}(V_G),$$
$$V_G \subseteq_{\mathrm{fin}} \mathcal{V},\ acyclic(G)\ and\ linear(G),\ where\ G = (V_G, \Lambda_G, E_G)\}$$

The definitions are mutually recursive, and are well-formed because the definienda (left-hand sides) appear only positively in the definientia (right-hand sides).[1] The first of these equations defines an assertion-gadget $A \in \mathsf{AsnGadget}$ *to be either a ribbon or an existential box. The second defines a command-gadget* $C \in \mathsf{ComGadget}$ *to be either a basic step, a choice diagram, or a loop diagram. The third equation defines a graphical diagram* $G \in \mathsf{GDiagram}$ *to be a triple* (V_G, Λ_G, E_G) *that comprises:*

- *a finite set* $V_G \subseteq_{\mathrm{fin}} \mathcal{V}$ *of* node identifiers;
- *a labelling* $\Lambda_G : V_G \to \mathsf{AsnGadget}$ *that associates each node identifier with an assertion-gadget; and*

---

[1] This is true even for the occurrence of ComGadget in the definiens of GDiagram, because the set in which it appears is finite.

– *a finite set $E_G \subseteq_{\text{fin}} \mathcal{P}(V_G) \times \text{ComGadget} \times \mathcal{P}(V_G)$ of hyperedges $(\mathbf{v}, C, \mathbf{w})$, each comprising a set $\mathbf{v}$ of tail identifiers, a command-gadget C, and a set $\mathbf{w}$ of head identifiers,*

and which satisfies the following two properties.

ACYCLICITY: *Let us write $v \rightarrow w$ if $v \in \mathbf{v}$ and $w \in \mathbf{w}$ for some $(\mathbf{v}, C, \mathbf{w}) \in E_G$. Then define $acyclic(G)$ to hold iff the transitive closure of $\rightarrow$ is irreflexive.*

LINEARITY: *Define $linear(G)$ to hold iff the hyperedges in E_G have no common heads and no common tails. (This forbids the duplication or merging of ribbons, in accordance with $p \Rightarrow p * p$ and $p * p \Rightarrow p$ being invalid in separation logic.)*

Remark 2. We could represent our diagrams by a single graph, with dedicated 'parent' edges to simulate the nesting hierarchy. However, mindful of our *Isabelle* formalisation, and that "reasoning about graphs [...] can be a real hassle in HOL-based theorem provers" [34], we prefer to use an algebraic datatype to depict the hierarchy.

Figure 6b presents a term of GDiagram that corresponds to a fragment of the picture in Fig. 3b. Unlike Fig. 6a, this representation does not impose a strict ordering between the 'y:=x' and 'x:=z' instructions. As such, this proof is *invalid*; the figure serves merely to demonstrate how the graphical syntax is used.

   The problem is that the graph does not take into account dependencies on program variables. To address this, let us remove the side-condition on the frame rule in our axiomatisation $\vdash_{\text{SL}}$ of separation logic (Fig. 5). The new proof system thus obtained shall be written as $\vdash_{\text{SL}}^*$. We shall now develop proof rules for graphical diagrams, and show them to be sound and complete with respect to $\vdash_{\text{SL}}^*$. Section 4.3 describes the application of ribbon proofs to variables-as-resource, which is one instance of $\vdash_{\text{SL}}^*$.

4.1   Proof rules for graphical diagrams

Proof rules for graphical diagrams, command-gadgets and assertion-gadgets are defined in Fig. 9, which refers to the *top* and *bot* functions defined below. The judgement $\vdash^{\text{gra}} G : P \rightarrow Q$ means that the diagram G, precondition P, and postcondition Q form a valid proof. The interfaces P and Q are always equal to $top(G)$ and $bot(G)$ respectively, so we sometimes omit them. The judgements for command-gadgets and assertion-gadgets are similar, the latter without interfaces.

Definition 6 (Top and bottom interfaces). *These functions extract interfaces from assertion-gadgets and from diagrams. For assertion-gadgets:*

$$top \boxed{p} = p \qquad bot \boxed{p} = p \qquad top \boxed{\exists x\, G} = \,_!^{\exists x} top\, G| \qquad bot \boxed{\exists x\, G} = \,_!^{\exists x} bot\, G|.$$

For diagrams:

$$top(G) = \bigotimes_{v \in initials\, G} top(\Lambda_G\, v) \qquad bot(G) = \bigotimes_{v \in terminals\, G} bot(\Lambda_G\, v)$$

where $initials(G) = V_G \setminus \bigcup_{(\_,\_,\mathbf{v}) \in E_G} \mathbf{v}$ and $terminals(G) = V_G \setminus \bigcup_{(\mathbf{v},\_,\_) \in E_G} \mathbf{v}$.

As was the case for stratified diagrams, one can define operations for composing elements of GDiagram in sequence or parallel, and hence additional proof rules for graphical diagrams so composed [33].

GRIBBON

$$\frac{}{\vdash^{\text{asn}} \boxed{p}}$$

GBASIC

$$\frac{\vdash^*_{\text{SL}} \{asn\, P\}\, c\, \{asn\, Q\}}{\vdash^{\text{com}} \boxed{\blacksquare c\ } : P \to Q}$$

GEXISTS

$$\frac{\vdash^{\text{gra}} G}{\vdash^{\text{asn}} \boxed{\exists x G}}$$

GCHOICE

$$\frac{\vdash^{\text{gra}} G_1 : P \to Q \qquad \vdash^{\text{gra}} G_2 : P \to Q}{\vdash^{\text{com}} \boxed{\begin{array}{c} G_1 \\ \text{or} \\ G_2 \end{array}} : P \to Q}$$

GLOOP

$$\frac{\vdash^{\text{gra}} G : P \to P}{\vdash^{\text{com}} \boxed{\begin{array}{c} \text{loop} \\ G \end{array}} : P \to P}$$

GMAIN

$$\frac{\forall v \in V_G.\, \vdash^{\text{asn}} \Lambda_G\, v \qquad \forall (\mathbf{v}, C, \mathbf{w}) \in E_G.\, \vdash^{\text{com}} C : \otimes_{v \in \mathbf{v}}\, bot(\Lambda_G\, v) \to \otimes_{w \in \mathbf{w}}\, top(\Lambda_G\, w)}{\vdash^{\text{gra}} G : top(G) \to bot(G)}$$

Fig. 9. Proof rules for graphical diagrams

$$coms(G) = \{c_0 \mathbin{;} \cdots \mathbin{;} c_{k-1} \mathbin{;} \text{skip} \mid \exists [x_0, \ldots, x_{k-1}] \in lin\, G.\, \forall i < k.\, c_i \in coms\, x_i\}$$

$$coms\, \boxed{p} = \{\text{skip}\} \qquad coms\, \boxed{\exists x G} = coms\, G \qquad coms\, \boxed{\blacksquare c\ } = \{c\}$$

$$coms\, \boxed{\begin{array}{c} G_1 \\ \text{or} \\ G_2 \end{array}} = \begin{array}{l} \{c_1\ \text{or}\ c_2 \mid \\ \quad c_1 \in coms\, G_1, \\ \quad c_2 \in coms\, G_2\} \end{array} \qquad coms\, \boxed{\begin{array}{c} \text{loop} \\ G \end{array}} = \{\text{loop}\, c \mid c \in coms\, G\}$$

Fig. 10. Extracting commands from a diagram

4.2 Semantics of graphical diagrams

Since graphical diagrams have a parallel nature, but our language is only sequential, it follows that each graphical diagram proves not a single command, but a set of commands, each one a linear extension of the partial order imposed by the diagram. The *coms* function defined in Fig. 10 is responsible for extracting this set from a given diagram. Each command is obtained by picking an ordering of command- and assertion-gadgets that is compatible with the partial order defined by the edges (this is the purpose of the *lin* function defined below), then recursively extracting a command from each gadget and sequentially composing the results.

Definition 7 (Linear extensions). *For a diagram G, we define $lin\, G$ as the set of all lists $[x_0, \ldots, x_{k-1}]$ of* AsnGadget*s and* ComGadget*s, for which there exists a bijection $\pi : k \to V_G \cup E_G$ that satisfies, for all $(\mathbf{v}, C, \mathbf{w}) \in E_G$:*

$$\forall v \in \mathbf{v}.\, \pi^{-1}(v) < \pi^{-1}(\mathbf{v}, C, \mathbf{w}) \qquad \forall w \in \mathbf{w}.\, \pi^{-1}(\mathbf{v}, C, \mathbf{w}) < \pi^{-1}(w)$$

and where, for all $i < k$: $x_i = \Lambda_G(v)$ if $\pi(i) = v$, and $x_i = C$ if $\pi(i) = (\mathbf{v}, C, \mathbf{w})$.

By ACYCLICITY, every diagram admits at least one linear extension.

Theorem 3 (Soundness – graphical diagrams). *Separation logic without the side-condition on the frame rule can encode any provable ribbon diagram:*

$$\vdash^{\mathrm{gra}} G : P \to Q \implies \forall c \in coms\, G. \vdash^{*}_{\mathsf{SL}} \{asn\, P\}\, c\, \{asn\, Q\}.$$

Proof. By mutual induction on $\vdash^{\mathrm{gra}}$, $\vdash^{\mathrm{com}}$ and $\vdash^{\mathrm{dia}}$. See [33] for details.   ♣

Theorem 4 (Completeness – graphical diagrams). *A strengthened ribbon proof system in which the* GBASIC *rule is replaced by*

$$\frac{(asn\, P, c, asn\, Q) \in \mathsf{Axioms}}{\vdash^{\mathrm{com}} \boxed{c} : P \to Q} \quad and \quad \frac{asn\, P \Rightarrow asn\, Q}{\vdash^{\mathrm{com}} \boxed{\texttt{skip}} : P \to Q}$$

can encode any proof in separation logic without the side-condition on the frame rule.

$$\vdash^{*}_{\mathsf{SL}} \{p\}\, c\, \{q\} \implies \exists G, P, Q.\, c \in coms\, G \wedge p = asn\, P \wedge q = asn\, Q \wedge \vdash^{\mathrm{gra}} G : P \to Q$$

Proof. By rule induction on $\vdash^{*}_{\mathsf{SL}}$.   □

4.3   Using variables-as-resource

The variables-as-resource paradigm [4] treats program variables a little like separation logic treats heap cells. Each program variable x is associated with a piece of resource, all of which (written $Own_1(x)$) must be held to write to x, and some of which ($Own_\pi(x)$ for some $0 < \pi \leq 1$) must be held to read it. This treatment replaces the use of rd and wr sets in Fig. 5. The variables-as-resource proof system is an instance of separation logic without the side-condition on the frame rule, and can be obtained from $\vdash^{*}_{\mathsf{SL}}$ simply by selecting an appropriate Axioms set.

Figure 11 exhibits a ribbon proof, conducted using variables-as-resource, of the list-reversal program from Sect. 2. Variables-as-resource dictates that every assertion in the proof is accompanied by one Own predicate per program variable it mentions. For instance, the precondition $list\, \alpha_0\, x$ is paired with some of x's resource. The extra shading is merely syntactic sugar; for instance:

$$\boxed{\texttt{x}, \tfrac{1}{2}\texttt{y}\ \ \texttt{x} \mapsto i, \texttt{y}} \quad \overset{\mathrm{def}}{=} \quad \boxed{Own_1(\texttt{x}) * Own_{.5}(\texttt{y}) * \texttt{x} \mapsto i, \texttt{y}}.$$

The other preconditions – the resources associated with y and z – entitle the program to write to these program variables in due course. Note that at the entry to the while loop, part of x's resource is required in order to carry out the test of whether x is zero. At various points in the proof, variable resources are split or combined, but their total is always conserved.

Figure 11 introduces a couple of novel visual features: ribbons may pass 'underneath' basic steps to reduce the need for twisting (see the three 'Choose . . .' steps), and horizontal space is conserved by writing some assertions sideways. The diagram can be laid out in several ways, unconstrained by the stratification strategy of the previous section, so there exists the potential to use the same diagram to justify several variations of a program. Recall the shortcoming of Fig. 3b: that it misleadingly suggested that 'y:=x' and 'x:=z' could be safely permuted. Figure 11 forbids this by inserting a ribbon between them labelled 'x'. On the other hand, both figures agree that the 'Reassociate i' step can be safely manoeuvred up or down a little.
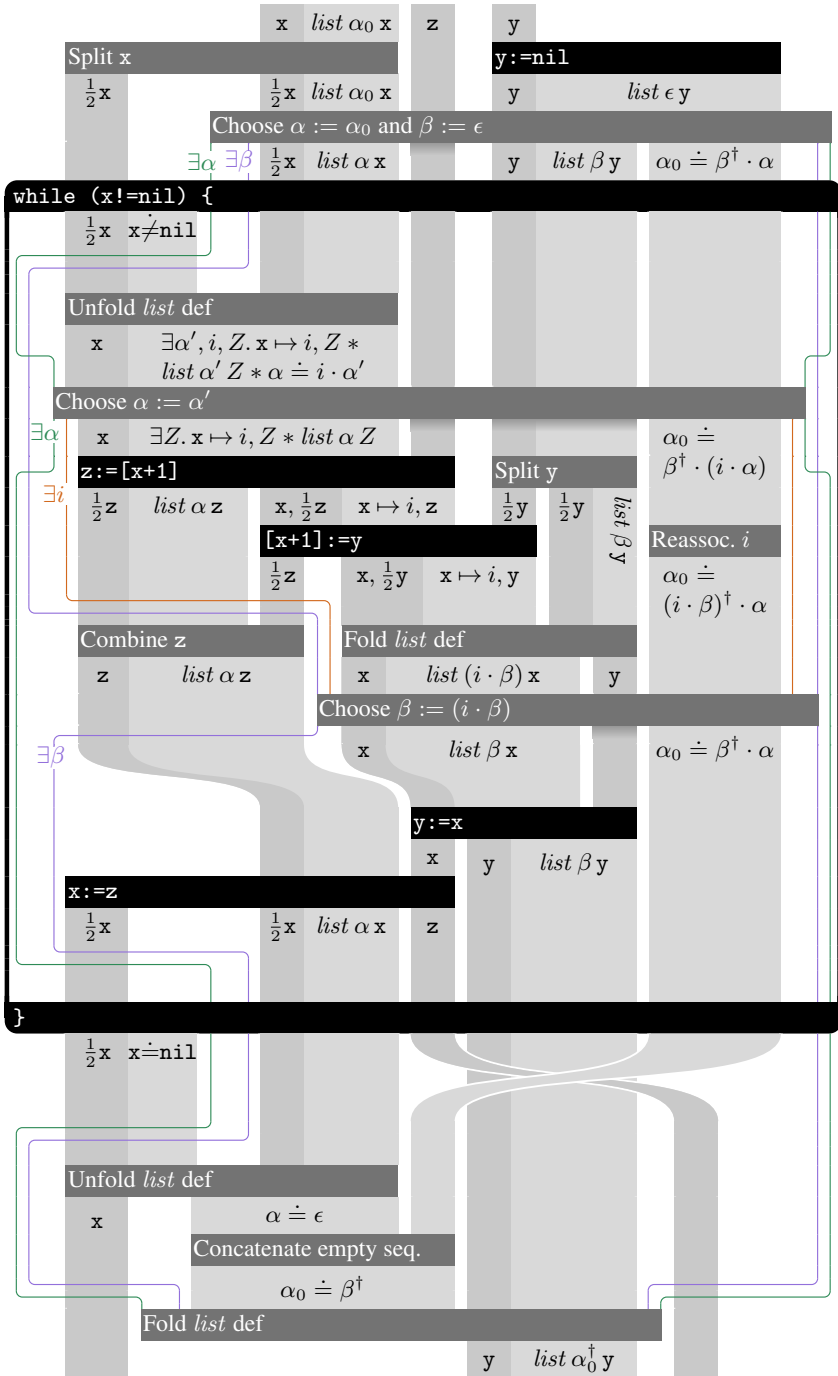
Fig. 11. A ribbon proof of list reverse using variables-as-resource

4.4   Stratified or graphical?

We have presented two alternative formalisations of ribbon diagrams.

The stratified version supports traditional separation logic (with its side-condition on the frame rule), and the formalisation is simpler, but its proof objects are less manoeuvrable. Concrete pictures should be drawn carefully so they can be successfully parsed into a sequence of rows.

The graphical version works with any separation logic whose frame rule has no side-conditions, variables-as-resource being one example. Another example is Views [7], which can encode a wide variety of program logics. The use of variables-as-resource requires much splitting, distributing and re-combining of the resources associated with each program variable, and this is perhaps an unnecessary burden if one seeks merely to present a proof of a particular program. (Figure 11 is significantly larger and fiddlier than Fig. 3b, which does not use variables-as-resource.) However, one seeking to explore potential optimisations, or to analyse the dependencies between various components of a program, should consider investing in variables-as-resource.

5   Tool support

Several properties of ribbon proofs make them a potentially appealing partner for automatic verification tools based on separation logic, such as *Bedrock* [6] and *VeriFast* [19]. Because ribbon proofs can be decomposed both horizontally and vertically, into independent proof blocks, they may suggest more opportunities for modular verification. One problem with automation is that users can lose track of their position in the proof: ribbons could provide an interface to the proof as it develops. Moreover, when automation fails, partial ribbon proofs could be used to view and guide the process manually. Ribbon proofs also shift the bureaucracy of rearranging assertions (in accordance with the associativity and commutativity of ∗) from the individual proof steps into the surrounding graphical structure, where it is more naturally handled.

To demonstrate the potential of ribbon proofs to complement automation, we have developed a prototype tool whose inputs are a ribbon diagram and a collection of small *Isabelle* proof scripts, one for each basic step. Our tool uses our *Isabelle* formalisation of Thm. 1 and the proof rules of Fig. 7 to assemble the proof scripts for the individual commands into a single script that verifies the entire diagram.

Supplied with appropriate proof rules for primitive commands and a collection of axioms about lists, our tool has successfully verified a number of small ribbon proofs, among them Fig. 3b. All of the proof scripts for the individual basic steps are small, and they can often be discharged without manual assistance. Individual proof scripts can be checked in any order – even concurrently. This feature recalls recent developments in theorem proving that allow proofs to be processed in a non-serial manner [32].

The input to the tool is a graphical ribbon diagram, following Defn. 5. Our tool begins by converting this graphical diagram into a stratified diagram, resolving any ambiguity about the node order by reference to the order of their input. (By taking this approach, we avoid having to invest in variables-as-resource.) It outputs a pictorial representation of the graph it has verified, laid out using the *dot* tool in the *Graphviz*
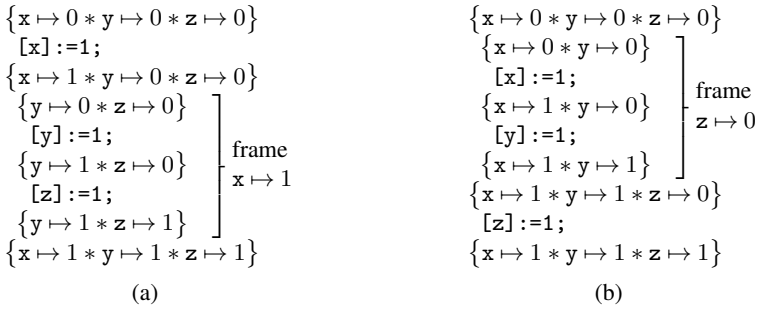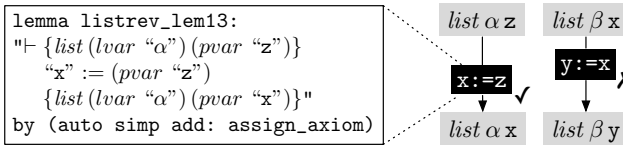
$$\{x \mapsto 0 * y \mapsto 0 * z \mapsto 0\}$$
$$[\mathtt{x}] := 1;$$
$$\{x \mapsto 1 * y \mapsto 0 * z \mapsto 0\}$$
$$\{y \mapsto 0 * z \mapsto 0\}$$
$$[\mathtt{y}] := 1;$$
$$\{y \mapsto 1 * z \mapsto 0\}$$
frame $x \mapsto 1$
$$[\mathtt{z}] := 1;$$
$$\{y \mapsto 1 * z \mapsto 1\}$$
$$\{x \mapsto 1 * y \mapsto 1 * z \mapsto 1\}$$

(a)

$$\{x \mapsto 0 * y \mapsto 0 * z \mapsto 0\}$$
$$\{x \mapsto 0 * y \mapsto 0\}$$
$$[\mathtt{x}] := 1;$$
$$\{x \mapsto 1 * y \mapsto 0\}$$
frame $z \mapsto 0$
$$[\mathtt{y}] := 1;$$
$$\{x \mapsto 1 * y \mapsto 1\}$$
$$\{x \mapsto 1 * y \mapsto 1 * z \mapsto 0\}$$
$$[\mathtt{z}] := 1;$$
$$\{x \mapsto 1 * y \mapsto 1 * z \mapsto 1\}$$

(b)

Fig. 12. Two alternatives to the proof outline in Fig. 1a

library. Clicking on any basic step loads the corresponding proof script, which can then be edited. When a step's proof is admitted by *Isabelle*, the corresponding node in the pictorial representation is marked with a tick; a failed or incomplete proof is marked with a cross. The picture below illustrates this on a snippet of Fig. 6b, and also shows the proof script for one of the steps.

```
lemma listrev_lem13:
"⊢ {list (lvar "α") (pvar "z")}
    "x" := (pvar "z")
    {list (lvar "α") (pvar "x")}"
by (auto simp add: assign_axiom)
```

$list\,\alpha\,\mathtt{z}$ $list\,\beta\,\mathtt{x}$

$\mathtt{x:=z}$ ✓ $\mathtt{y:=x}$ ✗

$list\,\alpha\,\mathtt{x}$ $list\,\beta\,\mathtt{y}$

In the current prototype, the user must supply the input in textual form, but in the future, we intend to enable direct interaction with the graphical representation, perhaps through a framework for diagrammatic reasoning such as *Diabelli* [30]. We envisage an interactive graphical interface for exploring and modifying proofs, that allows steps to be collapsed or expanded to the desired granularity – whether that is the fine details of every rule and axiom, or a coarse bird's-eye view of the overall structure of the proof.

The ribbon proofs in this paper have all been laid out manually (and we are preparing a public release of the LaTeX macros we use to do this) but there is scope for additional tool support for discovering pleasing layouts automatically.

6   Related and further work

Ribbon proofs are more than just a pretty syntax; they are a sound and complete proof system. Proof outlines have previously been promoted from a notational device to a formal system by Schneider [28], and by Ashcroft, who remarks that "the essential property of [proof outlines] is that each piece of program appears *once*" [1]. Very roughly speaking, ribbon proofs extend this property to each piece of assertion.

When constructing a proof outline, one can reduce the repetition by 'framing off' state that is unused for several instructions. For instance, Fig. 12a depicts one variation of Fig. 1a obtained by framing off x during the latter two instructions; another option is to frame off z during the first two (Fig. 12b). It is unsatisfactory that there are several

```
while true {
 x:=new();
 with buff when !full {
  full:=true;
    c:=x;
 }
}
```

(a) Code for 'producer' thread

```
while true {
 with buff when full {
  full:=false;
  y:=c;
 }
 dispose(y);
}
```

(b) Code for 'consumer' thread

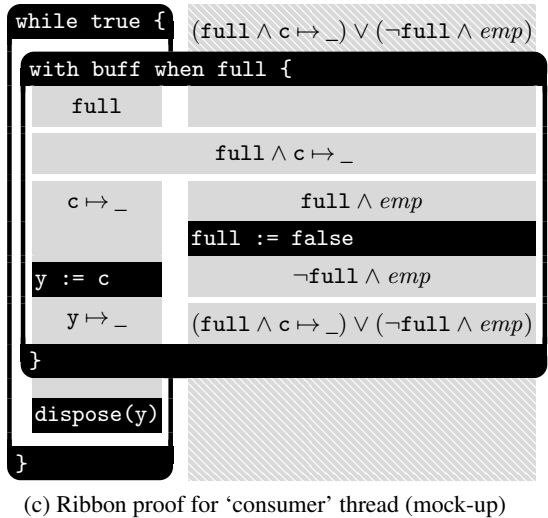
(c) Ribbon proof for 'consumer' thread (mock-up)

Fig. 13. Concurrency example: a single-cell buffer

different proof outlines for what is essentially the same proof. More pragmatically, deciding among these options can be difficult with large proof outlines. Happily, each of these options yields the same ribbon proof (Fig. 1b). We note a parallel here with *proof nets* [13], which are a graphical mechanism for unifying proofs in linear logic that differ only in uninteresting ways, such as the order of rule applications.

The graphical structures in Defn. 5 resemble Milner's *bigraphs* [22]. Assertions and commands are nodes, the deductions of the proof form the *link graph*, and existential boxes, choices and loops form the *place graph*. In fact, our diagrams correspond to *binding* bigraphs, in which links may not cross place boundaries. Relaxing this restriction may enable a model of the 'dynamic' scoping of existential boxes exhibited in Fig. 4, which our current formalisation dismisses as a purely syntactic artefact.

Ribbon proofs can be understood as objects of a symmetric monoidal category, and our pictures as *string diagrams*, which are widely used as graphical languages for such categories [29]. In future work we intend to investigate this categorical semantics of ribbon proofs; in particular, the use of *traces* [21] to model the loop construction depicted in Fig. 2a, and coproducts to model if-statements and existential boxes.

Another avenue for future work is the connection between ribbon proofs and Raza et al.'s *labelled separation logic* [26]. Labelled separation logic seeks to justify compiler reorderings by analysing the dependencies between program statements, and checking that these are not violated. The dependencies are detected by first labelling each component of each assertion with the commands that access it, and then propagating these labels through program proofs. Raza's labels recall the *columns* in our ribbon diagrams: each ribbon and each command occupies one or more columns of a diagram, and commands that occupy common columns (modulo twisting) may share a dependency.

We have so far considered only sequential programs, but our proofs have a distinctly concurrent flavour. It may be possible to extend ribbon proofs to *concurrent*

separation logic [23] as follows. Figure 13 gives a program (adapted from [23]) in which two threads communicate through a shared buffer at location c. The *resource invariant* $(\texttt{full} \wedge \texttt{c} \mapsto \_) \vee (\neg\texttt{full} \wedge emp)$ protected by the lock buff signifies that c is shared exactly when full is set. Figure 13c imagines a ribbon proof of the 'consumer' thread. The resource invariant is initially in a protected ribbon, inaccessible to the thread (as suggested by the hatching). Upon entering the critical region, the ribbon becomes available, and upon leaving it, the resource invariant is re-established and the ribbon is inaccessible once again.

Beyond concurrent separation logic, we intend to apply our system to more advanced separation logics. It has already aided the development of a logic for relaxed memory [5]; other candidates handle fine-grained concurrency [8, 10, 11, 31], dynamic threads [9], storable locks [14], loadable modules [20] and garbage collection [17]. Increasingly complicated logics for increasingly complicated programming features make techniques for intuitive construction and clear presentation ever more crucial.

7 Conclusion

Ribbon proofs are an attractive and practical approach for constructing and presenting proofs in separation logic or any derivative thereof. They contain less redundancy than a proof outline, and express the intent of the proof more clearly. Each step of the proof can be checked locally, by focusing only on the relevant resources. They are useful pedagogically for explaining how a simple proof is constructed, but also scale to more complex programs (as demonstrated in [33]), and have aided the development of a separation logic for relaxed memory [5]. They show graphically the distribution of resource in a program, and in particular, which parts of a program operate on disjoint resources, and this may prove useful for exploring parallelisation opportunities.

References

1. E. A. Ashcroft. Program verification tableaus. Technical Report CS-76-01, University of Waterloo, 1976.
2. J. Bean. *Ribbon Proofs - A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary University of London, 2006.
3. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL '05*. ACM Press, 2005.
4. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *MFPS XXI*, volume 155 of *ENTCS*, 2006.
5. R. Bornat and M. Dodds. Abducing barriers for Power and ARM. Draft, 2012.
6. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI '11*. ACM Press, 2011.

7. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *POPL '13*. ACM Press, 2013.
8. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP '10*, volume 6183 of *LNCS*. Springer, 2010.
9. M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP '09*, volume 5502 of *LNCS*. Springer, 2009.
10. X. Feng. Local rely-guarantee reasoning. In *POPL '09*. ACM Press, 2009.
11. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP '07*, volume 4421 of *LNCS*. Springer, 2007.
12. F. B. Fitch. *Symbolic Logic: An Introduction*. Ronald Press Co., 1952.
13. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50, 1987.
14. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS '07*, volume 4807 of *LNCS*. Springer, 2007.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.
16. C. A. R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1), 1971.
17. C.-K. Hur, D. Dreyer, and V. Vafeiadis. Separation logic in the presence of garbage collection. In *LICS '11*. IEEE Computer Society, 2011.
18. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01*. ACM Press, 2001.
19. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM '11*, volume 6617 of *LNCS*. Springer, 2011.
20. B. Jacobs, J. Smans, and F. Piessens. Verification of unloadable modules. In *FM '11*, volume 6664 of *LNCS*. Springer, 2011.
21. A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. of the Cambridge Philosophical Society*, 119(3), 1996.
22. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
23. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3), 2007.
24. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *B. Symb. Log.*, 5(2), 1999.
25. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6, 1976.
26. M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP '09*, volume 5502 of *LNCS*. Springer, 2009.
27. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*. IEEE Computer Society, 2002.
28. F. B. Schneider. *On Concurrent Programming*, chapter 4. Springer, 1997.
29. P. Selinger. A survey of graphical languages for monoidal categories. In *New Structures for Physics*, volume 813, chapter 4. Springer, 2011.
30. M. Urbas and M. Jamnik. Diabelli: A heterogeneous proof system. In *IJCAR '12*, volume 7364 of *LNCS*. Springer, 2012.
31. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR '07*, volume 4703 of *LNCS*. Springer, 2007.
32. M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In *UITP '10*, volume 285 of *ENTCS*, 2012.
33. J. Wickerson. *Concurrent Verification for Sequential Programs*. PhD thesis, University of Cambridge, 2013.
34. C. Wu, X. Zhang, and C. Urban. A formalisation of the Myhill-Nerode theorem based on regular expressions. In *ITP '11*, volume 6898 of *LNCS*. Springer, 2011.