Ribbon Proofs for Separation Logic

A verification pearl

John Wickerson
University of Cambridge

Mike Dodds
University of Cambridge

Matthew Parkinson
Microsoft Research Cambridge

Abstract

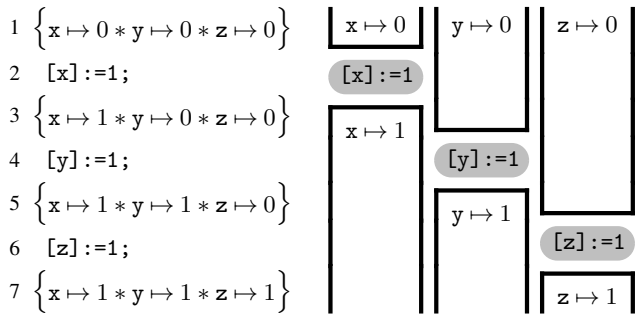
We present *ribbon proofs*, a diagrammatic proof system for separation logic. Inspired by an eponymous system due to Bean, ribbon proofs emphasise the structure of a proof, so are intelligible and hence useful pedagogically. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they are highly scalable (and we illustrate this with a ribbon proof of the Version 7 Unix memory manager). 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 that are readable, scalable, and easily modified.

A program proof in Hoare logic [14] 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, 15, 23, 27]) and its recent descendent, separation logic (e.g. [3–10, 13, 16–18, 26, 30]). 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; separation logic's $*$-operator specifies that these cells are distinct. First, there is much repetition: '$\mathtt{x} \mapsto 1$' appears three times. Second, it is difficult to interpret the effect of each instruction because there is 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. These are only minor problems in our toy example, but they quickly become devastating when scaled to larger programs.



(a) A proof outline (b) A ribbon proof

Figure 1. A simple example

The crux of the problem is what might be called *resource locality*. Separation logic [17, 26] 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 on 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 uses Bornat's method [5] to depict this mechanism for line 4 of Fig. 1a.

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

Showing such detail throughout a proof outline clarifies the effect of each instruction, but escalates the repetition. Cleverer use of the frame rule can help, but only a little – see §7. 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 still invoke 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 illuminate the proof structure, that can be navigated and modified by leveraging human visual intuition, and whose basic steps correspond exactly to separation logic's small axioms. The intuitive nature of ribbon proofs can be exploited most effectively by applying them to one of the many recent extensions of separation logic (e.g. [4, 7–10, 13, 16, 18, 21, 30]). 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* [11] to handle the propositional fragment of bunched implications logic (BI) [22]. BI being the basis of the assertion language used in separation logic [17], 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. Proof outlines focus on Hoare triples $\{p\}\, c\, \{q\}$, and often neglect the details of entailments between assertions, $p \Rightarrow q$, despite such entailments often encoding important insights about the program being verified. Our ribbon proofs treat both types of judgement equally, within the same system. Our system is further distinguished by its treatment of ribbon proofs as graphs, which gives our diagrams an appealing degree of flexibility.

Contributions We describe a diagrammatic proof system that enables a natural presentation of separation logic proofs. We prove it sound and (trivially) complete with respect to separation logic.

We describe a prototype tool for mechanically checking ribbon proofs in Isabelle (including several presented in this paper). 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.

Ribbon proofs are often bigger than their proof outline counterparts, but because they contain much less redundancy, they are actually a far more scalable proof representation. To illustrate the ability of our diagrams to present readable proofs of more complex programs, Appendix E presents a ribbon proof of the memory manager from Version 7 Unix (an example previously studied in [32]).

Outline

§2 describes our ribbon proof system through two examples.
§3 formally defines the two-dimensional language of ribbon diagrams and provides proof rules. We prove completeness with respect to separation logic. Soundness is non-trivial, and is attained by making one of the two following compromises.
§4 proves that our proof rules are sound if we sacrifice some flexibility of our diagrams by committing to a particular ordering of instructions.
§5 proves that our proof rules are sound if, instead, we adopt the *variables-as-resource* paradigm [24].
§6 describes our prototype tool for mechanically checking ribbon proofs in Isabelle.
§7 discusses related and future work, including extensions to handle concurrent separation logic [21] and possible applications to parallelisation.

```
1   {ls x 0 * ls y 0}
2   if (x==0) {
3     {ls y 0}
4     x:=y;
5     {ls x 0}
6   } else {
7     {ls x x * ls x 0 * x ≠̇ 0 * ls y 0}
8     t:=x;
9     {∃U. ls x t * t ↦ U * ls U 0 * ls y 0}
10    u:=[t];
11    while {ls x t * t ↦ u * ls u 0 * ls y 0} (u!=0) {
12      {ls x u * ls u 0 * u ≠̇ 0 * ls y 0}
13      t:=u;
14      {∃U. ls x t * t ↦ U * ls U 0 * ls y 0}
15      u:=[t];
16      {ls x t * t ↦ u * ls u 0 * ls y 0}
17    }
18    {ls x t * t ↦ 0 * ls y 0}
19    [t]:=y;
20    {ls x 0}
21  }
22  {ls x 0}
```

Figure 2. Proof outline for list append

2. Anatomy of a ribbon proof

We describe our ribbon proof system using two examples.

2.1 List append

Figure 2 presents a proof outline for an imperative program that appends two lists (adapted from [3]). It comprises (rather weak) pre- and postconditions, a loop invariant, and several intermediate assertions to guide the reader through the proof. For a binary relation r, we write $x\ \dot{r}\ y$ for $x\ r\ y \wedge emp$, where emp describes an empty heap. The ls predicate is the smallest satisfying:

$$ls\, x\, y \;\Leftrightarrow\; (x \doteq y \vee x \mathrel{\dot{\neq}} y * \exists x'.\, x \mapsto x' * ls\, x'\, y).$$

Despite the abundance of assertions, the proof outline obscures several features of the proof. For instance, the assertion at the entry to the else-branch (line 7) is potentially confusing because it differs in multiple ways from its predecessor on line 1: '$\mathtt{x} \mathrel{\dot{\neq}} 0$' has appeared, and so has '$ls\,\mathtt{x}\,\mathtt{x}$'. Only the former results from the failure of the test condition; the latter is from a lemma about ls. Likewise, in lines 8 and 13 we perform assignments while expanding the definition of ls, and in line 19 the heap update coincides with the use of an entailment lemma. This common practice of combining multiple proof steps avoids a proliferation of assertions, but comes at the expense of readability. (Displaying each step separately has problems too, as our next example shows.)

In contrast, the corresponding ribbon proof in Fig. 3 displays each proof step individually without resorting to repetition. The proof advances vertically, and the resources (memory cells) being operated upon are distributed horizontally. The precondition $ls\,\mathtt{x}\,0 * ls\,\mathtt{y}\,0$ is divided between two ribbons at the top of the diagram. That those assertions are connected via separation logic's $*$-operator means that they describe disjoint resources, and this is reflected in the diagram by the horizontal separation between the ribbons. Because $*$ is commutative, we can cross one ribbon over another – see Fig. 5b for an example of this 'twist' operation. Not only is the resource distribution unordered, it is not uniform, so the width of a

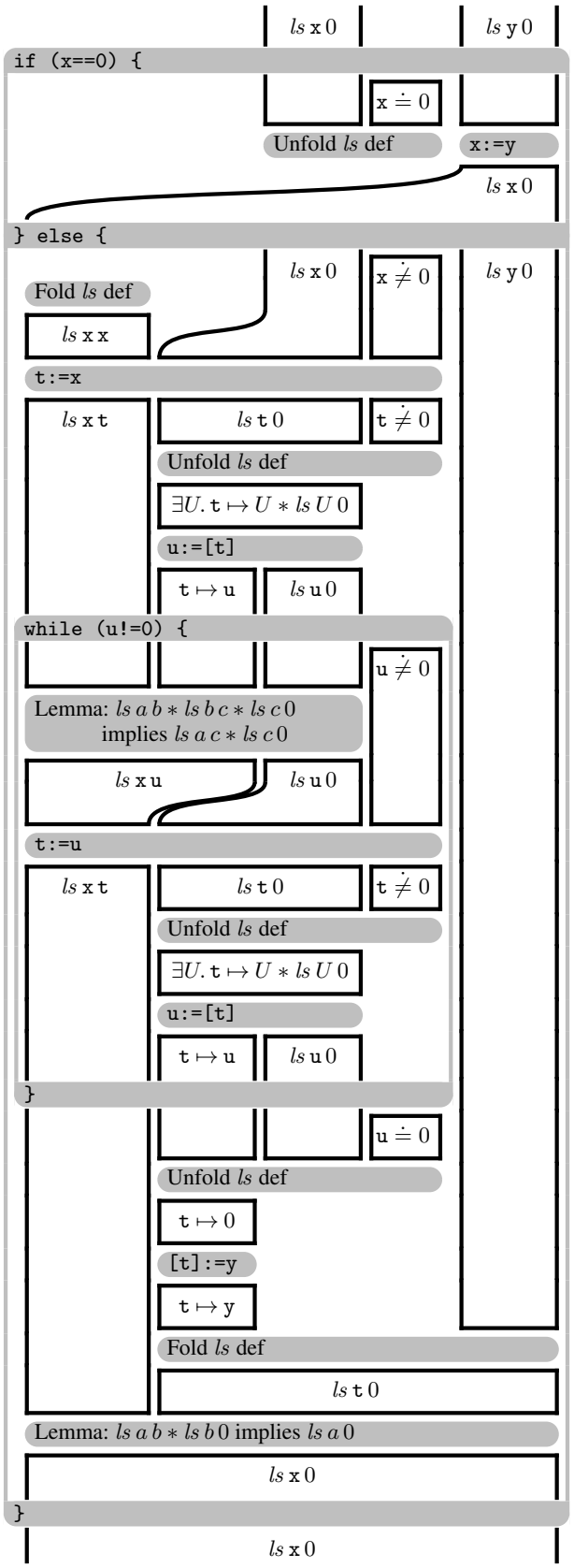


Figure 3. Ribbon proof of list append

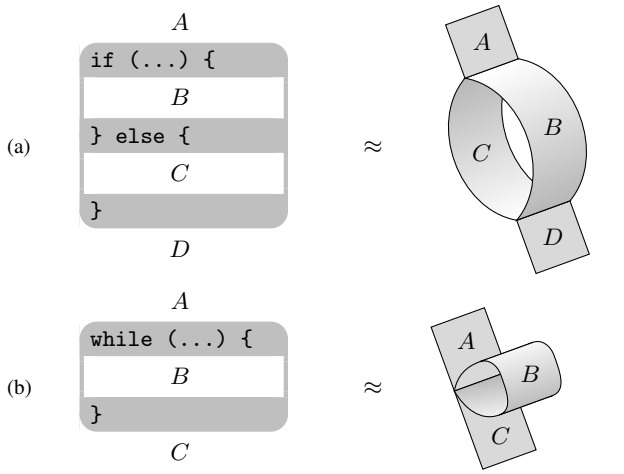


Figure 4. If-statements and while-loops, pictorially

ribbon is not proportional to the amount of resource it describes. In particular, the assertion '$\mathtt{x} \doteq 0$' obtained upon entering the then-branch describes no memory cells at all; it is merely a fact about variables. A gap in the diagram (e.g. above the 'fold' step at the start of the else-branch) corresponds to the 'emp' assertion.

Just above '$\mathtt{t}\!:=\!\mathtt{u}$' we stretch the '$ls\,\mathtt{x}\,\mathtt{u}$' and '$ls\,\mathtt{u}\,0$' ribbons so they align with the corresponding ribbons below the assignment. Such distortions are semantically meaningless but can aid readability. Similarly, at the end of the then-branch we stretch the '$ls\,\mathtt{x}\,0$' ribbon to mimic the ribbon at the end of the else-branch. The general rule is that the collection of ribbons entering the then-branch of an if-statement must match that entering the else-branch, as must the collections at the two exits, so that the proof could be cut and folded into the three-dimensional shape suggested in Fig. 4a.

The while-loop has a similar proof structure to the if-statement. Inside the loop body we assume that the test succeeds ($\mathtt{u} \neq \dot{} \ 0$); the complementary assumption appears after exiting the loop. The loop invariant is the collection of ribbons entering the top of the loop: $ls\,\mathtt{x}\,\mathtt{t}$, $\mathtt{t} \mapsto \mathtt{u}$ and $ls\,\mathtt{u}\,0$. 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. 4b.

In the else-branch, the assertion '$ls\,\mathtt{y}\,0$' is not needed until nearly the end, when it is merged with '$\mathtt{t} \mapsto \mathtt{y}$'. In a proof outline, this assertion would either be temporarily removed via an explicit application of the frame rule or, as is done in Fig. 2, redundantly repeated at every intermediate point. In the ribbon proof, it slides discreetly down the right-hand column. This indicates that the assertion is *inactive* without suggesting that it has been *removed*.

2.2 List reverse

Our second example provides a side-by-side comparison of a proof outline and a ribbon proof, and also explains how ribbon proofs handle existentially-quantified logical variables.

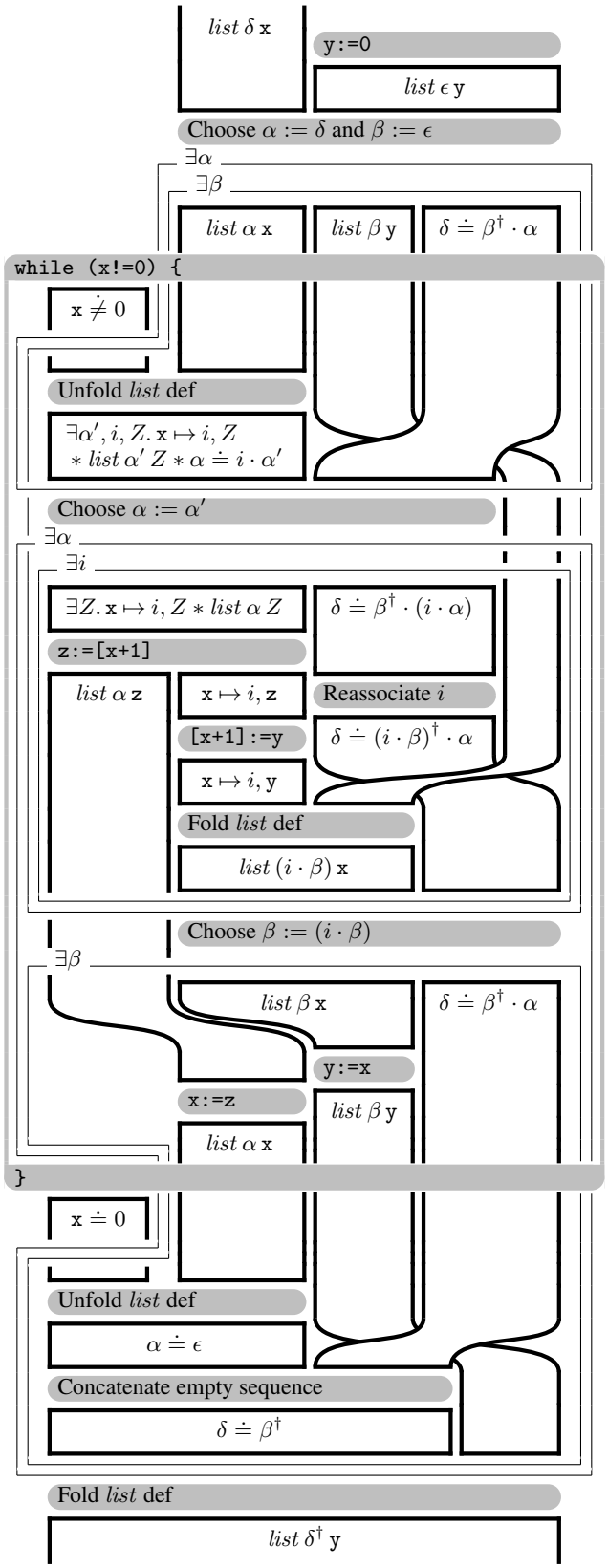
Figure 5a gives a proof outline of a program (adapted from [26]) for in-place reversal of a list. We write $\cdot$ for sequence concatenation, $(-)^\dagger$ for sequence reversal and ϵ for the empty sequence, and we define $list$ as the smallest predicate satisfying

$$\begin{aligned} list\,\alpha\,x \;\Leftrightarrow\; & x \doteq 0 * \alpha \doteq \epsilon \vee \\ & x \neq \dot{} \ 0 * \exists \alpha', i, x'.\, x \mapsto i, x' * \alpha \doteq i \cdot \alpha' * list\,\alpha'\,x'. \end{aligned}$$

In contrast to Fig. 2, this proof outline seeks to clarify the proof by making minimal changes between successive assertions. The cost of this is a large and highly redundant proof. And still the structure of the proof is unclear.

1 $\left\{ list\ \delta\ \mathtt{x} \right\}$
2 `y:=0;`
3 $\left\{ list\ \delta\ \mathtt{x} * list\ \epsilon\ \mathtt{y} \right\}$
4 `// Choose` $\alpha := \delta$ `and` $\beta := \epsilon$
5 `while` $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
6 `(x!=0) {`
7 $\left\{ \mathtt{x} \dot{\neq} 0 * (\exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha) \right\}$
8 $\left\{ \exists \alpha, \beta.\ \mathtt{x} \dot{\neq} 0 * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
9 `// Unfold` *list* `def`
10 $\left\{ \begin{array}{l} \exists \alpha, \beta.\ (\exists \alpha', i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha'\ \mathtt{z} * \alpha \doteq i \cdot \alpha') \\ * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \end{array} \right\}$
11 `// Choose` $\alpha := \alpha'$
12 $\left\{ \begin{array}{l} \exists \alpha, \beta, i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha\ Z * \delta \doteq \beta^\dagger \cdot (i \cdot \alpha) \\ * list\ \beta\ \mathtt{y} \end{array} \right\}$
13 `z:=[x+1];`
14 $\left\{ \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} * \delta \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y} \right\}$
15 `// Reassociate` i
16 $\left\{ \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y} \right\}$
17 `[x+1]:=y;`
18 $\left\{ \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{y} * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y} \right\}$
19 `// Fold` *list* `def`
20 $\left\{ \exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * list\ (i \cdot \beta)\ \mathtt{x} * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha \right\}$
21 `// Choose` $\beta := (i \cdot \beta)$
22 $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{z} * list\ \beta\ \mathtt{x} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
23 `y:=x;`
24 $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{z} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
25 `x:=z;`
26 $\left\{ \exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
27 `}`
28 $\left\{ \mathtt{x} \doteq 0 * (\exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha) \right\}$
29 $\left\{ \exists \alpha, \beta.\ \mathtt{x} \doteq 0 * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
30 `// Unfold` *list* `def`
31 $\left\{ \exists \alpha, \beta.\ \alpha \doteq \epsilon * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha \right\}$
32 `// Concatenate empty sequence`
33 $\left\{ \exists \beta.\ list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \right\}$
34 `// Fold` *list* `def`
35 $\left\{ list\ \delta^\dagger\ \mathtt{y} \right\}$

(a) A proof outline



(b) A ribbon proof

Figure 5. Two proofs of list reverse

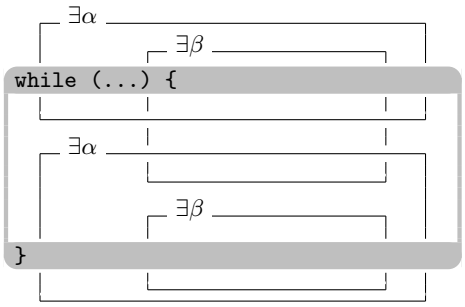


Figure 6. Vertical overlapping of existential boxes

$$\frac{\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\} \qquad wr(c) \not\frown rd(r)}{\vdash_{\mathsf{SL}} \{p * r\}\, c\, \{q * r\}} \qquad \frac{}{\vdash_{\mathsf{SL}} \{p\}\, \mathtt{skip}\, \{p\}}$$

$$\frac{\vdash_{\mathsf{SL}} \{p\}\, c_1\, \{q\} \qquad \vdash_{\mathsf{SL}} \{q\}\, c_2\, \{r\}}{\vdash_{\mathsf{SL}} \{p\}\, c_1 \,;\, c_2\, \{r\}} \qquad \frac{\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\}}{\vdash_{\mathsf{SL}} \{\exists x.\, p\}\, c\, \{\exists x.\, q\}}$$

$$\frac{\vdash_{\mathsf{SL}} \{p\}\, c_1\, \{q\} \qquad \vdash_{\mathsf{SL}} \{p\}\, c_2\, \{q\}}{\vdash_{\mathsf{SL}} \{p\}\, c_1 \ \mathtt{or}\ c_2\, \{q\}} \qquad \frac{\vdash_{\mathsf{SL}} \{p\}\, c\, \{p\}}{\vdash_{\mathsf{SL}} \{p\}\, \mathtt{loop}\ c\, \{p\}}$$

Figure 7. Proof rules for commands

In particular, the proof outline obscures the usage of the logical variables α and β. For instance, the α in line 12 is not the same as the α in line 5, though visually it seems to be. The witness for β is constant through lines 5 to 20, after which it becomes the previous β prepended with i. These subtle changes can only be spotted through careful examination of the proof outline (or else, as we have done, an explicit textual comment). The handling of logical variables in the ribbon proof is far more satisfactory. The scope of a logical variable is delimited by a thin *existential box*. Boxes extend horizontally across several ribbons, but also vertically to indicate the range of steps over which the same witness is used. We are permitted to stretch boxes horizontally – for instance, immediately below the loop in Fig. 5b. This corresponds to the implication $p * \exists x.\, q \;\Rightarrow\; \exists x.\, p * q$ (where x is not in p). Within any single row projected from the proof, existential boxes must be well-nested; this corresponds to the static scoping of existential quantifiers in assertions. Vertically, however, boxes may overlap; this corresponds to the implication $\exists x.\, \exists y.\, p \Rightarrow \exists y.\, \exists x.\, p$. Figure 6 depicts how the boxes for α and β overlap in Fig. 5b. We thus obtain an intriguing proof structure – present in neither the proof outline nor the derivation tree – in which the scopes of logical variables do not follow the program's syntactic structure, but are instead *dynamically* scoped. See §7 for further discussion.

We close this section by explaining a serious shortcoming in the proof system as currently presented. One nicety of Fig. 5b is that the 'Reassociate i' entailment is clearly independent of the neighbouring proof steps, being horizontally separated from them, and hence can be safely moved a little earlier or later. Close inspection is necessary to discover this from the proof outline. But by the same reasoning, the assignments '`y:=x`' and '`x:=z`' can be swapped, and this is unsound. This observation will cause difficulties in our formalisation (§3), but we shall overcome them, either by forbidding such manoeuvres altogether (§4) or by embedding information about variable dependencies into the ribbons by using the variables-as-resource paradigm (§5).

3. Formalisation

We now formalise the concepts introduced in the previous section. We introduce in §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 §3.2, plus additional rules in §3.3 for composing diagrams in sequence and in parallel. We relate ribbon proofs to separation logic in §3.4.

Notation. We tend to use letters in upper case or bold type to range over sets. For sets X and Y, we write $X \not\frown Y$ as shorthand for $X \cap Y = \emptyset$. Let $X \uplus Y$ be defined when $X \not\frown Y$ as $X \cup Y$. We sometimes treat a natural number k as the ordinal $\{0, \ldots, k-1\}$. Let $\mathcal{V}$ be an infinite set of node-identifiers. Proofs performed by hand are annotated with $\Box$, while those mechanically verified in Isabelle are annotated with Isabelle, 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, assumed to contain at least the following constructions:

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

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

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

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:

$$\mathtt{if}(b, c_1, c_2) \stackrel{\text{def}}{=} (\mathtt{assume}\, b \,;\, c_1)\ \mathtt{or}\ (\mathtt{assume}\, \neg b \,;\, c_2)$$
$$\mathtt{while}(b, c) \stackrel{\text{def}}{=} \mathtt{loop}(\mathtt{assume}\, b \,;\, c) \,;\, \mathtt{assume}\, \neg b.$$

We assume a separation logic over these commands and assertions, containing at least those rules given in Fig. 7. The first of these rules, the *frame rule*, employs the following definition.

Definition 3 (Reading and writing program variables)**.** The rd and wr functions extract the sets of program variables read and written. They can be applied to a variety of objects, such as assertions (which only read) and commands. We write $X \# Y$ when both $rd(X) \not\frown wr(Y)$ and $rd(Y) \not\frown wr(X)$.

Placing such minimal constraints on the form of commands, assertions and proof rules makes our formalisation applicable not just to separation logic but to any program logic based thereon.

3.1 Syntax of ribbon diagrams

We present a syntax that, with minor adjustments to be discussed shortly, can generate the pictures seen in the preceding section. The syntax is designed to be independent from the particular sizings and layout used in a picture. We begin with the concept of an *interface*, which roughly corresponds to a single row of a ribbon diagram.

Definition 4 (Interfaces)**.** An interface is either a single ribbon labelled with an assertion, an empty interface (shown as white-space in pictures), two interfaces side by side, or an existential box wrapped around an interface:

$$\mathsf{Interface} \stackrel{\text{def}}{=} \{P ::= \ \boxed{p} \ \mid\ \varepsilon \ \mid\ P\,P \ \mid\ \overset{\exists x}{\boxed{P}}\ \}.$$

Interfaces are identified up to $(P\,Q)\,R = P\,(Q\,R)$, $P\,\varepsilon = \varepsilon\,P = P$ and $P\,Q = Q\,P$. Where clarity demands it, we shall write $P \otimes Q$ instead of $P\,Q$, and hence $\otimes_{i \in I} P_i$ for iterated composition. There exists a straightforward mapping $asn : \mathsf{Interface} \to \mathsf{Assertion}$,

$$\mathsf{AsnGadget} \;=\; \{A ::= \boxed{p} \mid \boxed{\exists x\, G}\}$$

$$\mathsf{ComGadget} \;=\; \{C ::= c \mid \boxed{\begin{array}{c} G \\ \texttt{or} \\ G \end{array}} \mid \boxed{\begin{array}{c} \texttt{loop} \\ G \end{array}}\}$$

$$\begin{array}{rcl} \mathsf{Diagram} & = & \{G \mid V_G \subseteq_{\mathrm{fin}} \mathcal{V},\ \Lambda_G \in V_G \to \mathsf{AsnGadget}, \\ & & E_G \subseteq \mathcal{P}(V_G) \times \mathsf{ComGadget} \times \mathcal{P}(V_G), \\ & & \mathit{acyclic}(G), \text{ and } \mathit{linear}(G), \\ & & \text{where } G = (V_G, \Lambda_G, E_G)\} \end{array}$$

Figure 8. Syntax of ribbon diagrams

given by:

$$\begin{array}{lll} \mathit{asn}\,\boxed{p} = p & & \mathit{asn}\,(P\,Q) = \mathit{asn}\,P * \mathit{asn}\,Q \\ \mathit{asn}\,\varepsilon = \mathit{emp} & & \mathit{asn}\,\boxed{\exists x P} = \exists x.\, \mathit{asn}\,P. \end{array}$$

Next, we define a *diagram*, which can be thought of as a mapping between two interfaces.

Definition 5 (Diagrams, assertion-gadgets and command-gadgets)**.** The equations in Fig. 8 define a language of *diagrams*, *assertion-gadgets* and *command-gadgets*. 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. (Note that the latter contains a nested diagram because our pictures allow commands to reside within existential boxes.) 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 diagram $G \in \mathsf{Diagram}$ 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
- a set $E_G \subseteq \mathcal{P}(V_G) \times \mathsf{ComGadget} \times \mathcal{P}(V_G)$ of *hyperedges* $(\mathbf{v}, C, \mathbf{w})$, each of which comprises 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 $v \rightarrowtail w$ if $v \in \mathbf{v}$ and $w \in \mathbf{w}$ for some $(\mathbf{v}, C, \mathbf{w}) \in E_G$. Then $\mathit{acyclic}(G)$ iff $v \rightarrowtail^i v$ implies $i = 0$.

LINEARITY: Define $\mathit{linear}(G)$ to hold iff the hyperedges in E_G have no common heads and no common tails.

Linearity models the fact that ribbons cannot be duplicated, which in turn is a result of $p \Rightarrow p * p$ being invalid in separation logic.

Remark 6. The diagrams that we present above are nested, directed, acyclic graphs; that is, dags whose nodes and edges may be labelled with further dags. They might typically be presented as a single graph, with dedicated 'parent' edges to simulate the nesting hierarchy. However, as Wu et al. observe, "reasoning about graphs [...] can be a real hassle in HOL-based theorem provers" [33]. Hence, with our Isabelle formalisation in mind, we prefer to use a (mutually) inductive datatype to depict the hierarchy (although each level of the hierarchy must remain non-inductive).

Remark 7. We usually work with *abstract diagrams*. These diagrams are identified up to graph isomorphism; that is, the particular choice of node-identifiers is unimportant. In particular, the diagrams that appear within assertion-gadgets or command-gadgets are treated abstractly. However, some definitions and proofs work with *concrete diagrams* where the node-identifiers are exposed.

Having presented the syntax of ribbon diagrams, we now show how it corresponds to the pictures presented in the previous section. Figure 9 presents the term of Diagram that corresponds to Fig. 5b. The discrepancies that remain are the result of our pictures using the following pieces of 'syntactic sugar':

- entailments are performed using basic steps whose command is `skip` – rather than write '`skip`', we label the step with a justification of the entailment;
- the ribbons and existential boxes at the top and bottom of a diagram are left visually 'open' to suggest the potential for connections to other diagrams;
- repeated labels on ribbons or existential boxes are removed; and
- for existential boxes, the operations of extending, contracting and commuting are really the entailments depicted informally in Fig. 10. When such entailments appear in Fig. 9, they are displayed like so: ●. Having to show these entailments explicitly makes Fig. 9 significantly more repetitive than Fig. 5b. See §7 for further discussion of this issue.

3.2 Proof rules for ribbon 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.

Definition 8 (Initial and terminal nodes)**.** We shall be interested in those nodes of a diagram G that have no outgoing or no incoming edges:

$$\begin{array}{rcl} \mathit{initials}(G) & = & V_G \setminus \bigcup\{\mathbf{v} \mid (\_, \_, \mathbf{v}) \in E_G\} \\ \mathit{terminals}(G) & = & V_G \setminus \bigcup\{\mathbf{v} \mid (\mathbf{v}, \_, \_) \in E_G\}. \end{array}$$

Definition 9 (Top and bottom interfaces)**.** These constructions are used to extract interfaces from diagrams. For a diagram G:

$$\begin{array}{rcl} \mathit{top}(G) & = & \otimes_{v \in \mathit{initials}\,G}\, \mathit{top}(\Lambda_G\, v) \\ \mathit{bot}(G) & = & \otimes_{v \in \mathit{terminals}\,G}\, \mathit{bot}(\Lambda_G\, v) \end{array}$$

and for assertion-gadgets:

$$\begin{array}{lll} \mathit{top}\,\boxed{p} = \lfloor p \rfloor & & \mathit{top}\,\boxed{\exists x G} = \lfloor \exists x\, \mathit{top}\, G \rfloor \\ \mathit{bot}\,\boxed{p} = \lfloor p \rfloor & & \mathit{bot}\,\boxed{\exists x G} = \lfloor \exists x\, \mathit{bot}\, G \rfloor. \end{array}$$

Figure 11 defines proof rules for diagrams, command-gadgets and assertion-gadgets. The judgement $\vdash^{\mathsf{dia}}_{\mathsf{SL}} G : P \to Q$ means that the diagram G, precondition P, and postcondition Q form a valid proof. The interfaces P and Q are always equal to $\mathit{top}(G)$ and $\mathit{bot}(G)$ respectively, so we sometimes omit them. The judgements for command-gadgets and assertion-gadgets are similar, the latter without interfaces.

The MAIN rule embodies the 'locally checkable' nature of ribbon proofs. It says that in order to check an entire proof, one need only check each assertion-gadget (first antecedant) and each command-gadget (second antecedant) in isolation, recursing inside those gadgets as required.

The BASIC rule corresponds to an ordinary separation logic judgement $\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\}$. Note that this judgement may be arbitrarily complex, and hence that 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 there are certain

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

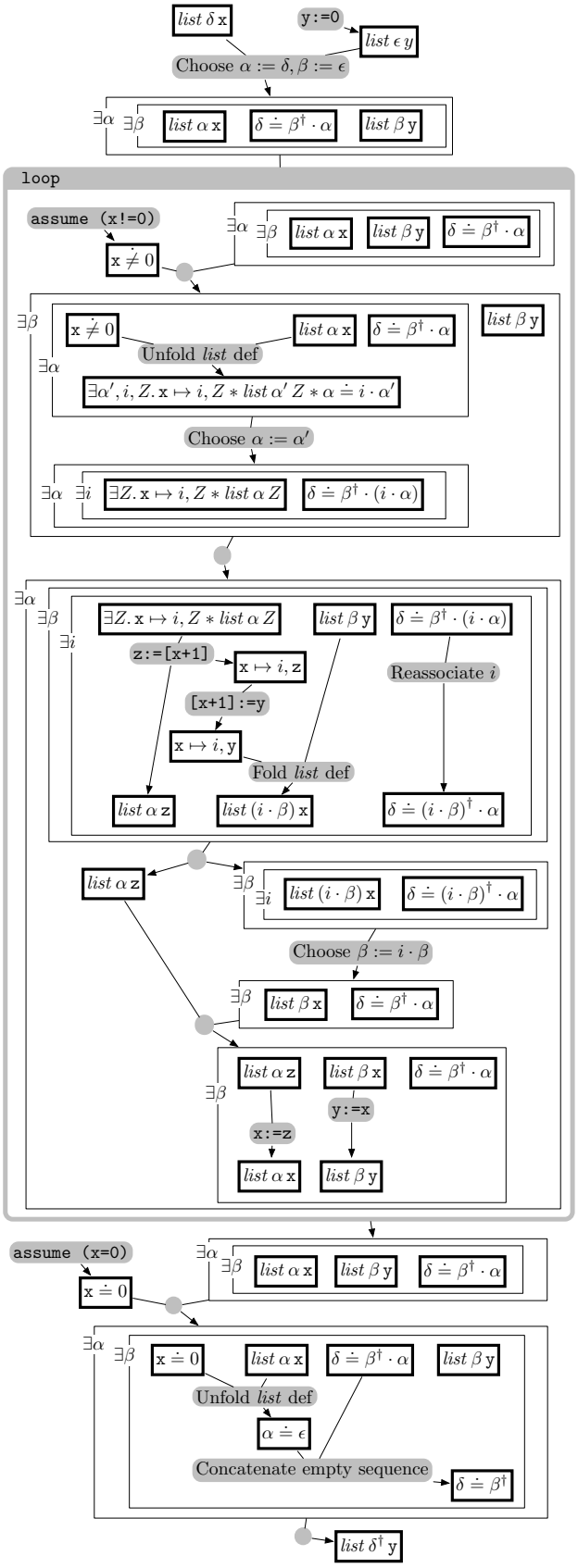


Figure 9. 'De-sugared' ribbon proof of list reverse

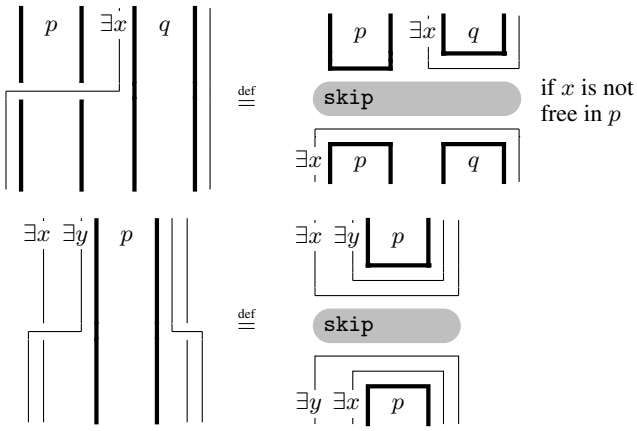


Figure 10. Syntactic sugar for existential boxes

$$\textsc{Ribbon}\ \frac{}{\vdash^{\text{asn}}_{\text{SL}} \boxed{p}} \qquad \textsc{Basic}\ \frac{\vdash_{\text{SL}} \{asn\ P\}\ c\ \{asn\ Q\}}{\vdash^{\text{com}}_{\text{SL}} c : P \to Q} \qquad \textsc{Exists}\ \frac{\vdash^{\text{dia}}_{\text{SL}} G}{\vdash^{\text{asn}}_{\text{SL}} \exists x \boxed{G}}$$

$$\textsc{Choice}\ \frac{\vdash^{\text{dia}}_{\text{SL}} G_1 : P \to Q \quad \vdash^{\text{dia}}_{\text{SL}} G_2 : P \to Q}{\vdash^{\text{com}}_{\text{SL}} \boxed{\begin{array}{c} G_1 \\ \texttt{or} \\ G_2 \end{array}} : P \to Q} \qquad \textsc{Loop}\ \frac{\vdash^{\text{dia}}_{\text{SL}} G : P \to P}{\vdash^{\text{com}}_{\text{SL}} \boxed{\begin{array}{c} \texttt{loop} \\ G \end{array}} : P \to P}$$

$$\textsc{Main}\ \frac{\begin{array}{c} \forall v \in V_G . \vdash^{\text{asn}}_{\text{SL}} \Lambda_G\, v \\ \forall (\mathbf{v}, C, \mathbf{w}) \in E_G . \vdash^{\text{com}}_{\text{SL}} C : \otimes_{v \in \mathbf{v}}\, bot(\Lambda_G\, v) \to \otimes_{w \in \mathbf{w}}\, top(\Lambda_G\, w) \end{array}}{\vdash^{\text{dia}}_{\text{SL}} G : top(G) \to bot(G)}$$

Figure 11. Proof rules for diagrams

cases where this is not desirable. For instance, although '`while true do skip`' could be proved using a full-blown loop diagram, one basic step may provide sufficient detail. A ribbon diagram can thus be viewed as a flexible combination of diagrammatic and traditional proofs, with the BASIC rule as the interface between the two levels. By invoking the BASIC rule right at the root of the proof tree, we obtain a trivial completeness result (see Thm. 15).

3.3 Composing ribbon diagrams

The proof rules already presented provide only limited mechanisms for building new diagrams from old. We can wrap diagrams in existential boxes, or put them inside choice or loop diagrams, but the rules do not describe how to stack two diagrams vertically, or place them side by side. In this subsection we derive two additional proof rules for composing diagrams.

Definition 10 (Sequential composition of diagrams)**.** We notate sequential composition by vertical stacking. We overload this notation for both diagrams and assertion-gadgets. If G and H are diagrams that satisfy:

- $terminals\ G = initials\ H = V_G \cap V_H$, and
- $\frac{\Lambda_G(v)}{\Lambda_H(v)}$ is defined for all $v \in V_G \cap V_H$

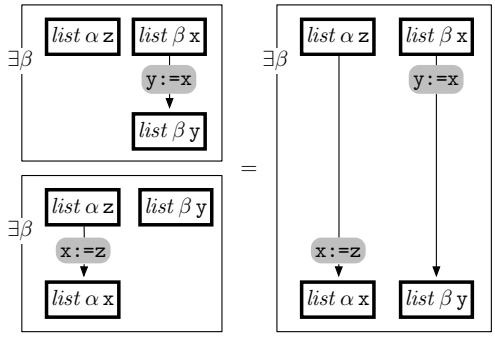


Figure 12. An example of sequential composition

then we write $\begin{smallmatrix} G \\ H \end{smallmatrix}$ for the diagram $(V_G \cup V_H, \Lambda, E_G \cup E_H)$, where

$$\Lambda(v) = \begin{cases} \Lambda_G(v) & \text{if } v \in V_G \setminus V_H \\ \Lambda_H(v) & \text{if } v \in V_H \setminus V_G \\ \begin{pmatrix} \Lambda_G(v) \\ \Lambda_H(v) \end{pmatrix} & \text{if } v \in V_G \cap V_H. \end{cases}$$

Simultaneously, sequential composition on assertion-gadgets is (partially) defined as follows:

$$\boxed{\begin{smallmatrix} p \\ p \end{smallmatrix}} = \boxed{p} \qquad \boxed{\begin{smallmatrix} \exists x G \\ \exists x H \end{smallmatrix}} = \boxed{\exists x \begin{smallmatrix} G \\ H \end{smallmatrix}} \text{ provided } \begin{smallmatrix} G \\ H \end{smallmatrix} \text{ is defined.}$$

Example 11. The definition above appears fiddly, but it becomes natural once the diagrams are drawn. Figure 12 shows how two diagrams, each comprising a single existential box around three ribbons and one basic step, can be sequentially composed.

Definition 12 (Parallel composition of diagrams)**.** If G and H are diagrams with disjoint sets of node-identifiers, then we write $G \parallel H$ for the diagram $(V_G \cup V_H, \Lambda_G \cup \Lambda_H, E_G \cup E_H)$.

Theorem 13. *The following rules are derivable from those in Fig. 11.*

$$\textsc{Seq}\ \frac{\vdash^{\text{dia}}_{\mathsf{SL}} G : P \to Q \quad \vdash^{\text{dia}}_{\mathsf{SL}} H : Q \to R}{\vdash^{\text{dia}}_{\mathsf{SL}} \begin{smallmatrix} G \\ H \end{smallmatrix} : P \to R} \qquad \textsc{Par}\ \frac{\vdash^{\text{dia}}_{\mathsf{SL}} G : P \to Q \quad \vdash^{\text{dia}}_{\mathsf{SL}} H : P' \to Q'}{\vdash^{\text{dia}}_{\mathsf{SL}} G \parallel H : P\,P' \to Q\,Q'}$$

Proof. See Appx. A. □

3.4 Semantics of ribbon diagrams

Since our diagrams have a parallel nature, but our language is only sequential, it follows that each diagram proves not a single command, but a set of commands, each being one possible linear extension of the partial ordering imposed by the diagram. The *coms* function defined in Fig. 13 is responsible for extracting this set from a given diagram. Each command is obtained by picking a valid order of command- and assertion-gadgets (using the *lin* function defined below), recursively extracting a command from each gadget, and then sequentially composing the results.

Definition 14 (Linear extensions)**.** For a diagram G, we define $\mathit{lin}\, G$ as the set of all lists $[x_0, \ldots, x_{k-1}]$ of AsnGadgets and ComGadgets, where there exists a bijection $\pi : k \to V_G \uplus E_G$

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

$$\mathit{coms}\ \boxed{c} = \{c\} \qquad \mathit{coms}\ \boxed{\begin{smallmatrix} \mathtt{loop} \\ G \end{smallmatrix}} = \{\mathtt{loop}\, c \mid c \in \mathit{coms}\, G\}$$

$$\mathit{coms}\ \boxed{\begin{smallmatrix} G_1 \\ \mathtt{or} \\ G_2 \end{smallmatrix}} = \{c_1 \,\mathtt{or}\, c_2 \mid c_1 \in \mathit{coms}\, G_1,\ c_2 \in \mathit{coms}\, G_2\} \qquad \mathit{coms}\ \boxed{p} = \{\mathtt{skip}\}$$

$$\mathit{coms}\ \boxed{\exists x G} = \mathit{coms}\, G.$$

Figure 13. Extracting commands from a diagram

that satisfies, for all $(\mathbf{v}, C, \mathbf{w}) \in E_G$:

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

and where, for all $i \in k$:

$$x_i = \begin{cases} \Lambda_G(v) & \text{if } \pi(i) = v \\ C & \text{if } \pi(i) = (\mathbf{v}, C, \mathbf{w}). \end{cases}$$

By ACYCLICITY, all diagrams admit at least one linear extension.

Theorem 15 (Completeness)**.** *Any separation logic proof can be recreated as a ribbon diagram.*

$$\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\} \implies (\exists G, P, Q.\, c \in \mathit{coms}\, G \wedge p = \mathit{asn}\, P \wedge q = \mathit{asn}\, Q \wedge \vdash^{\text{dia}}_{\mathsf{SL}} G : P \to Q)$$

Proof. Choose $P = \boxed{p}$, $Q = \boxed{q}$, and $G =$

$$(\{v_1, v_2\}, \{v_1 \mapsto \boxed{p}, v_2 \mapsto \boxed{q}\}, \{(\{v_1\}, \boxed{c}, \{v_2\})\})$$

for some $v_1 \neq v_2 \in \mathcal{V}$. Isabelle

Although completeness is trivial, soundness is trickier. In fact, the system presented so far is unsound.

Non-Theorem 16 (Soundness)**.** *Any ribbon diagram can be recreated as a separation logic proof:*

$$\vdash^{\text{dia}}_{\mathsf{SL}} G : P \to Q \implies \forall c \in \mathit{coms}\, G.\, \vdash_{\mathsf{SL}} \{\mathit{asn}\, P\}\, c\, \{\mathit{asn}\, Q\}.$$

Counterexample. Consider the diagram on the right-hand side of Fig. 12. It provides a proof of two Hoare triples,

$$\vdash_{\mathsf{SL}} \{\exists \beta.\, \mathit{list}\, \alpha\, \mathtt{z} * \mathit{list}\, \beta\, \mathtt{x}\}\ \mathtt{y:=x}\ ;\ \mathtt{x:=z}\ \{\exists \beta.\, \mathit{list}\, \alpha\, \mathtt{x} * \mathit{list}\, \beta\, \mathtt{y}\}$$
$$\vdash_{\mathsf{SL}} \{\exists \beta.\, \mathit{list}\, \alpha\, \mathtt{z} * \mathit{list}\, \beta\, \mathtt{x}\}\ \mathtt{x:=z}\ ;\ \mathtt{y:=x}\ \{\exists \beta.\, \mathit{list}\, \alpha\, \mathtt{x} * \mathit{list}\, \beta\, \mathtt{y}\}$$

the second of which is invalid. □

The problem is that our diagrams do not take into account dependencies on program variables. There are two ways to salvage this situation, which we explore in the next two sections. The first is to use what we call *rasterisation* to limit the possible linear extensions of a diagram, and the second is to employ variables-as-resource.

4. Rasterisation

Our first proposal for attaining soundness involves making the ordering of the proof steps explicit; that is, committing to one particular chain of assertion- and command-gadgets. We achieve this by modifying Defn. 5 as follows.

$$com[(\gamma_0, F_0), \ldots, (\gamma_k, F_k)] = com\,\gamma_0 \;;\cdots;\; com\,\gamma_k$$

$$com\,P = \texttt{skip} \qquad com \begin{array}{c} P \\ \hline c \\ \hline Q \end{array} = c \qquad com\ \exists x D| = com\,D$$

$$com \begin{array}{|c|} \hline P \\ \hline D \\ \hline \texttt{or} \\ \hline E \\ \hline Q \\ \hline \end{array} = (com\,D)\ \texttt{or}\ (com\,E) \qquad com \begin{array}{|c|} \hline P \\ \hline \texttt{loop} \\ \hline D \\ \hline Q \\ \hline \end{array} = \texttt{loop}(com\,D)$$

Figure 14. Extracting a command from a rasterised diagram

Definition 17 (Ordered diagrams)**.** The set OrderedDiagram of *ordered diagrams* is defined in the same way as Diagram (Defn. 5), except we use natural numbers (rather than elements of $\mathcal{V}$) as node-identifiers, and have E_G as a *list* of hyperedges rather than an unordered set. This list must be consistent with the proof; that is, if $(\mathbf{v}, C, \mathbf{w})$ precedes $(\mathbf{v}', C', \mathbf{w}')$ in E_G, and $w' \in \mathbf{w}'$ and $v \in \mathbf{v}$, then there must not exist a chain $w' \rightarrow^* v$.

By reinterpreting Fig. 9 so that node positions are meaningful, it can be seen as an ordered diagram. That is, suppose that the assertion-gadgets are numbered from top to bottom, then from left to right, and that the command-gadgets are ordered by their vertical position. To achieve soundness, we need to check that whenever a command writes to a program variable, that variable does not appear in any concurrent ribbons. Adjusting the proof rules of Fig. 11 to accommodate this is possible, but fiddly. Instead, we observe that having imposed a total order on our assertion- and command-gadgets, our graphical syntax becomes mostly superfluous, and we can reinterpret our pictures as terms in the following, simpler language of *rasterised diagrams*. (The details of the conversion are given in Appx. D.)

Definition 18 (Rasterised diagrams)**.** Let D range over the set RDiagram of rasterised diagrams. A rasterised diagram is a non-empty list, written as

$$\begin{array}{c}(\gamma_0, F_0) \\ \cdots \\ (\gamma_k, F_k)\end{array} \qquad \text{or} \qquad [(\gamma_0, F_0), \ldots, (\gamma_k, F_k)],$$

of *cells* $\gamma_0, \ldots, \gamma_k \in$ Cell and *frames* $F_0, \ldots, F_k \in$ Interface. The syntax of cells is as follows:

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

The *top* and *bot* functions are straightforwardly amended for rasterised diagrams and cells. Where ordinary diagrams may admit multiple commands, rasterised diagrams admit only one; this is extracted by the *com* function defined in Fig. 14.

The rules given in Fig. 15 define the provability of cells and rasterised diagrams. For composing rasterised diagrams, it is possible to derive proof rules in the spirit of those in Thm. 13.

Definition 19 (Parallel composition of rasterised diagrams)**.** If D and E are rasterised diagrams of lengths m and n, and μ is a binary sequence containing m zeroes and n ones, then we define the parallel composition of D and E according to μ as follows:

$$D \parallel_\mu E \quad \stackrel{\text{def}}{=} \quad zip_\mu(D, E, top\,D, top\,E)$$

$$\textsc{RRibbon}\ \frac{}{\vdash^{\text{cell}}_{\mathsf{SL}} (P, F) : P\,F \to P\,F} \qquad \textsc{RExists}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \qquad wr\,D \not\frown rd\,F}{\vdash^{\text{cell}}_{\mathsf{SL}} (\exists x D|, F) : \exists x P|F \to \exists x Q|F}$$

$$\textsc{RBasic}\ \frac{\vdash_{\mathsf{SL}} \{asn\,P\}\,c\,\{asn\,Q\} \qquad wr\,c \not\frown rd\,F}{\vdash^{\text{cell}}_{\mathsf{SL}} (\begin{array}{c} P \\ \hline c \\ \hline Q \end{array}, F) : P\,F \to Q\,F}$$

$$\textsc{RLoop}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to P \quad wr\,D \not\frown rd\,F}{\vdash^{\text{cell}}_{\mathsf{SL}} (\begin{array}{|c|} \hline P \\ \hline \texttt{loop} \\ \hline D \\ \hline P \\ \hline \end{array}, F) : P\,F \to P\,F} \qquad \textsc{RChoice}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \quad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P \to Q \quad (wr\,D \cup wr\,E) \not\frown rd\,F}{\vdash^{\text{cell}}_{\mathsf{SL}} (\begin{array}{|c|} \hline P \\ \hline D \\ \hline \texttt{or} \\ \hline E \\ \hline Q \\ \hline \end{array}, F) : P\,F \to Q\,F}$$

$$\textsc{RMain}\ \frac{\forall i \in k+1.\, \vdash^{\text{cell}}_{\mathsf{SL}} (\gamma_i, F_i) : P_i \to P_{i+1}}{\vdash^{\text{rdia}}_{\mathsf{SL}} [(\gamma_0, F_0), \ldots, (\gamma_k, F_k)] : P_0 \to P_{k+1}}$$

Figure 15. Proof rules for rasterised diagrams

where *zip* is defined inductively as follows:

$$\begin{array}{rcl} zip_\epsilon(\_,\_,\_,\_) & = & [\,] \\ zip_{0\mu}((\gamma, F) :: D, E, \_, Q) & = & (\gamma, F \otimes Q) :: \\ & & zip_\mu(D, E, bot\,\gamma \otimes F, Q) \\ zip_{1\mu}(D, (\gamma, F) :: E, P, \_) & = & (\gamma, P \otimes F) :: \\ & & zip_\mu(D, E, P, bot\,\gamma \otimes F). \end{array}$$

The construction given above is far more intuitive when interpreted visually. The appropriate μ is simply determined by the order in which the commands in the two operands vertically interleave.

Example 20. One way to obtain the ribbon diagram in Fig. 1b (in rasterised form) is to compose its first two columns in parallel with its third, as follows:

$$\left(\begin{array}{c} |\texttt{x} \mapsto 0| \\ \texttt{[x]:=1}, |\texttt{y} \mapsto 0| \\ |\texttt{x} \mapsto 1| \end{array}\right) \left(\begin{array}{c} |\texttt{y} \mapsto 0| \\ \texttt{[y]:=1}, |\texttt{x} \mapsto 1| \\ |\texttt{y} \mapsto 1| \end{array}\right) \parallel_{001} \left(\begin{array}{c} |\texttt{z} \mapsto 0| \\ \texttt{[z]:=1}, \varepsilon \\ |\texttt{z} \mapsto 1| \end{array}\right) = \left(\begin{array}{c} |\texttt{x} \mapsto 0| \\ \texttt{[x]:=1}, |\texttt{y} \mapsto 0|\,|\texttt{z} \mapsto 0| \\ |\texttt{x} \mapsto 1| \end{array}\right) \left(\begin{array}{c} |\texttt{y} \mapsto 0| \\ \texttt{[y]:=1}, |\texttt{x} \mapsto 1|\,|\texttt{z} \mapsto 0| \\ |\texttt{y} \mapsto 1| \end{array}\right) \left(\begin{array}{c} |\texttt{z} \mapsto 0| \\ \texttt{[z]:=1}, |\texttt{x} \mapsto 1|\,|\texttt{y} \mapsto 1| \\ |\texttt{z} \mapsto 1| \end{array}\right)$$

Theorem 21. *The following rules are derivable from those in Fig. 15.*

$$\textsc{RSeq}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \quad \vdash^{\text{rdia}}_{\mathsf{SL}} E : Q \to R}{\vdash^{\text{rdia}}_{\mathsf{SL}} \begin{array}{c} D \\ E \end{array} : P \to R} \qquad \textsc{RPar}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \quad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P' \to Q' \qquad D \,\#\, E}{\vdash^{\text{rdia}}_{\mathsf{SL}} D \parallel_\mu E : P\,P' \to Q\,Q'}$$

Proof. See Appx. B. □

Theorem 22 (Soundness – rasterised)**.** *Any rasterised ribbon diagram can be recreated as a separation logic proof.*

$$\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \implies \vdash_{\mathsf{SL}} \{asn\ P\}\ com\ D\ \{asn\ Q\}.$$

Proof. By mutual rule induction. Isabelle

5. Variables-as-resource

Rasterisation sacrifices much of the flexibility of our ribbon diagrams. It is often sound to tweak the layout of a diagram by sliding steps up or down or reordering ribbons, but rasterisation rules out all such manoeuvres. In this section, we explain how the *variables-as-resource* paradigm [24] can be used to obtain soundness while preserving the graphical nature of diagrams.

The variables-as-resource paradigm 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 supplants the use of rd and wr sets in Fig. 7, so these can henceforth be assumed empty. The counterexample to Thm. 16 is no longer admissible, because neither assignment in the diagram holds the necessary resource in its precondition.

Thanks to the generality of our formalisation, the introduction of variables-as-resource does not contradict any previous results. We simply choose appropriate axioms and primitive commands, and amend the BASIC rule to use $\vdash_{\mathsf{VaR}}$ rather than $\vdash_{\mathsf{SL}}$.

Theorem 23 (Soundness – variables-as-resource)**.** *Using variables-as-resource, any ribbon diagram can be recreated as a separation logic proof:*

$$\vdash^{\text{dia}}_{\mathsf{VaR}} G : P \to Q \implies \forall c \in coms\, G. \vdash_{\mathsf{VaR}} \{asn\ P\}\ c\ \{asn\ Q\}.$$

Proof. See Appx. C. Isabelle

Figure 16 exhibits a ribbon proof, conducted using variables-as-resource, of the list-reversal program from §2.2. Variables-as-resource dictates that every assertion in the proof is accompanied by one Own predicate for each program variable it mentions. For instance, the precondition $list\,\delta\,\mathtt{x}$ is paired with some of `x`'s resource. The shading is merely syntactic sugar; for instance:

$$\left|\mathtt{x}, \tfrac{1}{2}\mathtt{y}\ \ \mathtt{x} \mapsto i, \mathtt{y}\right| \quad \stackrel{\text{def}}{=} \quad \left|Own_1(\mathtt{x}) * Own_{.5}(\mathtt{y}) * \mathtt{x} \mapsto i, \mathtt{y}\right|.$$

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 16 introduces a couple of novel features: ribbons may pass 'underneath' basic steps to reduce the need for twisting (see e.g. the 'Choose $\alpha := \delta$ and $\beta := \epsilon$' step), and horizontal space is conserved by writing some assertions sideways. The diagram can be laid out in several ways, unconstrained by the rasterisation 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. 5b, that it misleadingly suggested that '`y:=x`' and '`x:=z`' could be safely permuted. Figure 16 forbids this by showing the dependency on '`x`'. On the other hand, both figures agree that the 'Reassociate i' step can be safely manoeuvred up or down a little.

In this section and the previous one, we have presented two alternative formalisations of ribbon diagrams. We remark that one who seeks merely to present a proof of a particular program need not use variables-as-resource; the splitting, distributing, and recombining of the resource associated with each variable is an unnecessary burden. Figure 16 is significantly larger and fiddlier than

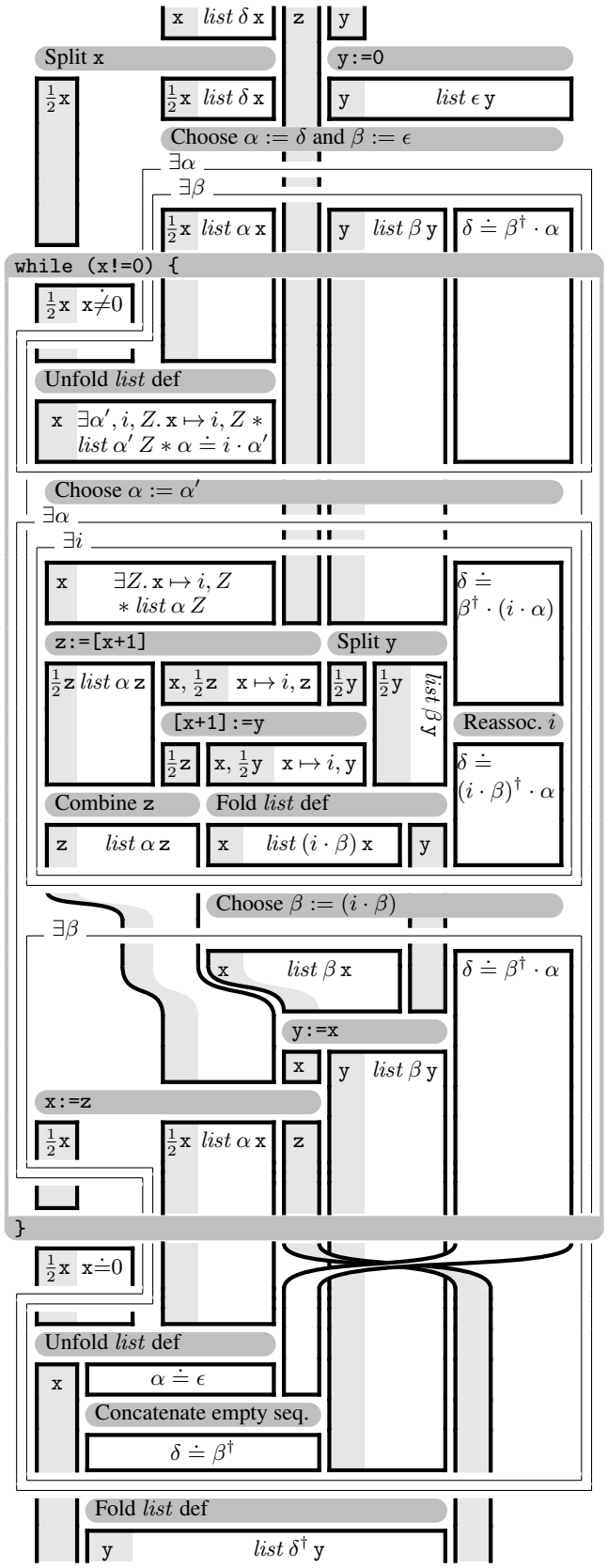


Figure 16. A ribbon proof of list reverse using variables-as-resource

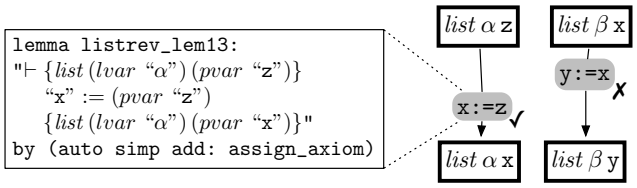


Figure 17. Tool support for checking ribbon proofs

Fig. 5b, which does not use variables-as-resource. Concrete pictures should be drawn carefully so they can be successfully rasterised. Conversely, one who seeks to explore potential optimisations, or to analyse the dependencies between various components of a program, should invest in variables-as-resource.

6. Tool support

We have developed a prototype tool whose inputs are an *ordered* diagram $G = (V_G, \Lambda_G, E_G)$ (see Defn. 17) and a collection of small Isabelle proof scripts: one for each basic step. Our tool translates G into a rasterised diagram, and then uses our Isabelle formalisation of Thm. 22 and the proof rules of Fig. 15 to assemble the Isabelle 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 the ribbon proofs in Figs. 3 and 5b. In both cases, all of the proof scripts for the individual basic steps are small, and they can often be discharged without manual assistance. Ordinarily, when formalising a proof of such a program in separation logic, much effort is expended in using the associativity and commutativity laws of the $*$-operator to manipulate long assertions into particular forms. A key feature of our ribbon proof tool is that this bureaucracy is shifted from the individual proof steps into the surrounding graphical structure, where it is more naturally handled. Note that the RMAIN rule (Fig. 15) permits the individual proof scripts to be checked in any order – even concurrently. This feature recalls recent developments in theorem proving that allow proof scripts to be processed in a non-serial manner [31].

Our tool outputs a pictorial representation of the graph it has verified, laid out using the *dot* tool in the *Graphviz* library.[2] One such picture (with the layout manually tweaked) is shown in Fig. 9. Clicking on any basic step loads the corresponding Isabelle 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. Figure 17 illustrates this on a snippet of Fig. 9, and also shows the Isabelle script for one of the steps.

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 [29]. 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.

[2] `http://www.graphviz.org`

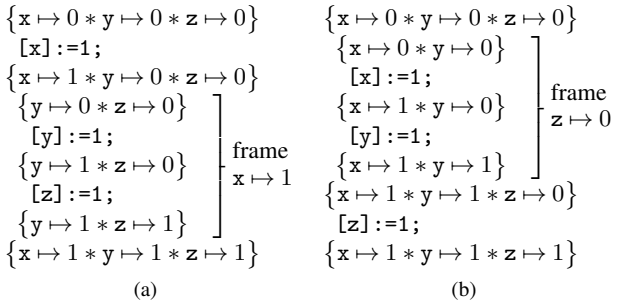


Figure 18. Two alternatives to the proof outline in Fig. 1a

7. 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 [27], 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. 18a depicts, using Bornat's technique [5], 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. 18b). It is unsatisfactory that there are several 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* [12], 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 described in Defn. 5 resemble Milner's *bigraphs* [20], with assertions and commands as nodes, a link graph to show the deductions of the proof, and a place graph to allow existential boxes, choices and loops to contain nested graphs. In fact, our diagrams correspond to a restricted form called *binding* bigraphs, in which edges may not cross place boundaries. Relaxing this restriction may enable a model of the 'dynamic' scoping of existential boxes exhibited in Fig. 6, 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 [28]. In future work we intend to investigate this categorical semantics of ribbon proofs; in particular, the use of *traces* [19] to model the loop construction depicted in Fig. 4b, and coproducts to model if-statements and existential boxes.

Another avenue for future work is to investigate the connection between our ribbon proofs and Raza's *labelled separation logic* [25]. 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 play a similar role to 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 may share a dependency (modulo ribbon twisting, which upsets the column ordering).

We have so far considered only sequential programs, even though the proofs themselves have a concurrent nature. It may

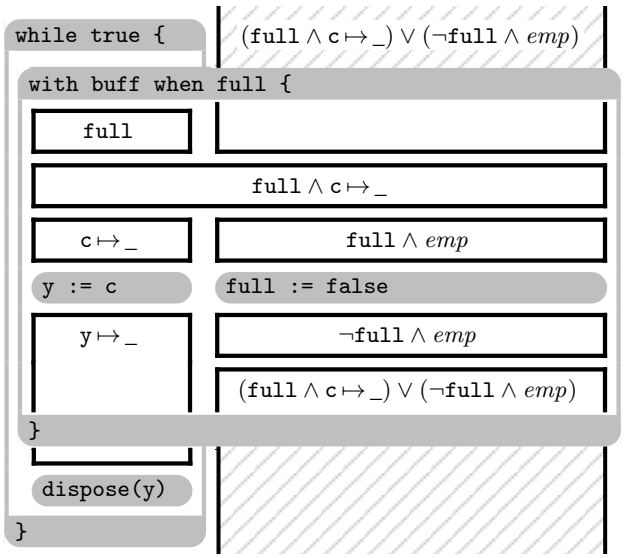


Figure 19. Ribbon proof of single-cell buffer

be possible to extend our ribbon proof system to handle *concurrent separation logic* [21] as follows. Consider a program (adapted from [21]) in which two threads communicate through a shared single-cell buffer at location c:

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

The *resource invariant* protected by the lock buff is (full ∧ c ↦ _) ∨ (¬full ∧ *emp*), which means that the location c is shared exactly when the full flag is set. Figure 19 imagines a ribbon proof of the right-hand thread. The resource invariant is initially placed in a protected ribbon that is inaccessible to the thread (as suggested by the diagonal hashing). Upon entering the critical region, the ribbon becomes available, and upon leaving it, the resource invariant is re-established and the ribbon becomes inaccessible once again.

Beyond concurrent separation logic, we intend our proof system to be applied fruitfully to more advanced separation logics. It has already been applied to a logic for relaxed memory [4]; some other candidates handle fine-grained concurrency [7, 9, 10, 30], dynamic threads [8], storable locks [13], loadable modules [18] and garbage collection [16]. Increasingly complicated logics for increasingly complicated programming features make techniques for intuitive construction and clear presentation ever more crucial.

8. 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 Appx. E). 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. In *MFPS*, 2003.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[4] R. Bornat and M. Dodds. Abducing memory barriers. Draft, 2012.

[5] R. Bornat, C. Calcagno, and P. W. O'Hearn. Local reasoning, separation and aliasing. In *SPACE*, 2004.

[6] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[8] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[9] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[10] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

[11] F. Fitch. *Symbolic logic: an introduction*. Ronald Press Co., 1952.

[12] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[13] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10), 1969.

[15] C. A. R. Hoare. Proof of a program: Find. *Comm. ACM*, 14(1), 1971.

[16] C.-K. Hur, D. Dreyer, and V. Vafeiadis. Separation logic in the presence of garbage collection. In *LICS*, 2011.

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

[18] B. Jacobs, J. Smans, and F. Piessens. Verification of unloadable modules. In *FM*, 2011.

[19] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. of the Cambridge Philosophical Society*, 119(3), 1996.

[20] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[21] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[22] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 1999.

[23] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 1976.

[24] M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, 2006.

[25] M. Raza. *Resource Reasoning and Labelled Separation Logic*. PhD thesis, Imperial College, 2010.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[27] F. Schneider. *On Concurrent Programming*, chapter 4. Springer, 1997.

[28] P. Selinger. A survey of graphical languages for monoidal categories. In *New Structures for Physics*, volume 813, chapter 4. Springer, 2011.

[29] M. Urbas and M. Jamnik. Diabelli: A heterogeneous reasoning framework. In *IJCAR*, 2012.

[30] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

[31] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In *UITP*, 2010.

[32] J. Wickerson, M. Dodds, and M. J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP*, 2010.

[33] C. Wu, X. Zhang, and C. Urban. A formalisation of the Myhill-Nerode theorem based on regular expressions. In *ITP*, 2011.

A. Proof of Theorem 13

Theorem 13. *The following rules are derivable from those in Fig. 11.*

$$\textsc{Seq}\ \frac{\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q \qquad \vdash^{\mathrm{dia}}_{\mathsf{SL}} H : Q \to R}{\vdash^{\mathrm{dia}}_{\mathsf{SL}} \begin{matrix} G \\ H \end{matrix} : P \to R} \qquad \textsc{Par}\ \frac{\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q \qquad \vdash^{\mathrm{dia}}_{\mathsf{SL}} H : P' \to Q'}{\vdash^{\mathrm{dia}}_{\mathsf{SL}} G \parallel H : P\,P' \to Q\,Q'}$$

When sequentially composing diagrams, we shall need to rename nodes.

Definition 24 (Support equivalence)**.** Two diagrams G and H are support-equivalent, written $G \simeq H$, iff there exists a bijection $\rho : V_G \to V_H$ that satisfies $\Lambda_G = \Lambda_H \circ \rho$, and for all $\mathbf{v}, C, \mathbf{w}$:

$$(\mathbf{v}, C, \mathbf{w}) \in E_G \Leftrightarrow (\{\rho\, v \mid v \in \mathbf{v}\}, C, \{\rho\, w \mid w \in \mathbf{w}\}) \in E_H.$$

Lemma 25. *For any diagrams G and H, if $G \simeq H$ then $top(G) = top(H)$ and $bot(G) = bot(H)$.*

Lemma 26. *For any diagrams G and H, if $G \simeq H$ then $\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q = \vdash^{\mathrm{dia}}_{\mathsf{SL}} H : P \to Q$.*

Proof. Suppose $\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q$. Perform rule-inversion on Main, apply the properties given in Defn. 24 and Lem. 25, then re-apply Main. □

We now provide a more careful definition of sequential composition that takes node-renaming into account.

Definition 27 (Sequential composition of diagrams – amended)**.** We notate sequential composition by vertical stacking. We overload this notation for both diagrams and assertion-gadgets. If G and H are diagrams, and there exists $H' \simeq H$ such that:

- $terminals\, G = initials\, H' = V_G \cap V_{H'}$, and
- $\begin{matrix} \Lambda_G(v) \\ \Lambda_{H'}(v) \end{matrix}$ is defined for all $v \in V_G \cap V_{H'}$

then we write $\begin{matrix} G \\ H \end{matrix}$ for the diagram $(V_G \cup V_{H'}, \Lambda, E_G \cup E_{H'})$, where

$$\Lambda(v) = \begin{cases} \Lambda_G(v) & \text{if } v \in V_G \setminus V_{H'} \\ \Lambda_{H'}(v) & \text{if } v \in V_{H'} \setminus V_G \\ \begin{pmatrix} \Lambda_G(v) \\ \Lambda_{H'}(v) \end{pmatrix} & \text{if } v \in V_G \cap V_{H'}. \end{cases}$$

Definition 28 (Parallel composition of diagrams – amended)**.** If G and H are diagrams, and there exists $H' \simeq H$ such that $V_G \not\parallel V_{H'}$, then we write $G \parallel H$ for the diagram

$$(V_G \cup V_{H'}, \Lambda_G \cup \Lambda_{H'}, E_G \cup E_{H'}).$$

Notation. We shall write $E{\downarrow}$ to mean that the expression E is defined.

Lemma 29. *Let:*

$$\begin{aligned}
\Phi_{\mathrm{dia}}(G) &\stackrel{\text{def}}{=} \forall H.\ \textit{if } bot(G) = top(H) \textit{ then} \\
&\quad (\begin{matrix} G \\ H \end{matrix}{\downarrow} \textit{ and } (\textit{if } \vdash^{\mathrm{dia}}_{\mathsf{SL}} G \textit{ and } \vdash^{\mathrm{dia}}_{\mathsf{SL}} H \textit{ then} \\
&\qquad \vdash^{\mathrm{dia}}_{\mathsf{SL}} \begin{matrix} G \\ H \end{matrix} \textit{ and } bot(\begin{matrix} G \\ H \end{matrix}) = bot(H) \\
&\qquad \textit{and } top(\begin{matrix} G \\ H \end{matrix}) = top(G))) \\
\Phi_{\mathrm{asn}}(A) &\stackrel{\text{def}}{=} \forall B.\ \textit{if } bot(A) = top(B) \textit{ then} \\
&\quad (\begin{matrix} A \\ B \end{matrix}{\downarrow} \textit{ and } (\textit{if } \vdash^{\mathrm{asn}}_{\mathsf{SL}} A \textit{ and } \vdash^{\mathrm{asn}}_{\mathsf{SL}} B \textit{ then} \\
&\qquad \vdash^{\mathrm{asn}}_{\mathsf{SL}} \begin{matrix} A \\ B \end{matrix} \textit{ and } bot(\begin{matrix} A \\ B \end{matrix}) = bot(B) \\
&\qquad \textit{and } top(\begin{matrix} A \\ B \end{matrix}) = top(A))) \\
\Phi_{\mathrm{com}}(C) &\stackrel{\text{def}}{=} \textit{true}.
\end{aligned}$$

Then we have

$$\Phi_{\mathrm{dia}}(G) \wedge \Phi_{\mathrm{asn}}(A) \wedge \Phi_{\mathrm{com}}(C)$$

for all diagrams G, assertion-gadgets A and command-gadgets C.

Proof. We proceed by structural induction on diagrams. The six cases are as follows.

1. $\forall p.\ \Phi_{\mathrm{asn}}(\boxed{p})$
2. $\forall x, G.\ \Phi_{\mathrm{dia}}(G) \Rightarrow \Phi_{\mathrm{asn}}(\exists x\, \boxed{G})$
3. $\forall c.\ \Phi_{\mathrm{com}}(c)$
4. $\forall G, H.\ \Phi_{\mathrm{dia}}(G) \wedge \Phi_{\mathrm{dia}}(H) \Rightarrow \Phi_{\mathrm{com}}\left(\begin{matrix} G \\ \texttt{or} \\ H \end{matrix}\right)$
5. $\forall G.\ \Phi_{\mathrm{dia}}(G) \Rightarrow \Phi_{\mathrm{com}}\left(\begin{matrix} \texttt{loop} \\ G \end{matrix}\right)$
6. $\forall G.\ (\forall v \in V_G.\ \Phi_{\mathrm{asn}}(\Lambda_G\, v)) \wedge (\forall (\_, C, \_) \in E_G.\ \Phi_{\mathrm{com}}(C)) \Rightarrow \Phi_{\mathrm{dia}}(G)$

Only the sixth is interesting. To show $\Phi_{\mathrm{dia}}(G)$, we start by picking an arbitrary H and assuming $bot(G) = top(H)$. That is,

$$\otimes_{v \in terminals(G)}\ bot(\Lambda_G\, v) = \otimes_{v \in initials(H)}\ top(\Lambda_H\, v).$$

Hence there exists a bijection $\pi : terminals(G) \to initials(H)$ for which:

$$\forall v \in terminals(G).\ bot(\Lambda_G\, v) = top(\Lambda_H(\pi\, v)).$$

We can apply the first of our two inductive hypotheses to this to obtain:

$$\forall v \in terminals(G).\ \begin{pmatrix} \Lambda_G\, v \\ \Lambda_H(\pi\, v) \end{pmatrix}{\downarrow}. \tag{1}$$

Now we pick a new diagram $H' \simeq H$, obtained by applying a node-renaming ρ to H that satisfies:

$$\begin{aligned} &\forall v \in initials(H).\ \rho(v) = \pi^{-1}(v) \\ &\forall v \in V_H \setminus initials(H).\ \rho(v) \notin V_G \end{aligned}$$

That is, ρ ensures that the initial nodes of H' coincide with the terminal nodes of G, and that its other nodes are disjoint from G's. We now have:

$$terminals(G) = initials(H') = V_G \cap V_{H'}.$$

With (1), we obtain:

$$\forall v \in V_G \cap V_{H'}.\ \begin{pmatrix} \Lambda_G\, v \\ \Lambda_{H'}\, v \end{pmatrix}{\downarrow}.$$

These two facts are sufficient for establishing $\begin{matrix} G \\ H \end{matrix}{\downarrow}$. For the second part of $\Phi_{\mathrm{dia}}(G)$, we must show

$$\vdash^{\mathrm{dia}}_{\mathsf{SL}} \begin{matrix} G \\ H \end{matrix}$$

under the additional assumptions that $\vdash^{\mathrm{dia}}_{\mathsf{SL}} G$ and $\vdash^{\mathrm{dia}}_{\mathsf{SL}} H$ both hold. We use rule inversion on Main, and then Lem. 26 to deduce:

$$\forall v \in V_G.\ \vdash^{\mathrm{asn}}_{\mathsf{SL}} \Lambda_G\, v \tag{2}$$

$$\forall v \in V_{H'}.\ \vdash^{\mathrm{asn}}_{\mathsf{SL}} \Lambda_{H'}\, v \tag{3}$$

$$\forall (\mathbf{v}, C, \mathbf{w}) \in E_G.\ \vdash^{\mathrm{com}}_{\mathsf{SL}} C : \otimes_{v \in \mathbf{v}}\ bot(\Lambda_G\, v) \to \otimes_{w \in \mathbf{w}}\ top(\Lambda_G\, w) \tag{4}$$

$$\forall (\mathbf{v}, C, \mathbf{w}) \in E_{H'}.\ \vdash^{\mathrm{com}}_{\mathsf{SL}} C : \otimes_{v \in \mathbf{v}}\ bot(\Lambda_{H'}\, v) \to \otimes_{w \in \mathbf{w}}\ top(\Lambda_{H'}\, w). \tag{5}$$

We are to show:

$$\forall v \in V_G \cup V_{H'}. \vdash^{\text{asn}}_{\mathsf{SL}} \Lambda\, v \tag{6}$$

$$\forall (\mathbf{v}, C, \mathbf{w}) \in E_G \cup E_{H'}. \quad \vdash^{\text{com}}_{\mathsf{SL}} C : \otimes_{v\in\mathbf{v}}\, bot(\Lambda\, v) \to \otimes_{w\in\mathbf{w}}\, top(\Lambda\, w) \tag{7}$$

where Λ is as defined in Defn. 27. To show (6), fix an arbitrary v in $V_G \cup V_{H'}$. If $v \in V_G \setminus V_{H'}$, use (2). If $v \in V_{H'} \setminus V_G$, use (3). For the case when $v \in V_G \cap V_{H'}$, we require

$$\vdash^{\text{asn}}_{\mathsf{SL}} \begin{pmatrix} \Lambda_G(v) \\ \Lambda_{H'}(v) \end{pmatrix},$$

which is obtained from the inductive hypothesis. To show (7), fix an arbitrary edge $(\mathbf{v}, C, \mathbf{w})$ in $E_G \cup E_{H'}$. Suppose it is in E_G; the other possibility is handled similarly. We can use (4), but only once we have established

$$\otimes_{v\in\mathbf{v}}\, bot(\Lambda_G\, v) = \otimes_{v\in\mathbf{v}}\, bot(\Lambda\, v) \tag{8}$$

$$\otimes_{w\in\mathbf{w}}\, top(\Lambda_G\, w) = \otimes_{w\in\mathbf{w}}\, top(\Lambda\, w). \tag{9}$$

Of these, (8) follows from

$$\forall v \in \mathbf{v}.\, bot(\Lambda_G\, v) = bot(\Lambda\, v),$$

which holds because if $v \in V_G \setminus V_{H'}$ then Λ_G and Λ coincide, and if $v \in V_G \cap V_{H'}$ then v must be a terminal node of G and hence cannot be an incoming node of the edge $(\mathbf{v}, C, \mathbf{w})$. We obtain (9) analogously.

The final part of $\Phi_{\text{dia}}(G)$ requires $top(\binom{G}{H}) = top(H)$ and $bot(\binom{G}{H}) = bot(H)$. We give details only for the latter. After unfolding the definition of bot , it suffices to exhibit a bijection $\pi : terminals(H) \to terminals\,\binom{G}{H}$ such that:

$$\forall v \in terminals\, H.\, bot(\Lambda(\pi\, v)) = bot(\Lambda_H\, v).$$

In fact ρ, restricted to the terminal nodes of H, is such a bijection. It then suffices to show:

$$\forall v \in terminals\, H'.\, bot(\Lambda\, v) = bot(\Lambda_{H'}\, v).$$

This, in turn, is proved by cases. When $v \in V_{H'} \setminus V_G$ then Λ and $\Lambda_{H'}$ coincide by definition. When $v \in V_{H'} \cap V_G$, then $bot(\Lambda\, v)$ is equal to

$$bot \begin{pmatrix} \Lambda_G(v) \\ \Lambda_{H'}(v) \end{pmatrix},$$

which is equal to $bot(\Lambda_{H'}\, v)$ by the induction hypothesis. □

Proof of Thm. 13. The SEQ rule is a straightforward consequence of Lem. 29. For the PAR rule, the key step is to show that

$$top(G \parallel H) = top(G) \otimes top(H)$$

Suppose that the composition operation renames H to H'. We can show

$$initials(G \parallel H) = initials(G) \uplus initials(H')$$

and hence:

$$\begin{array}{lll} & top(G \parallel H) & \\ = & \otimes_{v\in initials(G\parallel H)}(top(\Lambda_{G\parallel H}\, v)) & \\ = & (\otimes_{v\in initials\, G}(top(\Lambda_{G\parallel H}\, v))) & \\ & \otimes\, (\otimes_{v\in initials\, H'}(top(\Lambda_{G\parallel H}\, v))) & \\ = & (\otimes_{v\in initials\, G}(top(\Lambda_G\, v))) \otimes (\otimes_{v\in initials\, H'}(top(\Lambda_{H'}\, v))) & \\ = & top(G) \otimes top(H') & \\ = & top(G) \otimes top(H) & \text{(by Lem. 25).} \end{array}$$

□

B. Proof of Theorem 21

Theorem 21. *The following rules are derivable from those in Fig. 15.*

$$\text{RSEQ}\quad \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to P' \qquad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P' \to P''}{\vdash^{\text{rdia}}_{\mathsf{SL}} \begin{matrix} D \\ E \end{matrix} : P \to P''} \qquad \text{RPAR}\quad \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \qquad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P' \to Q' \qquad D \,\#\, E}{\vdash^{\text{rdia}}_{\mathsf{SL}} D \parallel_\mu E : P\, P' \to Q\, Q'}$$

Proof of RSEQ *rule.* Suppose $D = [D_0, \ldots, D_k]$ and $E = [E_0, \ldots, E_l]$ for non-negative k and l. By rule inversion on RMAIN, we obtain:

$$\forall i \in k+1.\, \vdash^{\text{cell}}_{\mathsf{SL}} D_i : Q_i \to Q_{i+1} \tag{10}$$

$$\forall i \in l+1.\, \vdash^{\text{cell}}_{\mathsf{SL}} E_i : R_i \to R_{i+1} \tag{11}$$

for some $[Q_0, \ldots, Q_{k+1}]$ and $[R_0, \ldots, R_{l+1}]$ with $Q_0 = P$, $Q_{k+1} = P' = R_0$ and $R_{l+1} = P''$. Now define a list

$$[S_0, \ldots, S_{k+l+1}]$$

such that:

$$S_i = \begin{cases} Q_i & \text{if } 0 \le i \le k+1 \\ R_{i-k-1} & \text{if } k+1 \le i \le l+1, \end{cases}$$

noting that $S_{k+1} = Q_{k+1} = R_0$. By the RMAIN rule, it suffices to show:

$$\forall i \in k+l+2.\, \vdash^{\text{cell}}_{\mathsf{SL}} \left(\begin{matrix} D \\ E \end{matrix}\right)_i : S_i \to S_{i+1}.$$

If $i < k+1$, then $\left(\begin{matrix} D \\ E \end{matrix}\right)_i = D_i$, $S_i = Q_i$ and $S_{i+1} = Q_{i+1}$, so the result follows from (10). Otherwise, if $k+1 \le i < k+l+2$, then $\left(\begin{matrix} D \\ E \end{matrix}\right)_i = E_{i-k-1}$, $S_i = R_{i-k-1}$ and $S_{i+1} = R_{i-k}$, so the result follows from (11). □

For proving the RPAR rule, we shall require a little more machinery. We employ the following generalisation of the Hoare triple.

Definition 30 (Hoare chain)**.** A *Hoare chain* Π is a term of the following language:

$$\Pi ::= \{P\} \mid \{P\}\, (\gamma, F)\, \Pi$$

where $P, F \in \mathsf{Interface}$ and $\gamma \in \mathsf{Cell}$. A Hoare chain of length k can be written

$$\{P_0\}\, (\gamma_0, F_0)\, \{P_1\} \cdots \{P_{k-1}\}\, (\gamma_{k-1}, F_{k-1})\, \{P_k\}.$$

If this chain is called Π, then we define $pre(\Pi)$ as P_0 and $post(\Pi)$ as P_k.

Definition 31 (Provability of a Hoare chain)**.** A chain is provable, written $\vdash^{\text{chain}}_{\mathsf{SL}} \Pi$, if each of its triples is provable; that is:

$$\begin{array}{lll} \vdash^{\text{chain}}_{\mathsf{SL}} \{P\} & = & true \\ \vdash^{\text{chain}}_{\mathsf{SL}} \{P\}\, (\gamma, F)\, \Pi & = & \vdash^{\text{rdia}}_{\mathsf{SL}} (\gamma, F) : P \to pre(\Pi) \\ & & \text{and } \vdash^{\text{chain}}_{\mathsf{SL}} \Pi. \end{array}$$

Definition 32 (Extracting a Hoare chain from a rasterised diagram)**.** Note that the empty list is not a rasterised diagram.

$$\begin{array}{lll} chain[(\gamma, F)] & = & \{top\, \gamma \otimes F\}\, (\gamma, F)\, \{bot\, \gamma \otimes F\} \\ chain((\gamma, F) :: D) & = & \{top\, \gamma \otimes F\}\, (\gamma, F)\, (chain\, D). \end{array}$$

Lemma 33. *We have:*

$$pre(chain\, D) = top\, D \qquad post(chain\, D) = bot\, D.$$

Lemma 34. $\vdash^{\text{chain}}_{\mathsf{SL}} (chain\, D)$ *if and only if* $\vdash^{\text{rdia}}_{\mathsf{SL}} D$.

Proof. By structural induction on D. □

Definition 35 (Parallel composition of Hoare chains)**.** If Π_0 and Π_1 are Hoare chains of lengths k and l, and μ is a sequence containing k zeroes and l ones, then $\Pi_0 \parallel_\mu \Pi_1$ is defined according to the following equations:

$$\{P\} \parallel_\epsilon \{Q\} \quad = \quad \{P \otimes Q\}$$

$$(\{P\}\,(\gamma, F)\,\Pi_0) \parallel_{0\mu} \Pi_1 \quad = \quad \{P \otimes \mathit{pre}(\Pi_1)\}\,(\gamma, F \otimes \mathit{pre}(\Pi_1))\,(\Pi_0 \parallel_\mu \Pi_1)$$

$$\Pi_0 \parallel_{1\mu} (\{Q\}\,(\gamma, F)\,\Pi_1) \quad = \quad \{\mathit{pre}(\Pi_0) \otimes Q\}\,(\gamma, \mathit{pre}(\Pi_0) \otimes F)\,(\Pi_0 \parallel_\mu \Pi_1).$$

Lemma 36. *For any $k \geq 0$, for any binary sequence μ containing $k_0 + 1$ zeroes and $k_1 + 1$ ones, where $k = k_0 + k_1$, and for any provable rasterised diagrams D (of length k_0+1) and E (of length $k_1 + 1$):*

$$\begin{array}{rcl} \mathit{chain}(D \parallel_\mu E) & = & \mathit{chain}(D) \parallel_\mu \mathit{chain}(E) \\ \mathit{top}(D \parallel_\mu E) & = & \mathit{top}(D) \otimes \mathit{top}(E) \\ \mathit{bot}(D \parallel_\mu E) & = & \mathit{bot}(D) \otimes \mathit{bot}(E) \end{array}$$

Proof. By mathematical induction on k. In the base case, μ is either 01 or 10. In the inductive step, μ is either $0\mu'$ or $1\mu'$, for some μ' containing at least one zero and one one. □

Lemma 37. *If $\vdash^{\mathrm{cell}}_{\mathsf{SL}}(\gamma, F) : P \to Q$ and $wr(\gamma) \not\asymp rd(R)$ then $\vdash^{\mathrm{cell}}_{\mathsf{SL}}(\gamma, F \otimes R) : P \otimes R \to Q \otimes R$.*

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

Lemma 38. *For any $k \geq 0$, for any binary sequence μ containing k_0 zeroes and k_1 ones, where $k = k_0 + k_1$, and for any chains Π_0 and Π_1 of lengths k_0 and k_1, if $\vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_0$ and $\vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_1$ and $\Pi_0 \# \Pi_1$ then $\vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_0 \parallel_\mu \Pi_1$ and $\mathit{pre}(\Pi_0 \parallel_\mu \Pi_1) = \mathit{pre}(\Pi_0) \otimes \mathit{pre}(\Pi_1)$.*

Proof. By mathematical induction on k. When $k = 0$, then $k_0 = k_1 = 0$, so Π_0 and Π_1 both comprise single interfaces, say $\{P\}$ and $\{Q\}$. Hence $\Pi_0 \parallel_\mu \Pi_1 = \{P \otimes Q\}$, which is vacuously provable. For the inductive step, assume $k = 1 + k'$ for some $k' \geq 0$. Then μ is non-empty, and hence begins with 0 or 1. Suppose it begins with 0; the alternative case is argued similarly. That is, $\mu = 0\mu'$ for some μ'. We deduce $k_0 > 0$, which means Π_0 can be written as

$$\{P\}\,(\gamma, F)\,\Pi_0'.$$

for some P, γ, F and Π_0'. Since Π_0 is provable, then so is Π_0', and

$$\vdash^{\mathrm{cell}}_{\mathsf{SL}}(\gamma, F) : P \to \mathit{pre}(\Pi_0') \tag{12}$$

holds. Now, $\Pi_0 \parallel_\mu \Pi_1$ is equal to:

$$\{P \otimes \mathit{pre}(\Pi_1)\}\,(\gamma, F \otimes \mathit{pre}(\Pi_1))\,(\Pi_0' \parallel_{\mu'} \Pi_1)$$

by Defn. 35. This Hoare chain is provable if

$$\vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_0' \parallel_{\mu'} \Pi_1 \tag{13}$$

$$\vdash^{\mathrm{cell}}_{\mathsf{SL}}(\gamma, F \otimes \mathit{pre}(\Pi_1)) : P \otimes \mathit{pre}(\Pi_1) \to \mathit{pre}(\Pi_0' \parallel_{\mu'} \Pi_1). \tag{14}$$

But (13) holds as a direct result of the induction hypothesis. The induction hypothesis also allows (14) to be written as:

$$\vdash^{\mathrm{cell}}_{\mathsf{SL}}(\gamma, F \otimes \mathit{pre}(\Pi_1)) : P \otimes \mathit{pre}(\Pi_1) \to \mathit{pre}(\Pi_0') \otimes \mathit{pre}(\Pi_1)$$

which follows from (12) via Lem. 37, noting that the side-condition on variable interference is met having assumed $\Pi_0 \# \Pi_1$. □

Proof of RPAR *rule.* The soundness of the following rule:

$$\frac{\vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_0 \qquad \vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_1 \qquad \Pi_0 \# \Pi_1}{\vdash^{\mathrm{chain}}_{\mathsf{SL}} \Pi_0 \parallel_\mu \Pi_1}$$

follows from Lem. 38. The RPAR rule can be derived from this rule, together with Lems. 34 and 36. □

C. Proof of Theorem 23

This proof has been formalised in Isabelle, and the proof script can be viewed online at:

`http://www.cl.cam.ac.uk/~jpw48/ribbons.html`

Theorem 23 (Soundness – variables-as-resource)**.** *Using variables-as-resource, any ribbon diagram can be recreated as a separation logic proof:*

$$\vdash^{\mathrm{dia}}_{\mathsf{VaR}} G : P \to Q \Longrightarrow \forall c \in \mathit{coms}\, G. \vdash_{\mathsf{VaR}} \{\mathit{asn}\, P\}\, c\, \{\mathit{asn}\, Q\}.$$

Notation. For sets X and Y, let $X \uplus Y$ be defined when $X \not\asymp Y$ as $X \cup Y$, and $X - Y$ be defined when $Y \subseteq X$ as $X \setminus Y$.

To prove this theorem, we employ the following generalisation of a Hoare triple.

Definition 39 (Hoare chain)**.** A Hoare chain is a sequence

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

where each P_i is an Interface, and each x_i is either a ComGadget or an AsnGadget.

Definition 40 (Extracting Hoare chains)**.** First, we define the notion of a *proof state*. At any point while stepping through a Hoare chain, a proof state $\sigma \subseteq V_G \times \{\mathrm{TOP}, \mathrm{BOT}\}$ records those node-identifiers which are either initial or have been produced as the postcondition of an already-processed hyperedge and not yet consumed as a precondition of another. A node-identifier is tagged BOT if it has been processed, and TOP if it hasn't. Then, for a diagram G, we define $\mathit{chains}(G)$ as the set of all Hoare chains

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

for which there exist a list $[\sigma_0, \ldots, \sigma_k]$ of proof-states and a bijection $\pi : k \to V_G \uplus E_G$ with the following properties. First, for all $(\mathbf{v}, C, \mathbf{w}) \in E_G$,

$$\begin{array}{ll} \pi^{-1}(\mathbf{v}, C, \mathbf{w}) < \pi^{-1}(w) & \text{for all } w \in \mathbf{w} \\ \pi^{-1}(v) < \pi^{-1}(\mathbf{v}, C, \mathbf{w}) & \text{for all } v \in \mathbf{v}. \end{array}$$

Second,

$$\sigma_0 = (\mathit{initials}\, G) \times \{\mathrm{TOP}\}$$

and, for all $i \in k$,

$$\sigma_{i+1} = \begin{cases} (\sigma_i - \{(v, \mathrm{TOP})\}) \uplus \{(v, \mathrm{BOT})\} & \text{if } \pi(i) = v \\ (\sigma_i - (\mathbf{v} \times \{\mathrm{BOT}\})) \uplus (\mathbf{w} \times \{\mathrm{TOP}\}) & \\ \qquad\qquad \text{if } \pi(i) = (\mathbf{v}, C, \mathbf{w}) & \end{cases}$$

Third, for all $i \in k + 1$,

$$P_i = \left(\otimes_{(v,\mathrm{TOP}) \in \sigma_i} \mathit{top}(\Lambda_G\, v)\right) \otimes \left(\otimes_{(v,\mathrm{BOT}) \in \sigma_i} \mathit{bot}(\Lambda_G\, v)\right).$$

Finally, for all $i \in k$,

$$x_i = \begin{cases} \Lambda_G\, v & \text{if } \pi(i) = v \\ C & \text{if } \pi(i) = (\mathbf{v}, C, \mathbf{w}). \end{cases}$$

Because the '$-$' and '$\uplus$' operators are only partial, we require the following lemma to confirm that the list $[\sigma_0, \ldots, \sigma_k]$ in the above definition is well-defined.

Lemma 41 (Well-definedness of chains)**.** *For any diagram G, every Hoare chain in $\mathit{chains}(G)$ is well-defined, begins with $\mathit{top}(G)$ and ends with $\mathit{bot}(G)$.*

Our strategy for proving this lemma is mathematical induction on the size of G. First we must modify Defn. 40, as follows.

Definition 42 (Extracting Hoare chains – amended)**.** For a diagram G and a set $S \subseteq \mathit{initials}(G)$, we define $\mathit{chains}(G, S)$ as the set of all Hoare chains

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

for which there exist a list $[\sigma_0, \ldots, \sigma_k]$ of proof-states and a bijection $\pi : k \to (V_G \setminus S) \uplus E_G$ with the following properties. (The role of S is to contain those of G's initial nodes which have already been processed, and hence should not be included in the resultant Hoare chains.) First, for all $(\mathbf{v}, C, \mathbf{w}) \in E_G$:

$$\begin{array}{ll} \pi^{-1}(\mathbf{v}, C, \mathbf{w}) < \pi^{-1}(w) & \text{for all } w \in \mathbf{w} \\ \pi^{-1}(v) < \pi^{-1}(\mathbf{v}, C, \mathbf{w}) & \text{for all } v \in \mathbf{v} \setminus S. \end{array}$$

Second,

$$\sigma_0 = \{(v, \text{TOP}) \mid v \in (\mathit{initials}\, G) \setminus S\} \cup \{(v, \text{BOT}) \mid v \in S\}.$$

and, for all $i \in k$,

$$\sigma_{i+1} = \begin{cases} (\sigma_i - \{(v, \text{TOP})\}) \uplus \{(v, \text{BOT})\} & \text{if } \pi(i) = v \\ (\sigma_i - (\mathbf{v} \times \{\text{BOT}\})) \uplus (\mathbf{w} \times \{\text{TOP}\}) & \\ & \text{if } \pi(i) = (\mathbf{v}, C, \mathbf{w}) \end{cases}$$

Third, for all $i \in k+1$,

$$P_i = \left(\otimes_{(v,\text{TOP}) \in \sigma_i} \mathit{top}(\Lambda_G\, v)\right) \otimes \left(\otimes_{(v,\text{BOT}) \in \sigma_i} \mathit{bot}(\Lambda_G\, v)\right).$$

Finally, for all $i \in k$,

$$x_i = \begin{cases} \Lambda_G\, v & \text{if } \pi(i) = v \\ C & \text{if } \pi(i) = (\mathbf{v}, C, \mathbf{w}). \end{cases}$$

Lemma 43. *For all k:*

$$\begin{array}{l} \forall G.\, \forall S \subseteq \mathit{initials}(G).\, \forall H \in \mathit{chains}(G, S). \\ \textit{if } |V_G \setminus S| + |E_G| = k \textit{ then} \\ H \textit{ is well-defined and ends with } \mathit{bot}(G). \end{array}$$

Proof. We use mathematical induction on k.

Case 0**.** Each chain is of the form $\{P_0\}$, so is trivially well-defined. There being no edges, every node is both initial and terminal. We have $V_G = \mathit{initials}(G) = \mathit{terminals}(G) = S$. So

$$\sigma_0 = (\mathit{terminals}\, G) \times \{\text{BOT}\},$$

and hence

$$P_0 = \otimes_{v \in \mathit{terminals}\, G} \mathit{bot}(\Lambda_G\, v) = \mathit{bot}(G)$$

as required.

Case $k+1$**.** Each chain is of the form

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

We case-split on whether x_0 is an AsnGadget or a ComGadget.

Case $x_0 \in$ AsnGadget**.** Hence $\pi(0) = v$. Since $[x_0, \ldots, x_{k-1}]$ is a valid linear extension of G, we have $v \in \mathit{initials}(G)$. Since S is excluded from π's co-domain, we have $v \notin S$. Hence σ_0 contains (v, TOP) but not (v, BOT). This ensures that σ_1 is well-defined, and hence, so is the initial step $\{P_0\}\, x_0\, \{P_1\}$ of the chain. It now suffices to show that the rest of the chain is in $\mathit{chains}(G, S \uplus \{v\})$, for then, by the induction hypothesis, the remainder – and hence the entire chain – is well-defined and ends with $\mathit{bot}(G)$. To see this, define a new bijection π' such that $\pi(i) = \pi(i+1)$ for all $i \in k$, and a new list $[\sigma'_0, \ldots, \sigma'_{k-1}] = [\sigma_1, \ldots \sigma_{k+1}]$ and confirm that the four properties listed in Defn. 42 hold.

Case $x_0 \in$ ComGadget**.** Hence $\pi(0) = (\mathbf{v}, C, \mathbf{w})$. Since $[x_0, \ldots, x_{k-1}]$ is a valid linear extension of G, x_0 has no dependants; that is, $\mathbf{v} \subseteq S$. Hence $\mathbf{v} \times \{\text{BOT}\} \subseteq \sigma_0$. Moreover, $\mathbf{w} \times \{\text{BOT}\} \not\subseteq \sigma_0$ because $\mathbf{w} \not\subseteq (\mathit{initials}\, G)$, which follows from Defn. 8. This ensures that σ_1 is well-defined, and hence, so is the initial step of the chain. Consider the graph G' obtained by removing from G the hyperedge $(\mathbf{v}, C, \mathbf{w})$ and the vertices in $\mathbf{v}$ (which, by LINEARITY, are not endpoints of any remaining hyperedge). The removal preserves ACYCLICITY and LINEARITY, so G' is well-formed; moreover, $\mathit{bot}(G') = \mathit{bot}(G)$. Let S' be $S \setminus \mathbf{v}$, and note that $|V_{G'} \setminus S'| + |E_{G'}| = k$. The rest of the chain is in $\mathit{chains}(G', S')$, and hence, by the induction hypothesis, is well-defined and ends in $\mathit{bot}(G')$. Thus, the entire chain is well-defined and ends in $\mathit{bot}(G)$. Isabelle

Proof of Lemma 41. It is a straightforward consequence of the definition of σ_0 and Defn. 9 that every Hoare chain in $\mathit{chains}(G)$ begins with $\mathit{top}(G)$. We note that when S is empty, $\mathit{chains}(G, S)$ coincides with $\mathit{chains}(G)$, so Lem. 43 implies the result. Isabelle

Proof of Thm. 23. We prove the following three statements by mutual rule induction.

$$\begin{array}{l} \vdash^{\text{dia}}_{\mathsf{VaR}} G : P \to Q \Longrightarrow \forall c \in \mathit{coms}\, G.\, \vdash_{\mathsf{VaR}} \{\mathit{asn}\, P\}\, c\, \{\mathit{asn}\, Q\} \\ \vdash^{\text{com}}_{\mathsf{VaR}} C : P \to Q \Longrightarrow \forall c \in \mathit{coms}\, C.\, \vdash_{\mathsf{VaR}} \{\mathit{asn}\, P\}\, c\, \{\mathit{asn}\, Q\} \\ \vdash^{\text{asn}}_{\mathsf{VaR}} A \Longrightarrow \forall c \in \mathit{coms}\, A.\, \vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}\, A)\}\, c\, \{\mathit{asn}(\mathit{bot}\, A)\} \end{array}$$

We focus on the MAIN rule, as the others are straightforward consequences of the corresponding rules in Fig. 7. Our inductive hypotheses are:

$$\begin{array}{l} \forall v \in V_G.\, \forall c \in \mathit{coms}(\Lambda_G\, v). \\ \quad \vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}(\Lambda_G\, v))\}\, c\, \{\mathit{asn}(\mathit{bot}(\Lambda_G\, v))\} \end{array} \tag{15}$$

$$\begin{array}{l} \forall (\mathbf{v}, C, \mathbf{w}) \in E_G.\, \forall c \in \mathit{coms}\, C. \\ \quad \vdash_{\mathsf{VaR}} \{\mathit{asn}(\otimes_{v \in \mathbf{v}} \mathit{bot}(\Lambda_G\, v))\}\, c\, \{\mathit{asn}(\otimes_{w \in \mathbf{w}} \mathit{top}(\Lambda_G\, w))\} \end{array} \tag{16}$$

We are to prove, for all $c \in \mathit{coms}\, G$, that:

$$\vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}\, G)\}\, c\, \{\mathit{asn}(\mathit{bot}\, G)\}. \tag{17}$$

Observe that c can be written c_0 ; $\cdots$; c_{k-1} ; skip for some linear extension $[x_0, \ldots, x_{k-1}]$ of G, where $c_i \in \mathit{coms}(x_i)$ for all $i \in k$. Consider the corresponding Hoare chain:

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

We pick an arbitrary $i \in k$, and proceed depending on whether x_i is an AsnGadget or a ComGadget.

Case $x_i \in$ AsnGadget**.** Hence $\pi(i) = v$. Hence

$$\sigma_{i+1} = (\sigma_i - \{(v, \text{TOP})\}) \uplus \{(v, \text{BOT})\}.$$

By Lem. 41, this expression is well-defined, and hence

$$\begin{array}{lcl} \sigma_i & = & \{(v, \text{TOP})\} \uplus \sigma' \\ \sigma_{i+1} & = & \{(v, \text{BOT})\} \uplus \sigma' \end{array}$$

for some σ'. Hence

$$\begin{array}{lcl} P_i & = & \mathit{top}(\Lambda_G\, v) \otimes P' \\ P_{i+1} & = & \mathit{bot}(\Lambda_G\, v) \otimes P' \end{array}$$

for some P'. By (15), we have

$$\vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}(\Lambda_G\, v))\}\, c_i\, \{\mathit{asn}(\mathit{bot}(\Lambda_G\, v))\}$$

from which

$$\vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}(\Lambda_G\, v)) * \mathit{asn}\, P'\}\, c_i\, \{\mathit{asn}(\mathit{bot}(\Lambda_G\, v)) * \mathit{asn}\, P'\}$$

follows by separation logic's frame rule (which, under variables-as-resource, has no side-conditions). Hence

$$\vdash_{\mathsf{VaR}} \{\mathit{asn}\, P_i\}\, c_i\, \{\mathit{asn}\, P_{i+1}\}.$$

Case $x_i \in$ ComGadget**.** Similar, using (16) instead of (15) .

We then use Hoare logic's sequencing rule to assemble a proof of the entire chain:

$$\vdash_{\mathsf{VaR}} \{\mathit{asn}\, P_0\}\, c\, \{\mathit{asn}\, P_k\}.$$

It remains to show that P_0 is $\mathit{top}(G)$ and P_k is $\mathit{bot}(G)$; this follows directly from Lem. 41. Isabelle

D. Rasterisation

The following process converts an ordered diagram G into a rasterised diagram $\mathcal{R}_{\mathrm{dia}}\, G$. We assume that all sets of vertices are ordered. For both lists and sets, we write $\cdot$ for concatenation.

$$
\begin{aligned}
\mathcal{R}_{\mathrm{dia}}\, G &= \mathcal{R}_{\mathrm{edges}}\, \mathbf{v}\, E_G \\
&\quad \text{where } \mathbf{v} = \mathit{initials}\, G \\
\mathcal{R}_{\mathrm{edges}}\, \mathbf{v}\, [\,] &= \mathcal{R}_{\mathrm{ids}}\, \varepsilon\, (\emptyset, \mathbf{v}) \\
\mathcal{R}_{\mathrm{edges}}\, \mathbf{v}\, ((\mathbf{w_1}, C, \mathbf{w_2}) :: E) &= \\
(\mathcal{R}_{\mathrm{ids}}\, F\, (\emptyset, \mathbf{w_1})) \cdot [\mathcal{R}_{\mathrm{com}}(P, C, Q)] \cdot \mathcal{R}_{\mathrm{edges}}\, \mathbf{v}'\, E & \\
\text{where } F &= \otimes_{v \in \mathbf{v} \setminus \mathbf{w_1}}\, \mathit{top}(\Lambda_G\, v) \\
P &= \otimes_{w \in \mathbf{w_1}}\, \mathit{bot}(\Lambda_G\, w) \\
Q &= \otimes_{w \in \mathbf{w_2}}\, \mathit{top}(\Lambda_G\, w) \\
\mathbf{v}' &= (\mathbf{v} \setminus \mathbf{w_1}) \cup \mathbf{w_2}
\end{aligned}
$$

$$
\mathcal{R}_{\mathrm{com}}(P, \boxed{c}, Q) = \begin{array}{c} P \\ \boxed{c} \\ Q \end{array}
$$

$$
\mathcal{R}_{\mathrm{com}}(P, \begin{array}{|c|}\hline G_1 \\ \texttt{or} \\ G_2 \\ \hline \end{array}, Q) = \begin{array}{c} P \\ \begin{array}{|c|}\hline \mathcal{R}_{\mathrm{dia}}\, G_1 \\ \texttt{or} \\ \mathcal{R}_{\mathrm{dia}}\, G_2 \\ \hline \end{array} \\ Q \end{array}
$$

$$
\mathcal{R}_{\mathrm{com}}(P, \begin{array}{|c|}\hline \texttt{loop} \\ G \\ \hline \end{array}, Q) = \begin{array}{c} P \\ \begin{array}{|c|}\hline \texttt{loop} \\ \mathcal{R}_{\mathrm{dia}}\, G \\ \hline \end{array} \\ Q \end{array}
$$

$$
\begin{aligned}
\mathcal{R}_{\mathrm{ids}}\, F\, (\mathbf{v}_{\mathrm{left}}, \emptyset) &= [\,] \\
\mathcal{R}_{\mathrm{ids}}\, F\, (\mathbf{v}_{\mathrm{left}}, v \cdot \mathbf{v}_{\mathrm{right}}) &= \\
[(\mathcal{R}_{\mathrm{asn}}(\Lambda_G\, v), F \otimes F_{\mathrm{left}} \otimes F_{\mathrm{right}})] \cdot \mathcal{R}_{\mathrm{ids}}\, F\, (\mathbf{v}_{\mathrm{left}} \cdot v, \mathbf{v}_{\mathrm{right}}) & \\
\text{where } F_{\mathrm{left}} &= \otimes_{v \in \mathbf{v}_{\mathrm{left}}}\, \mathit{bot}(\Lambda_G\, v) \\
F_{\mathrm{right}} &= \otimes_{v \in \mathbf{v}_{\mathrm{right}}}\, \mathit{top}(\Lambda_G\, v)
\end{aligned}
$$

$$
\mathcal{R}_{\mathrm{asn}}\ \boxed{p} = \boxed{p}
$$

$$
\mathcal{R}_{\mathrm{asn}}\ \boxed{\exists x G} = \boxed{\exists x \mathcal{R}_{\mathrm{dia}}\, G}
$$

E. Ribbon proof of Version 7 Unix memory manager

We illustrate the ability of our system to produce readable proofs for more complex programs. Our case study is the memory manager from Version 7 Unix, the abridged and corrected source code of which is presented in Fig. 20.

The ribbon proof for this program, presented at the end of this section, is several times larger than the proof outline for the same program that has been published previously.[3] This is because it provides far more detail; to an extent that could not be supported by the proof outline without becoming tediously repetitive.

Preliminaries

One feature of this case study is that components of assertions may be undefined. When they are, the entire assertion is deemed false. Picking an example from the glossary in Fig. 21: the expression $C \circ\!\!- D$ is undefined when C does not begin with D. By exploiting undefinedness, our assertions become more expressive. For instance, the assertion $C = D \circ\!\!- [d]$ imparts not only that C is D's tail, but also that d is D's head.

Another convention we adopt is that both program variables and logical variables are typed, though the types are not written explicitly in the proof. Some types are defined in the glossary. We

[3] J. Wickerson, M. Dodds, and M. J. Parkinson, "Explicit stabilisation for modular rely-guarantee reasoning," University of Cambridge, Tech. Rep., 2010

```
#define WORD sizeof(st)
#define BLOCK 1024
#define testbusy(p) ((int)(p)&1)
#define setbusy(p) (st *)((int)(p)|1)
#define clearbusy(p) (st *)((int)(p)&~1)

struct store { struct store *ptr; };
typedef struct store st;
static st s[2]; /*initial arena*/
static st *v; /*search ptr*/
static st *t; /*arena top*/

char* sbrk();

char* malloc(unsigned int nbytes) {
  register st *p, *q;
  register nw; static temp;
  if(s[0].ptr == 0) { /*first time*/
    s[0].ptr = setbusy(&s[1]);
    s[1].ptr = setbusy(&s[0]);
    t = &s[1]; v = &s[0];
  }
  nw = (nbytes+WORD+WORD-1)/WORD;
  for(p = v; ; ) {
    for(temp = 0; ; ) {
      if(!testbusy(p->ptr)) {
        q = p->ptr;
        while(!testbusy(q->ptr)) {
          p->ptr = q->ptr; q = p->ptr;
        }
        if(q >= p+nw && p+nw >= p) goto found;
      }
      q = p; p = clearbusy(p->ptr);
      if(p > q) ;
      else if(q !=t || p != s) return 0;
      else if(++temp > 1) break;
    }
    temp = ((nw+BLOCK/WORD)
             /(BLOCK/WORD))*(BLOCK/WORD);
    q = (st *)sbrk(0);
    if(q+temp < q) return 0;
    q = (st *)sbrk(temp*WORD);
    if((int)q == -1) {
      v = s; //line added to fix bug
      return 0;
    }
    t->ptr = q;
    if(q != t+1) t->ptr = setbusy(t->ptr);
    t = q->ptr = q+temp-1;
    t->ptr = setbusy(s);
  }
  found:
  v = p+nw;
  if(q > v) v->ptr = p->ptr;
  p->ptr = setbusy(v);
  return((char *)(p+1));
}

free(register char *ap) {
  register st *p = (st *)ap;
  v = --p;
  p->ptr = clearbusy(p->ptr);
}
```

Figure 20. The Version 7 Unix memory manager. From `http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/libc/gen/malloc.c`. Abridged and corrected.

model the C types `int` and `unsigned int` using the sets $\mathbb{Z}$ and $\mathbb{N}$ respectively. (Thus, our proof does not consider integer overflows.) We model pointers as fractions in the following set:

$$\mathtt{ptr} \stackrel{\text{def}}{=} \{x \mid x \times \mathtt{WORD} \in \mathbb{Z}\},$$

where `WORD`, the number of bytes in a word, is typically either 4 or 8. Consequently, if x is a pointer, then '$x + 1$' denotes the next *word* rather than the next *byte*. We can access the lower bits of a pointer (as is frequently required in this case study) by adding or subtracting fractions. Note that having dividing all pointers by `WORD`, we must be very careful when comparing pointers with non-pointers.

The semantics of the single-cell assertion (with respect to a store s and a heap h) requires the address to be word-aligned, that is, to evaluate to a positive natural number:

$$s, h \models \boxed{\begin{array}{|c|}\hline e_1 \\ \hline e_2 \\ \hline \end{array}} \stackrel{\text{def}}{=} \begin{array}{l}\text{let } l_1 = [\![e_1]\!](s) \text{ and } l_2 = [\![e_2]\!](s) \\ \text{in } l_1 \in \mathbb{N}^+ \wedge h = \{l_1 \mapsto l_2\}.\end{array}$$

As seen in the definition above, this case study introduces a new notation for a memory cell: the usual separation logic notation '$e_1 \mapsto e_2$' is replaced by $\boxed{\begin{array}{|c|}\hline e_1 \\ \hline e_2 \\ \hline \end{array}}$. Although it sacrifices linearity, this notation is better able to extend to ranges of cells, in a manner recalling Reynolds' *partition diagrams*.[4] Ranges appear frequently in our case study, so a good notation is desirable. We use only what Reynolds calls 'regular' ranges, whose upper bound is greater than or equal to its lower bound:

$$s, h \models \boxed{\begin{array}{|c|c|}\hline e_1 & \\ \hline \end{array}}^{e_2} \stackrel{\text{def}}{=} \begin{array}{l}\text{let } l_1 = [\![e_1]\!](s) \text{ and } l_2 = [\![e_2]\!](s) \\ \text{in } l_1, l_2 \in \mathbb{N}^+ \wedge l_1 \leq l_2 \wedge \\ dom(h) = \{i \in \mathbb{N}^+ \mid l_1 \leq i < l_2\}.\end{array}$$

The default diagram has an inclusive lower bound and an exclusive upper bound, but alternatives can be obtained by switching between $\boxed{\begin{array}{|c|}\hline e \\ \hline \end{array}}$ and $\boxed{\begin{array}{|c|}\hline e+1 \\ \hline \end{array}}$. Two ranges may be concatenated; that is, $\boxed{\begin{array}{|c|c|}\hline e_1 & \\ \hline \end{array}}^{e_2} * \boxed{\begin{array}{|c|c|}\hline e_2 & \\ \hline \end{array}}^{e_3}$ implies $\boxed{\begin{array}{|c|c|}\hline e_1 & \\ \hline \end{array}}^{e_3}$. We can write $\boxed{\begin{array}{|c|c|}\hline e & e \\ \hline \end{array}}$ as $\boxed{\begin{array}{|c|}\hline e \\ \hline \end{array}}$. As an example: an unallocated chunk in the arena is written $\boxed{\begin{array}{|c|c|}\hline x & \\ \hline y & \\ \hline \end{array}}^{y}$. This can be rewritten in terms of the primitives defined above as $\boxed{\begin{array}{|c|}\hline x \\ \hline y \\ \hline \end{array}} * \boxed{\begin{array}{|c|c|}\hline x+1 & \\ \hline \end{array}}^{y}$. It depicts a single cell at x with contents y, followed by zero or more cells up to, but not including, the cell at y.

The memory manager

The specifications for the `malloc` and `free` routines are given at the beginning of Fig. 21. The precondition for `malloc` encompasses the possibility that the arena has not yet been initialised ($uninit$). Once initialised, an arena comprises a monotonic sequence of chunks, each preceded by a pointer to the next chunk's pointer. Since chunks are word-aligned, the lower bits of their pointers are redundant. The least significant of these is thus employed as a 'busy' bit, set when the following chunk is allocated. The final chunk is pointed to by `t`; it is permanently marked 'busy' and points back to the first chunk, which is pointed to by `s`.

The allocation strategy employed by `malloc` is circular first-fit. The search begins at the last-freed chunk (called the 'victim' chunk and pointed to by `v`), and coalesces consecutive unallocated chunks as it goes. If the request cannot be satisfied, further memory is requested from the system via a call to `sbrk`. The resulting chunk is appended to the end of the arena, and any gap is simply marked as an allocated chunk. If the call to `sbrk` fails, `malloc` fails too; this possibility is captured by the second disjunct in its postcondition.

The $arena$ predicate is parameterised by a mapping A of type `chunks_ext`, which associates the first cell of each allocated chunk with that chunk's size in words. The 'internal' representation of the arena, used in the proof but not displayed in the specification, is a list C of type `chunks_int`. Each element $\langle x, \tau, y\rangle$ of C describes a chunk whose pointer is located at x and points to y, and has busy status τ. From a client's perspective, such a chunk comprises $y - x - 1$ usable cells, the first of which is at location $x + 1$. In the definition of the $arena$ predicate, the list C is split, at the victim chunk, into C_1 and C_2. We write $A \subseteq (C_1 \circ C_2)^{\mathsf{a}}$ to express that every chunk recorded in A is indeed marked as allocated in the arena. The converse inclusion does not hold because, as noted above, not all chunks marked as allocated have actually been given to a client.

The $uninit$ and $arena$ predicates that appear in the specifications are treated as *abstract* by the clients of the memory manager.[5] That is, the definitions presented at the end of Fig. 21 remain private to the manager for the sake of modularity. Likewise, the $brka$ predicate (which means that the breakpoint is positioned at or after the given address) is only defined within the `sbrk` routine, although the axiom $x \leq y \wedge brka(y) \Rightarrow brka(x)$ is exposed.

We remark that an alternative specification has been proposed, which avoids the exposure of the logical variable A and the program variables `s`, `v` and `t`.[5] It is possible to modify our proof to satisfy this improved specification [32], but we refrain from doing so here to avoid introducing several technical complications of little relevance to ribbon proofs.

The proof

The ribbon proof is best read by concentrating on each command c in turn, and checking that it correctly transforms those ribbons directly above it into those directly below it. Ribbons to the left or right of c can largely be ignored; the only requirement upon them is not to mention any program variable that c writes.

Finally, we note three further conventions adopted in this proof. First, whenever a ribbon in a command's postcondition also appears in its precondition, and the size and positioning makes the correspondence unambiguous, the label in the postcondition can be replaced by a 'ditto' mark. Second, to aid the reader, we repeat ribbon labels at the top of each page, slightly dimmed. We sometimes issue similar reminders at the beginning of else-branches of if-statements. Third, we use 'jigsaw puzzle pieces' to help the reader connect the ribbons correctly at the boundaries of if- and while-blocks (in accordance with Fig. 4).

[4] J. C. Reynolds, "Reasoning about arrays," *Communications of the ACM*, vol. 22, no. 5, pp. 290–299, May 1979

[5] M. J. Parkinson and G. M. Bierman, "Separation logic and abstraction," in *POPL*, 2005

Specifications of main routines

$$\left\{\begin{array}{l} \mathit{uninit}\ \mathtt{s}\ A \vee \\ \mathit{arena}\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A \end{array}\right\} \mathtt{malloc(n)} \left\{\begin{array}{l} (\mathit{arena}\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \lceil \mathtt{n}/\mathtt{WORD} \rceil\}) * \boxed{\begin{array}{|l|}\hline \mathtt{ret} \\ \hline \\ \hline \end{array}}^{\mathtt{ret} + \lceil \mathtt{n}/\mathtt{WORD} \rceil}) \vee \\ (\mathit{arena}\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A * \mathtt{ret} \doteq 0) \end{array}\right\}$$

$$\left\{\mathit{arena}\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{x} \mapsto n\}) * \boxed{\begin{array}{|l|}\hline \mathtt{x} \\ \hline \\ \hline \end{array}}^{\mathtt{x}+n}\right\} \mathtt{free(x)} \left\{\mathit{arena}\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A\right\}$$

Specifications of sub-routines

$$\left\{\mathit{brka}\ x\right\} \mathtt{sbrk(n)} \left\{\begin{array}{l} (\mathit{brka}\ x * \mathtt{ret} \doteq -1/\mathtt{WORD} \wedge \mathtt{n} \neq 0) \vee \\ (\mathit{brka}(\mathtt{ret} + \lceil \mathtt{n}/\mathtt{WORD} \rceil) * \boxed{\begin{array}{|l|}\hline \mathtt{ret} \\ \hline \\ \hline \end{array}}^{\mathtt{ret} + \lceil \mathtt{n}/\mathtt{WORD} \rceil} * x \dot{\leq} \mathtt{ret}) \end{array}\right\}$$

Types

$$\begin{array}{rcl} \mathsf{tag} & \stackrel{\text{def}}{=} & \{\mathsf{u}, \mathsf{a}\} \qquad \text{(unallocated/allocated)} \\ \mathsf{ptr} & \stackrel{\text{def}}{=} & \{x \in \mathbb{R} \mid x \times \mathtt{WORD} \in \mathbb{Z}\} \\ \mathsf{chunks\_int} & \stackrel{\text{def}}{=} & \{C : (\mathbb{N}^+ \times \mathsf{tag} \times \mathbb{N}^+)\ \mathsf{list} \mid \\ & & \quad \text{each } \langle x, \tau, y\rangle \text{ in } C \text{ satisfies } x < y, \text{ and each} \\ & & \quad \text{consecutive pair } \langle x, \tau, y\rangle, \langle x', \tau', y'\rangle \text{ satisfies } y \leq x'\} \\ \mathsf{chunks\_ext} & \stackrel{\text{def}}{=} & \mathbb{N}^+ \rightharpoonup \mathbb{N} \end{array}$$

Operators

$$\begin{array}{rcl} (-)^{\mathsf{a}} : \mathsf{chunks\_int} \to \mathsf{chunks\_ext} & \stackrel{\text{def}}{=} & \lambda C.\, \{(x+1 \mapsto y-x-1) \mid \langle x, \mathsf{a}, y\rangle \in C\} \\ (A : \mathsf{chunks\_ext}) \uplus (A' : \mathsf{chunks\_ext}) & \stackrel{\text{def}}{=} & \begin{cases} A \cup A' & \text{if } dom(A) \cap dom(A') = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases} \\ (C : \mathsf{chunks\_int}) \circ (C' : \mathsf{chunks\_int}) & \stackrel{\text{def}}{=} & \begin{cases} \text{concatenation of } C \text{ and } C' & \text{if result is a valid } \mathsf{chunks\_int} \\ \text{undefined} & \text{otherwise} \end{cases} \\ (C : \mathsf{chunks\_int}) \circ\!\!- (C' : \mathsf{chunks\_int}) & \stackrel{\text{def}}{=} & \begin{cases} C'' & \text{if } C = C' \circ C'' \\ \text{undefined} & \text{otherwise} \end{cases} \\ y_{\blacksquare} & \stackrel{\text{def}}{=} & y + \frac{1}{\mathtt{WORD}} \end{array}$$

Predicates

$$\begin{array}{rcl} \mathit{chunk}_{\mathsf{u}}(x : \mathsf{ptr})\,(y : \mathsf{ptr}) & \stackrel{\text{def}}{=} & \boxed{\begin{array}{|l|l|}\hline x & \\ \hline y & \\ \hline \end{array}}^{y} \\ \mathit{chunk}_{\mathsf{a}}(x : \mathsf{ptr})\,(y : \mathsf{ptr}) & \stackrel{\text{def}}{=} & x \dot{<} y * \boxed{\begin{array}{|l|}\hline x \\ \hline y_{\blacksquare} \\ \hline \end{array}} \\ \mathit{chunks}(x : \mathsf{ptr})\,(y : \mathsf{ptr})\,(C : \mathsf{chunks\_int}) & \stackrel{\text{def}}{=} & (x \doteq y * C \doteq [\,]) \vee \exists z : \mathsf{ptr}.\, \exists \tau : \mathsf{tag}. \\ & & \quad \mathit{chunk}_\tau\ x\ z * \mathit{chunks}\ z\ y\,(C \circ\!\!- [\langle x, \tau, z\rangle]) \\ \mathit{uninit}\,(s : \mathsf{ptr})\,(A : \mathsf{chunks\_ext}) & \stackrel{\text{def}}{=} & \boxed{\begin{array}{|l|l|}\hline s & \\ \hline 0 & 0 \\ \hline \end{array}} * A \doteq \emptyset * \mathit{brka}(s+2) \\ \mathit{arena}\,(s : \mathsf{ptr})\,(v : \mathsf{ptr})\,(t : \mathsf{ptr})\,(A : \mathsf{chunks\_ext}) & \stackrel{\text{def}}{=} & \exists C_1, C_2 : \mathsf{chunks\_int}.\, \mathit{chunks}\ s\ v\ C_1 * \mathit{chunks}\ v\ t\ C_2 \\ & & \quad * A \dot{\subseteq} (C_1 \circ C_2)^{\mathsf{a}} * \boxed{\begin{array}{|l|}\hline t \\ \hline s_{\blacksquare} \\ \hline \end{array}} * \mathit{brka}(t+1) \end{array}$$

Lemma A. $\mathit{chunks}\ x\ y\ C_1 * \mathit{chunks}\ y\ z\ C_2 \implies \mathit{chunks}\ x\ z\ (C_1 \circ C_2)$

Lemma B. $\mathit{chunk}_\tau\ x\ y \implies \mathit{chunks}\ x\ y\ [\langle x, \tau, y\rangle]$

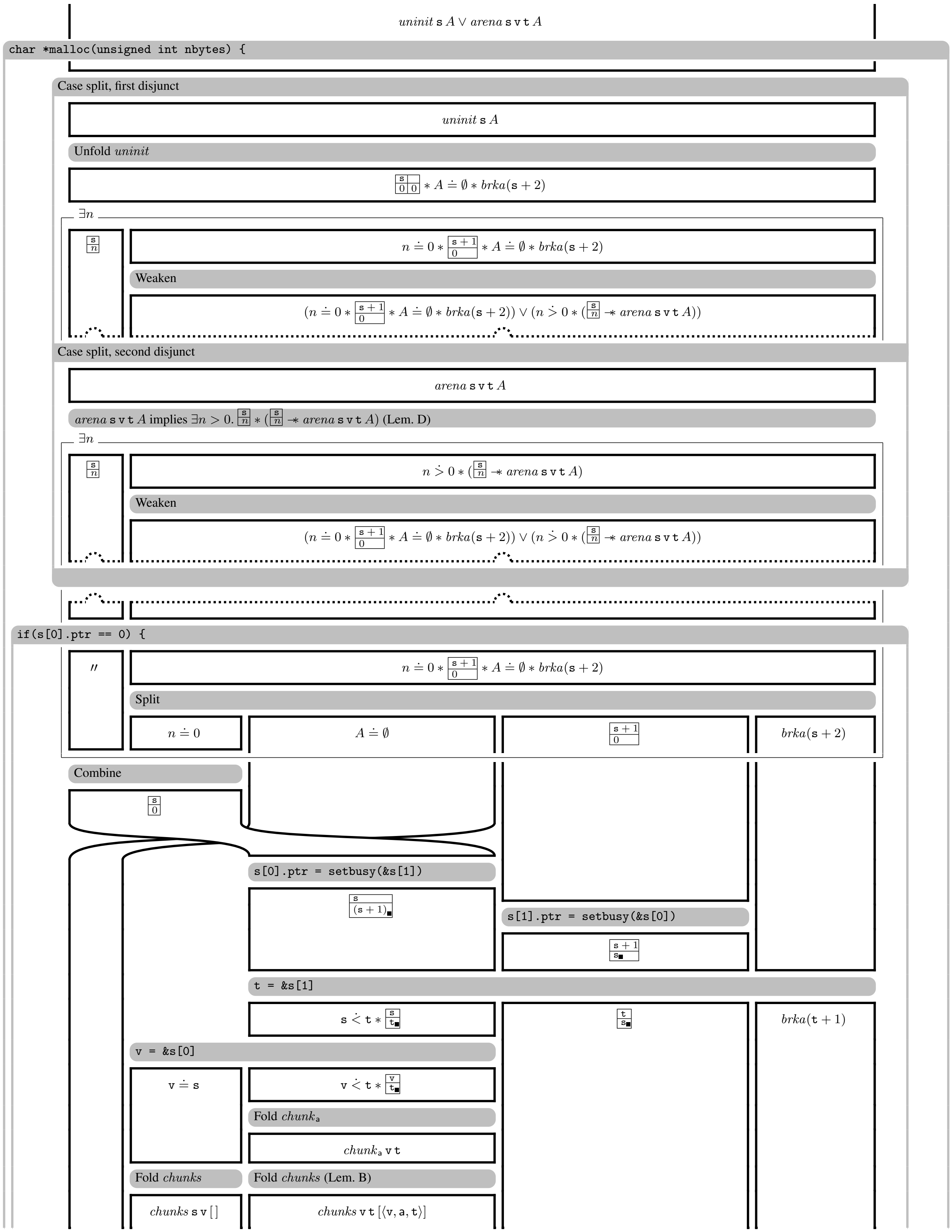
Lemma C. $\mathit{chunks}\ w\ x\ C_1 * x \dot{\leq} y * \mathit{chunks}\ y\ z\ C_2 \implies (C_1 \circ C_2)$ *is defined*

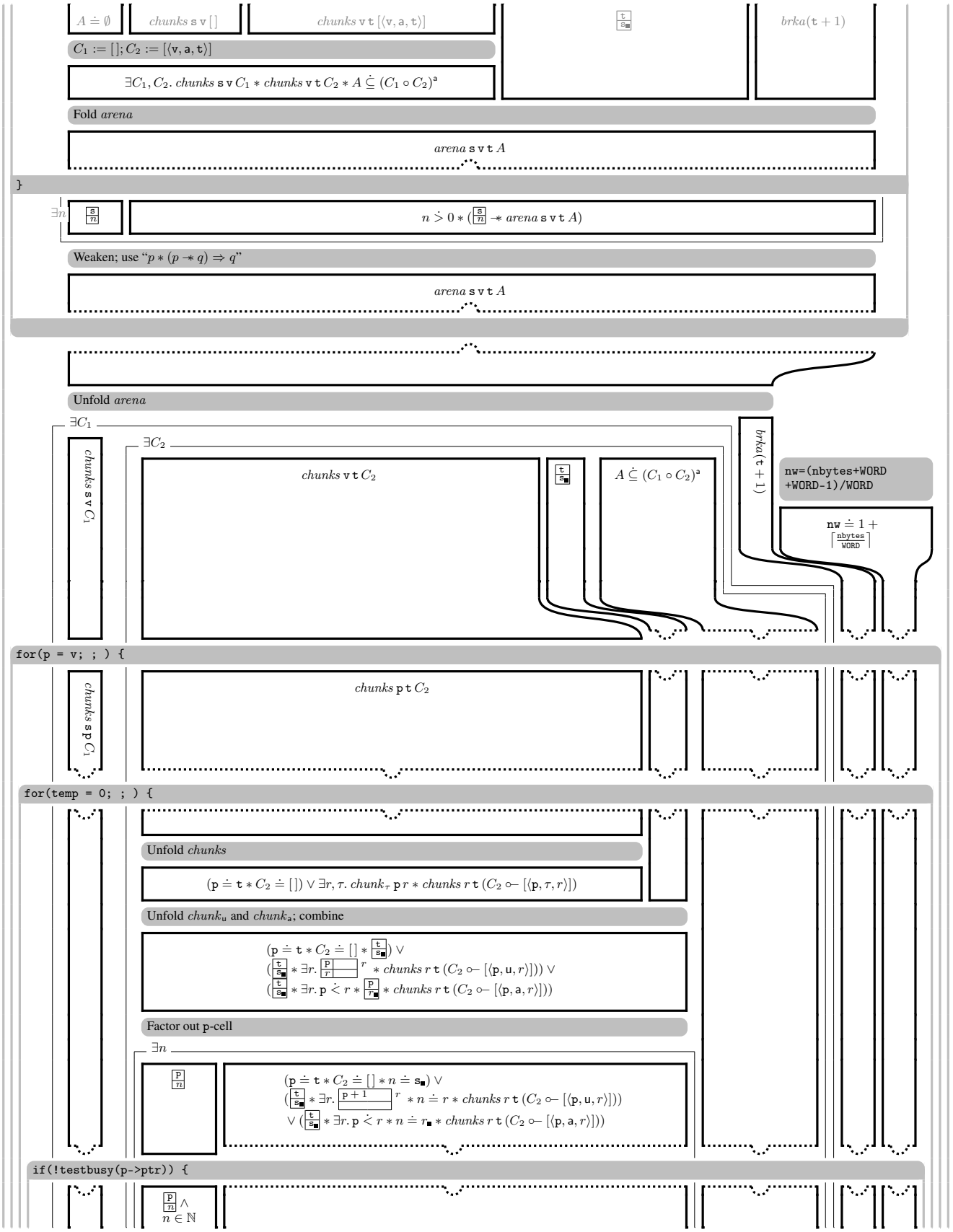
Lemma D. $\mathit{arena}\ s\ v\ t\ A \implies \exists n > 0.\, \boxed{\begin{array}{|l|}\hline s \\ \hline n \\ \hline \end{array}} * (\boxed{\begin{array}{|l|}\hline s \\ \hline n \\ \hline \end{array}} \mathrel{-\!\!*} \mathit{arena}\ s\ v\ t\ A)$

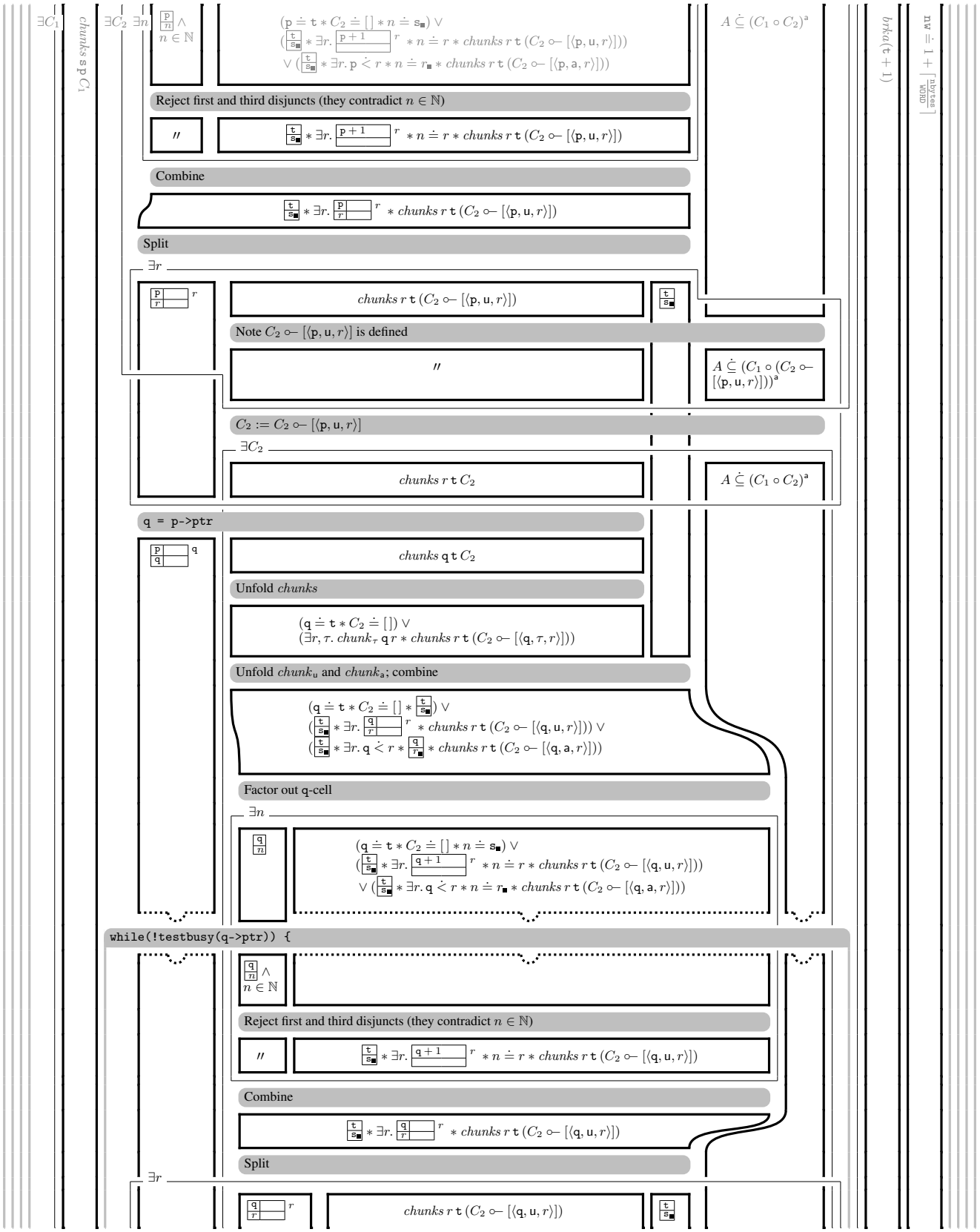
Lemma E. $\mathit{chunks}\ w\ x\ C \wedge \langle y, \tau, z\rangle \in C \implies \exists C_1, C_2.\, \mathit{chunks}\ w\ y\ C_1 * \mathit{chunk}_\tau\ y\ z * \mathit{chunks}\ z\ x\ C_2$

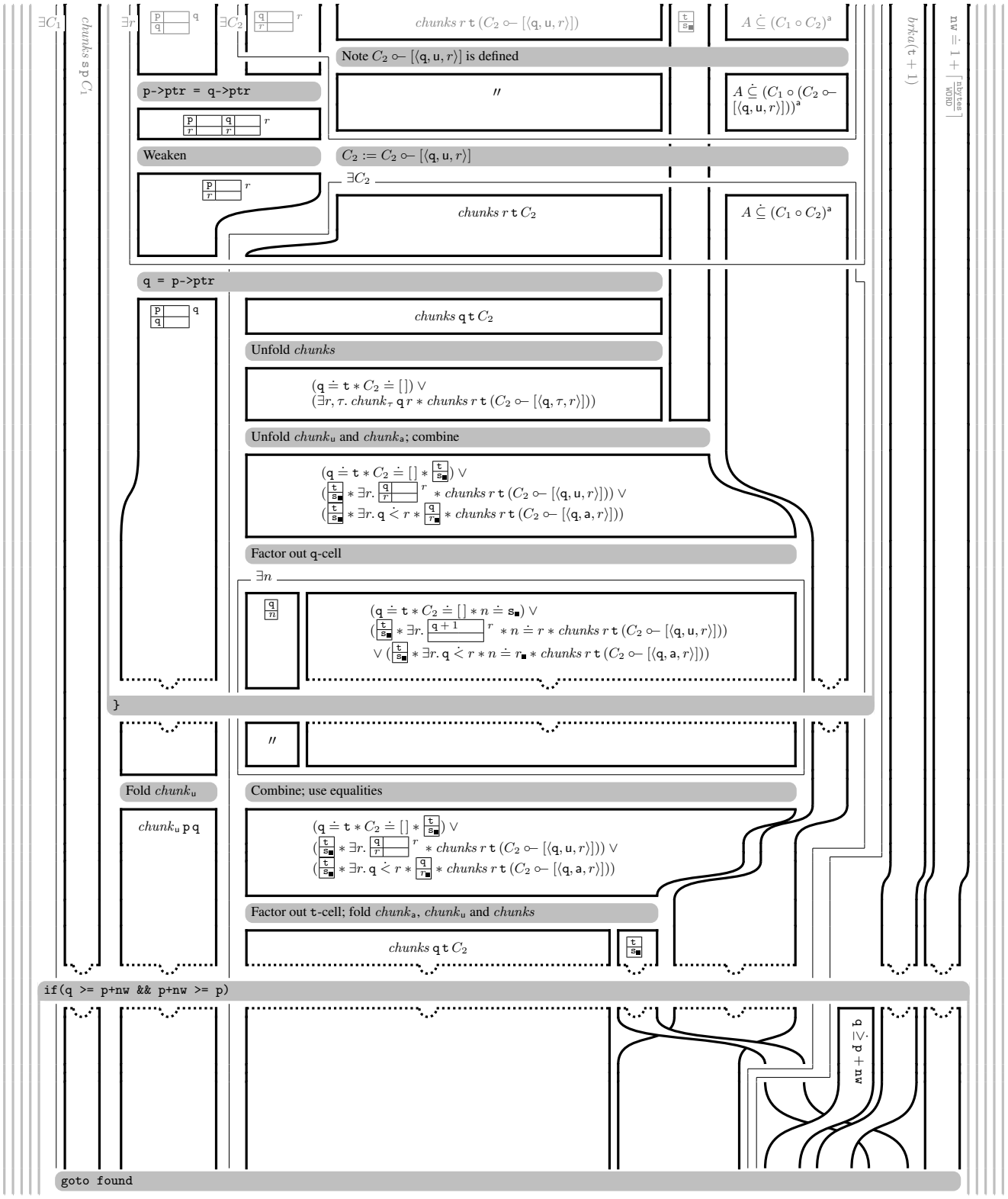
Lemma F. $x \leq y \wedge \mathit{brka}(y) \implies \mathit{brka}(x)$

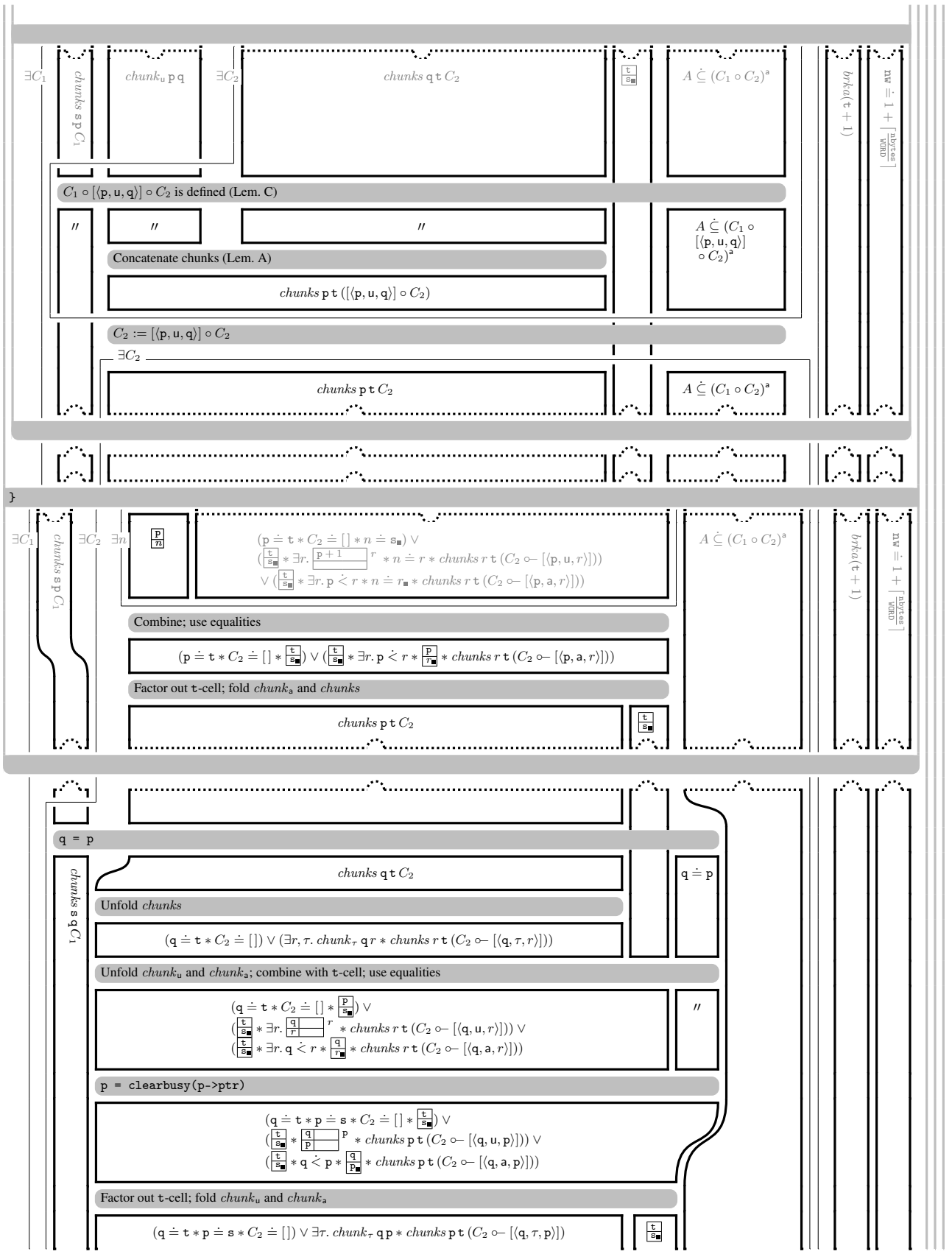
Figure 21. Glossary of definitions and lemmas used in the specifications and proofs of `malloc` and `free`

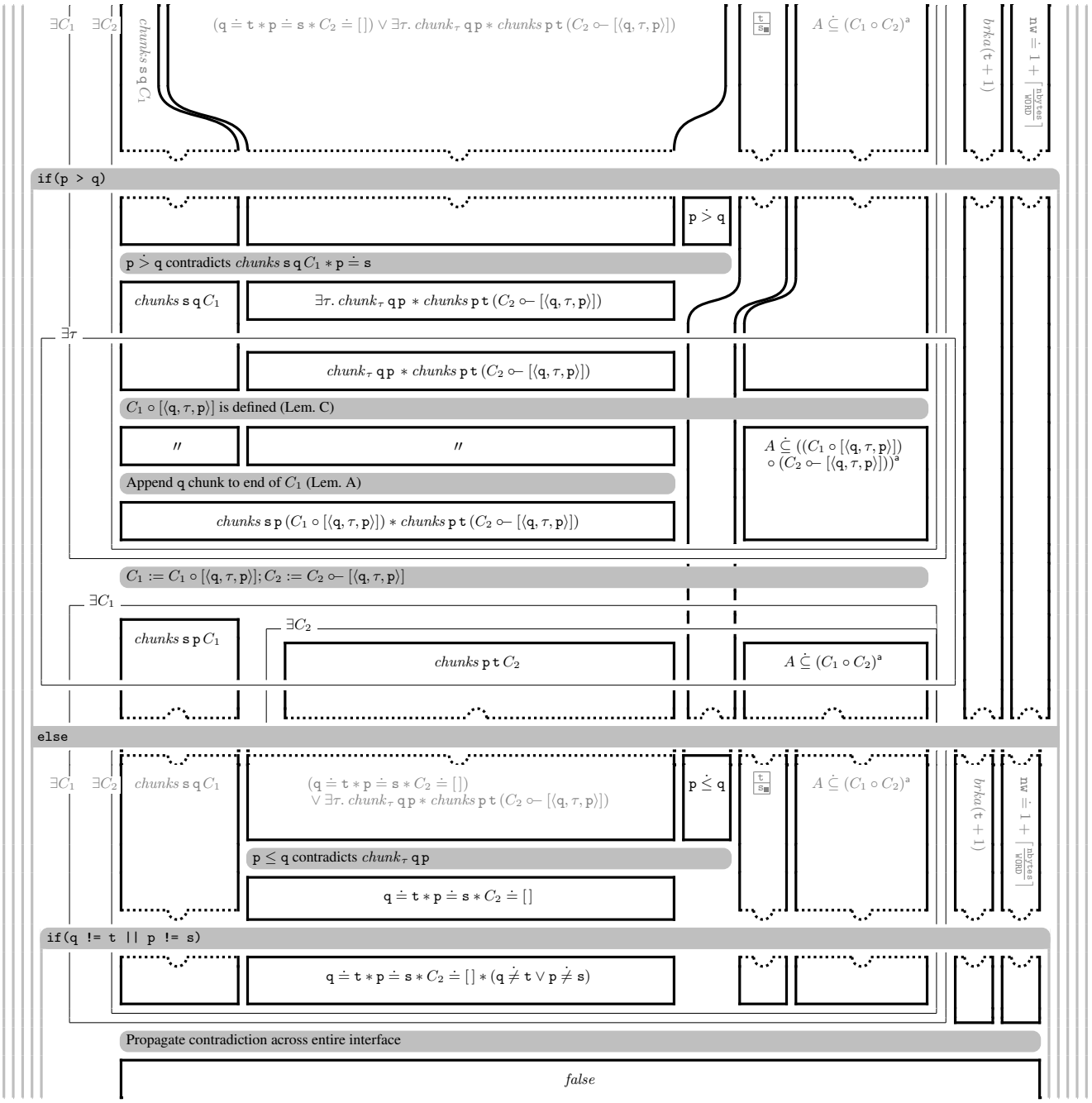

uninit s A ∨ arena s v t A
char *malloc(unsigned int nbytes) {
Case split, first disjunct
uninit s A
Unfold uninit
s 0 0 ∗ A ≐ ∅ ∗ brka(s + 2)
∃n
s n
n ≐ 0 ∗ s+1 0 ∗ A ≐ ∅ ∗ brka(s + 2)
Weaken
(n ≐ 0 ∗ s+1 0 ∗ A ≐ ∅ ∗ brka(s + 2)) ∨ (n > 0 ∗ (s n −∗ arena s v t A))
Case split, second disjunct
arena s v t A
arena s v t A implies ∃n > 0. s n ∗ (s n −∗ arena s v t A) (Lem. D)
∃n
s n
n > 0 ∗ (s n −∗ arena s v t A)
Weaken
(n ≐ 0 ∗ s+1 0 ∗ A ≐ ∅ ∗ brka(s + 2)) ∨ (n > 0 ∗ (s n −∗ arena s v t A))
if(s[0].ptr == 0) {
n ≐ 0 ∗ s+1 0 ∗ A ≐ ∅ ∗ brka(s + 2)
Split
n ≐ 0
A ≐ ∅
s+1 0
brka(s + 2)
Combine
s 0
s[0].ptr = setbusy(&s[1])
s (s+1)■
s[1].ptr = setbusy(&s[0])
s+1 s■
t = &s[1]
s < t ∗ s t■
t s■
brka(t + 1)
v = &s[0]
v ≐ s
v < t ∗ v t■
Fold chunk_a
chunk_a v t
Fold chunks
Fold chunks (Lem. B)
chunks s v [ ]
chunks v t [⟨v, a, t⟩]

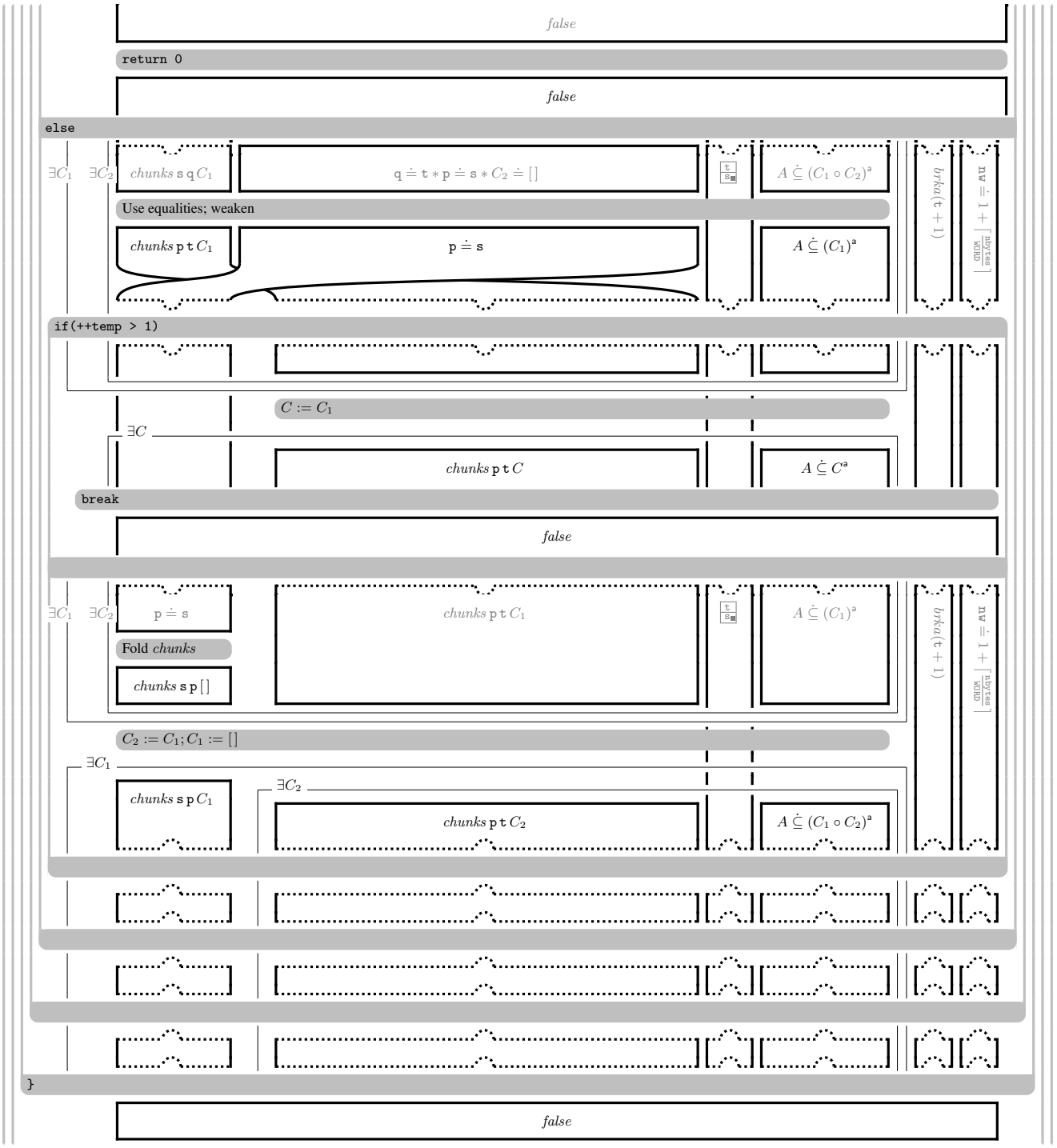

$A \doteq \emptyset$
$chunks\ \mathtt{s}\ \mathtt{v}\ [\,]$
$chunks\ \mathtt{v}\ \mathtt{t}\ [\langle \mathtt{v}, \mathtt{a}, \mathtt{t} \rangle]$
$\boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare}$
$brka(\mathtt{t}+1)$
$C_1 := [\,]; C_2 := [\langle \mathtt{v}, \mathtt{a}, \mathtt{t} \rangle]$
$\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
Fold *arena*
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A$
}
$\exists n$
$\boxed{\mathtt{s}}_{n\mathtt{l}}$
$n \dot{>} 0 * (\boxed{\mathtt{s}}_{n\mathtt{l}} \mathrel{-\!\!*} arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A)$
Weaken; use "$p * (p \mathrel{-\!\!*} q) \Rightarrow q$"
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A$
Unfold *arena*
$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{v}\ C_1$
$\exists C_2$
$chunks\ \mathtt{v}\ \mathtt{t}\ C_2$
$\boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare}$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
$brka(\mathtt{t}+1)$
nw=(nbytes+WORD
+WORD-1)/WORD
$\mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \right\rceil$
for(p = v; ;) {
$chunks\ \mathtt{s}\ \mathtt{d}\ \mathtt{p}\ C_1$
$chunks\ \mathtt{p}\ \mathtt{t}\ C_2$
for(temp = 0; ;) {
Unfold *chunks*
$(\mathtt{p} \doteq \mathtt{t} * C_2 \doteq [\,]) \vee \exists r, \tau.\ chunk_\tau\ \mathtt{p}\ r * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \tau, r \rangle])$
Unfold $chunk_{\mathtt{u}}$ and $chunk_{\mathtt{a}}$; combine
$(\mathtt{p} \doteq \mathtt{t} * C_2 \doteq [\,] * \boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare}) \vee$
$(\boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{p} \mid \quad}_{r}^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{u}, r \rangle])) \vee$
$(\boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare} * \exists r.\ \mathtt{p} \dot{<} r * \boxed{\mathtt{p}}_{r_\blacksquare} * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{a}, r \rangle]))$
Factor out p-cell
$\exists n$
$\boxed{\mathtt{p}}_{n}$
$(\mathtt{p} \doteq \mathtt{t} * C_2 \doteq [\,] * n \doteq \mathtt{s}_\blacksquare) \vee$
$(\boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{p}+1 \mid \quad}_{r}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{u}, r \rangle]))$
$\vee (\boxed{\mathtt{t}}_{\mathtt{s}_\blacksquare} * \exists r.\ \mathtt{p} \dot{<} r * n \doteq r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{a}, r \rangle]))$
if(!testbusy(p->ptr)) {
$\boxed{\mathtt{p}}_{n\mathtt{l}} \wedge$
$n \in \mathbb{N}$

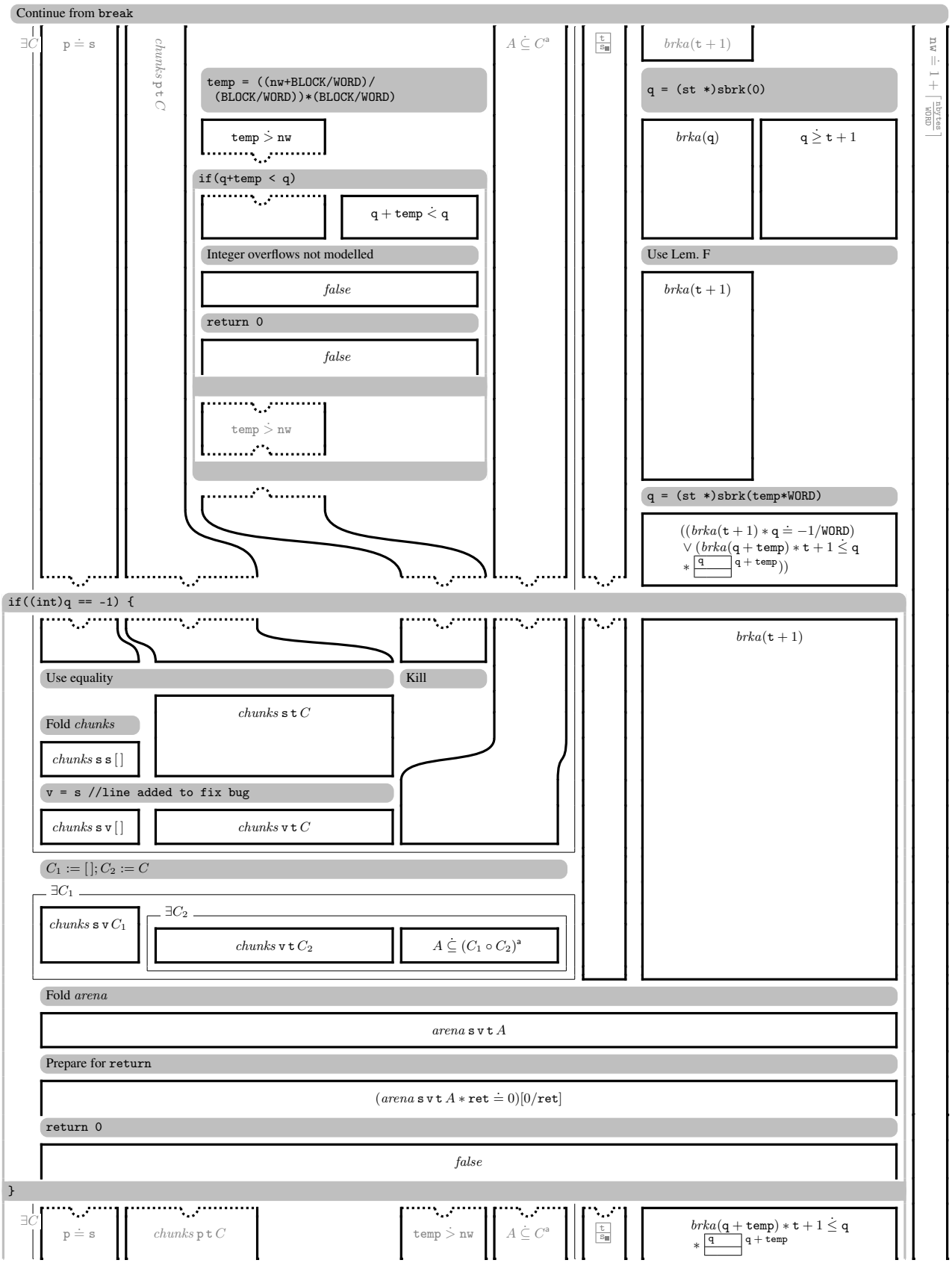

$\exists C_1$
$chunks\ \mathtt{s}\ C_1$
$\exists C_2\ \exists n$
$\boxed{\mathtt{p}\mid n} \wedge n \in \mathbb{N}$
$(\mathtt{p} \doteq \mathtt{t} * C_2 \doteq [\,] * n \doteq \mathtt{s}_\blacksquare) \vee$
$(\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{p}+1}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle])) \vee$
$(\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \mathtt{p} \dot{<} r * n \doteq r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{p},\mathtt{a},r\rangle]))$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
Reject first and third disjuncts (they contradict $n \in \mathbb{N}$)
〃
$\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{p}+1}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle])$
Combine
$\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{p}\mid r}^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle])$
Split
$\exists r$
$\boxed{\mathtt{p}\mid r}^{\,r}$
$chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle])$
$\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare}$
Note $C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle]$ is defined
〃
$A \dot{\subseteq} (C_1 \circ (C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle]))^{\mathtt{a}}$
$C_2 := C_2 \multimap [\langle \mathtt{p},\mathtt{u},r\rangle]$
$\exists C_2$
$chunks\ r\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
q = p->ptr
$\boxed{\mathtt{p}\mid \mathtt{q}}^{\,\mathtt{q}}$
$chunks\ \mathtt{q}\ \mathtt{t}\ C_2$
Unfold $chunks$
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,]) \vee$
$(\exists r, \tau.\ chunk_\tau\ \mathtt{q}\ r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\tau,r\rangle]))$
Unfold $chunk_{\mathtt{u}}$ and $chunk_{\mathtt{a}}$; combine
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,] * \boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare}) \vee$
$(\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}\mid r}^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{u},r\rangle])) \vee$
$(\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \mathtt{q} \dot{<} r * \boxed{\mathtt{q}\mid r_\blacksquare} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{a},r\rangle]))$
Factor out q-cell
$\exists n$
$\boxed{\mathtt{q}\mid n}$
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,] * n \doteq \mathtt{s}_\blacksquare) \vee$
$(\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}+1}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{u},r\rangle])) \vee$
$(\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \mathtt{q} \dot{<} r * n \doteq r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{a},r\rangle]))$
while(!testbusy(q->ptr)) {
$\boxed{\mathtt{q}\mid n} \wedge n \in \mathbb{N}$
Reject first and third disjuncts (they contradict $n \in \mathbb{N}$)
〃
$\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}+1}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{u},r\rangle])$
Combine
$\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}\mid r}^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{u},r\rangle])$
Split
$\exists r$
$\boxed{\mathtt{q}\mid r}^{\,r}$
$chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q},\mathtt{u},r\rangle])$
$\boxed{\mathtt{t}\mid \mathtt{s}_\blacksquare}$

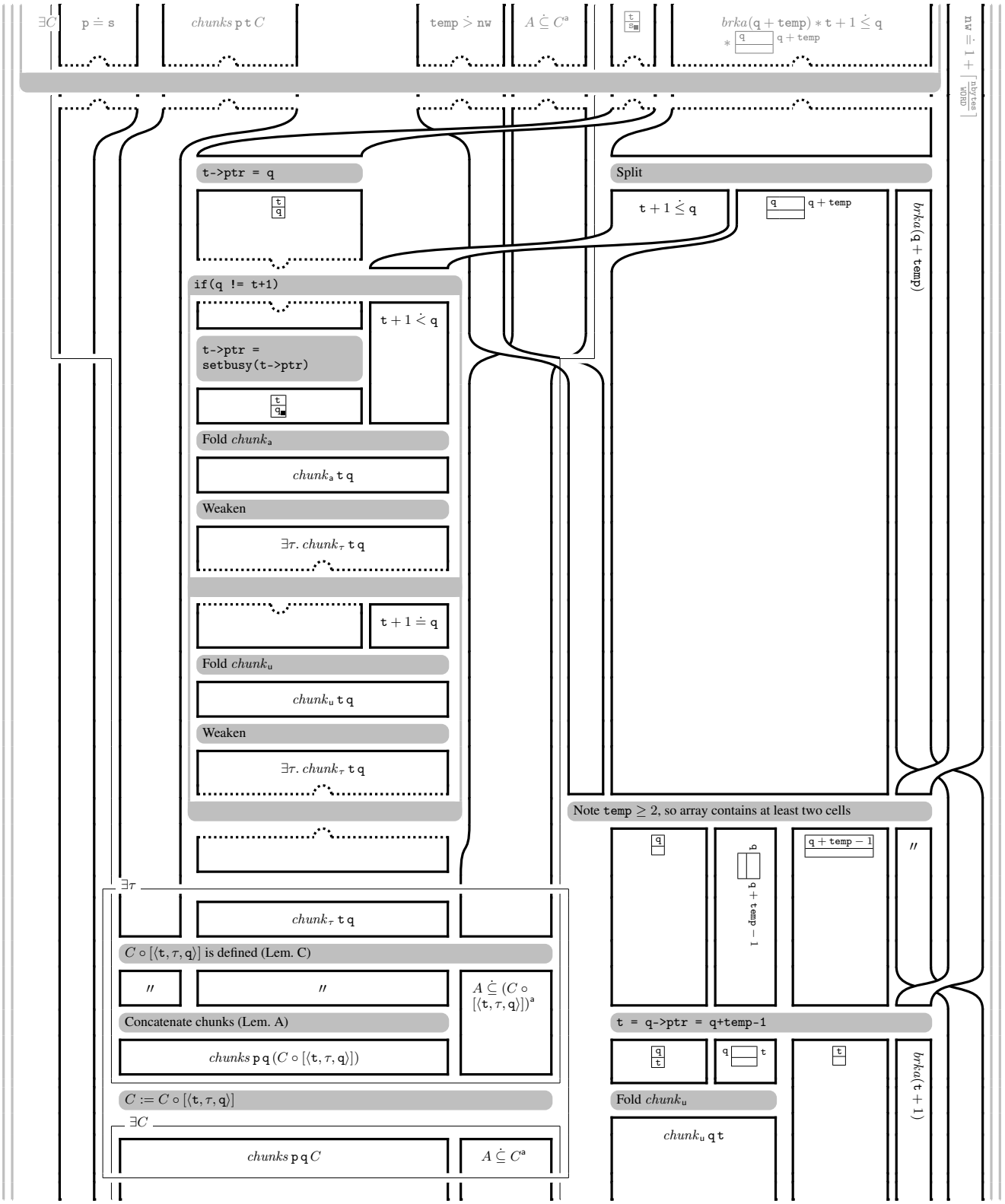

$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{p}\ C_1$
$\exists r$
$\exists C_2$
$chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle])$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
Note $C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle]$ is defined
p->ptr = q->ptr
//
$A \dot{\subseteq} (C_1 \circ (C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle]))^{\mathtt{a}}$
Weaken
$C_2 := C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle]$
$\exists C_2$
$chunks\ r\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
q = p->ptr
$chunks\ \mathtt{q}\ \mathtt{t}\ C_2$
Unfold $chunk$s
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,]) \vee$
$(\exists r, \tau.\ chunk_\tau\ \mathtt{q}\ r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \tau, r\rangle]))$
Unfold $chunk_{\mathtt{u}}$ and $chunk_{\mathtt{a}}$; combine
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,] * \boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare}) \vee$
$(\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}}\ \boxed{r}^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle])) \vee$
$(\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare} * \exists r.\ \mathtt{q} \dot{<} r * \boxed{\mathtt{q}}\ \boxed{r_\blacksquare} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{a}, r\rangle]))$
Factor out q-cell
$\exists n$
$\boxed{\mathtt{q}}\ \boxed{n\mathtt{t}}$
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,] * n \doteq \mathtt{s}_\blacksquare) \vee$
$(\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}+1}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle]))$
$\vee\ (\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare} * \exists r.\ \mathtt{q} \dot{<} r * n \doteq r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{a}, r\rangle]))$
}
//
Fold $chunk_{\mathtt{u}}$
Combine; use equalities
$chunk_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q}$
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [\,] * \boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare}) \vee$
$(\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare} * \exists r.\ \boxed{\mathtt{q}}\ \boxed{r}^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{u}, r\rangle])) \vee$
$(\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare} * \exists r.\ \mathtt{q} \dot{<} r * \boxed{\mathtt{q}}\ \boxed{r_\blacksquare} * chunks\ r\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \mathtt{a}, r\rangle]))$
Factor out t-cell; fold $chunk_{\mathtt{a}}$, $chunk_{\mathtt{u}}$ and $chunks$
$chunks\ \mathtt{q}\ \mathtt{t}\ C_2$
$\boxed{\mathtt{t}}\ \boxed{\mathtt{s}_\blacksquare}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
if(q >= p+nw && p+nw >= p)
$\mathtt{q} \geq \mathtt{p} + \mathtt{nw}$
goto found

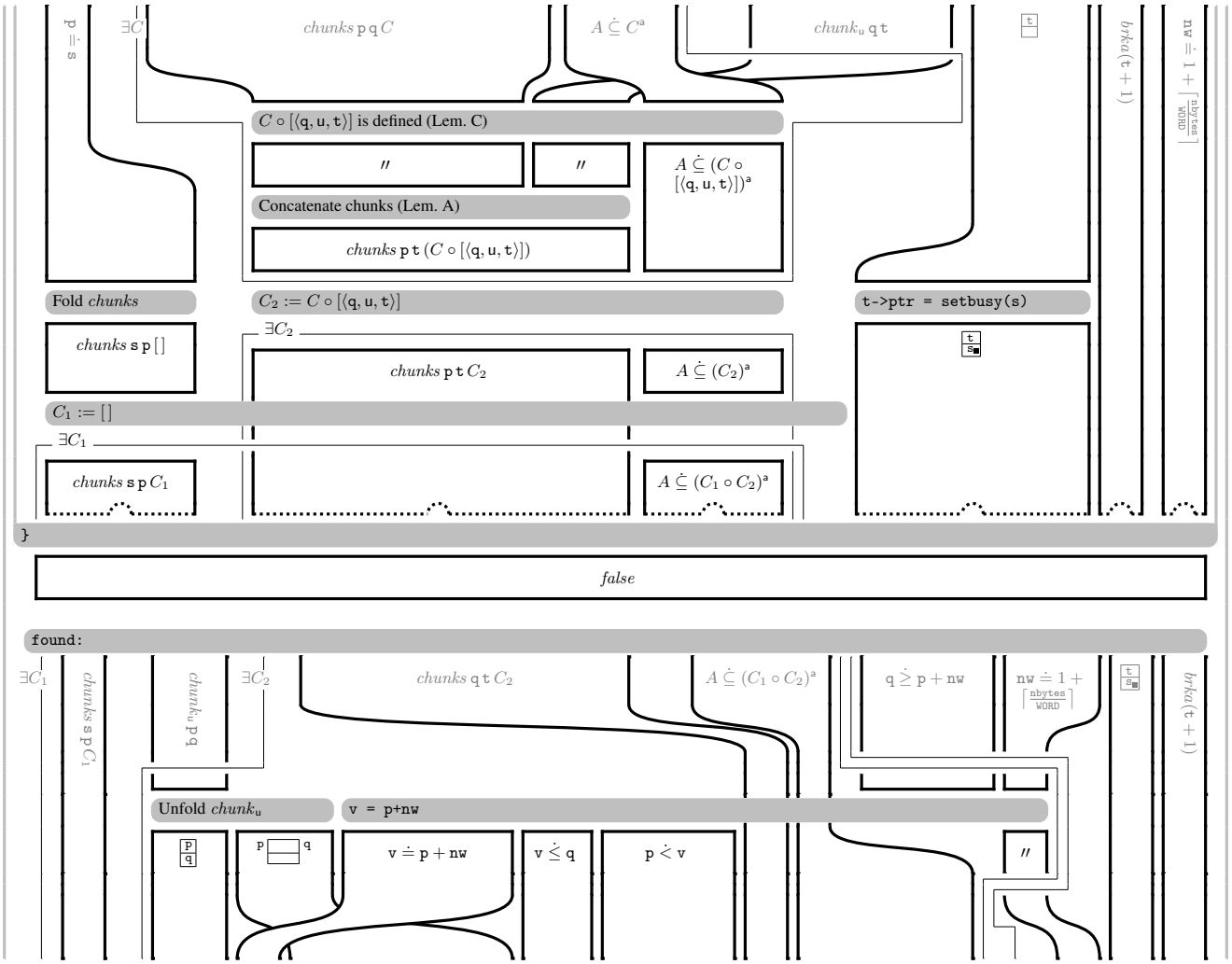

$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{p}\ C_1$
$chunk_{\mathsf{u}}\ \mathtt{p}\ \mathtt{q}$
$\exists C_2$
$chunks\ \mathtt{q}\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathsf{a}}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
$C_1 \circ [\langle \mathtt{p}, \mathtt{u}, \mathtt{q} \rangle] \circ C_2$ is defined (Lem. C)
〃
〃
〃
$A \dot{\subseteq} (C_1 \circ [\langle \mathtt{p}, \mathtt{u}, \mathtt{q} \rangle] \circ C_2)^{\mathsf{a}}$
Concatenate chunks (Lem. A)
$chunks\ \mathtt{p}\ \mathtt{t}\ ([\langle \mathtt{p}, \mathtt{u}, \mathtt{q} \rangle] \circ C_2)$
$C_2 := [\langle \mathtt{p}, \mathtt{u}, \mathtt{q} \rangle] \circ C_2$
$\exists C_2$
$chunks\ \mathtt{p}\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathsf{a}}$
}
$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{p}\ C_1$
$\exists C_2$
$\exists n$
$(\mathtt{p} \doteq \mathtt{t} * C_2 \doteq [] * n \doteq \mathtt{s}_\blacksquare) \vee$
$(\mathtt{t}/\mathtt{s}_\blacksquare * \exists r.\ \boxed{\mathtt{p}+1}^{\,r} * n \doteq r * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{u}, r \rangle])) \vee$
$(\mathtt{t}/\mathtt{s}_\blacksquare * \exists r.\ \mathtt{p} \dot{<} r * n \doteq r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{a}, r \rangle]))$
Combine; use equalities
$(\mathtt{p} \doteq \mathtt{t} * C_2 \doteq [] * \mathtt{t}/\mathtt{s}_\blacksquare) \vee (\mathtt{t}/\mathtt{s}_\blacksquare * \exists r.\ \mathtt{p} \dot{<} r * \mathtt{p}/r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{p}, \mathtt{a}, r \rangle]))$
Factor out t-cell; fold $chunk_{\mathsf{a}}$ and $chunks$
$chunks\ \mathtt{p}\ \mathtt{t}\ C_2$
$\mathtt{t}/\mathtt{s}_\blacksquare$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathsf{a}}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
q = p
$chunks\ \mathtt{s}\ \mathtt{q}\ C_1$
$chunks\ \mathtt{q}\ \mathtt{t}\ C_2$
$\mathtt{q} \doteq \mathtt{p}$
Unfold $chunks$
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq []) \vee (\exists r, \tau.\ chunk_\tau\ \mathtt{q}\ r * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \tau, r \rangle]))$
Unfold $chunk_{\mathsf{u}}$ and $chunk_{\mathsf{a}}$; combine with t-cell; use equalities
$(\mathtt{q} \doteq \mathtt{t} * C_2 \doteq [] * \mathtt{p}/\mathtt{s}_\blacksquare) \vee$
$(\mathtt{t}/\mathtt{s}_\blacksquare * \exists r.\ \boxed{\mathtt{q}\ \ }^{\,r} * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \mathtt{u}, r \rangle])) \vee$
$(\mathtt{t}/\mathtt{s}_\blacksquare * \exists r.\ \mathtt{q} \dot{<} r * \mathtt{q}/r_\blacksquare * chunks\ r\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \mathtt{a}, r \rangle]))$
〃
p = clearbusy(p->ptr)
$(\mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * C_2 \doteq [] * \mathtt{t}/\mathtt{s}_\blacksquare) \vee$
$(\mathtt{t}/\mathtt{s}_\blacksquare * \boxed{\mathtt{q}\ \ }^{\,\mathtt{p}} * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \mathtt{u}, \mathtt{p} \rangle])) \vee$
$(\mathtt{t}/\mathtt{s}_\blacksquare * \mathtt{q} \dot{<} \mathtt{p} * \mathtt{q}/\mathtt{p}_\blacksquare * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \mathtt{a}, \mathtt{p} \rangle]))$
Factor out t-cell; fold $chunk_{\mathsf{u}}$ and $chunk_{\mathsf{a}}$
$(\mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * C_2 \doteq []) \vee \exists \tau.\ chunk_\tau\ \mathtt{q}\ \mathtt{p} * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \tau, \mathtt{p} \rangle])$
$\mathtt{t}/\mathtt{s}_\blacksquare$

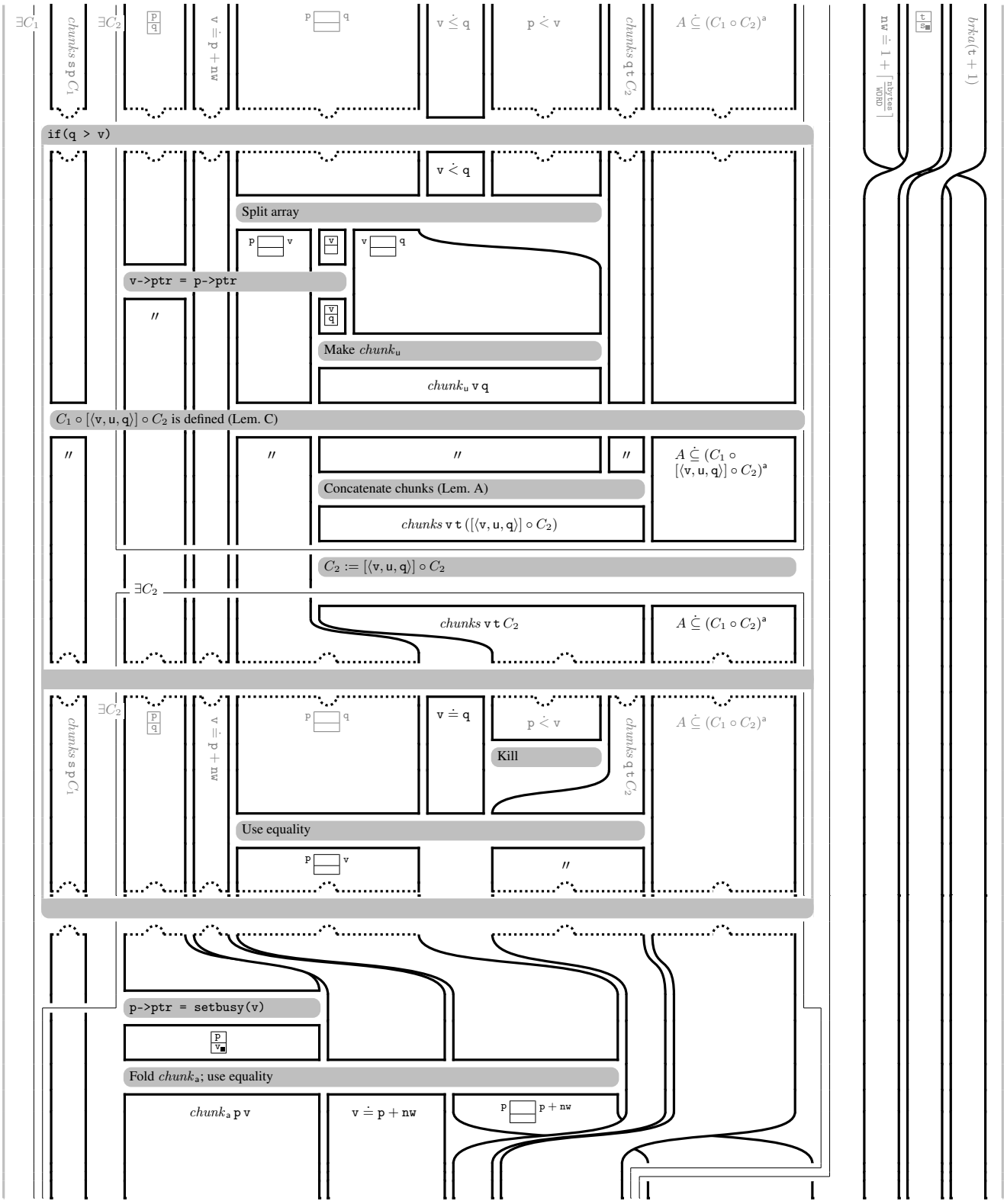

$\exists C_1$ $\exists C_2$
$chunks\ \mathtt{s}\ \mathtt{q}\ C_1$
$(\mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * C_2 \doteq [\,]) \vee \exists \tau.\ chunk_\tau\ \mathtt{q}\ \mathtt{p} * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle])$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
if(p > q)
$\mathtt{p} \dot{>} \mathtt{q}$
$\mathtt{p} \dot{>} \mathtt{q}$ contradicts $chunks\ \mathtt{s}\ \mathtt{q}\ C_1 * \mathtt{p} \doteq \mathtt{s}$
$chunks\ \mathtt{s}\ \mathtt{q}\ C_1$
$\exists \tau.\ chunk_\tau\ \mathtt{q}\ \mathtt{p} * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle])$
$\exists \tau$
$chunk_\tau\ \mathtt{q}\ \mathtt{p} * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle])$
$C_1 \circ [\langle \mathtt{q}, \tau, \mathtt{p} \rangle]$ is defined (Lem. C)
〃
〃
$A \dot{\subseteq} ((C_1 \circ [\langle \mathtt{q}, \tau, \mathtt{p} \rangle]) \circ (C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle]))^{\mathtt{a}}$
Append q chunk to end of C_1 (Lem. A)
$chunks\ \mathtt{s}\ \mathtt{p}\ (C_1 \circ [\langle \mathtt{q}, \tau, \mathtt{p} \rangle]) * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle])$
$C_1 := C_1 \circ [\langle \mathtt{q}, \tau, \mathtt{p} \rangle]; C_2 := C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle]$
$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{p}\ C_1$
$\exists C_2$
$chunks\ \mathtt{p}\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
else
$\exists C_1$ $\exists C_2$
$chunks\ \mathtt{s}\ \mathtt{q}\ C_1$
$(\mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * C_2 \doteq [\,])$
$\vee \exists \tau.\ chunk_\tau\ \mathtt{q}\ \mathtt{p} * chunks\ \mathtt{p}\ \mathtt{t}\ (C_2 \multimap [\langle \mathtt{q}, \tau, \mathtt{p} \rangle])$
$\mathtt{p} \dot{\leq} \mathtt{q}$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
$\mathtt{p} \leq \mathtt{q}$ contradicts $chunk_\tau\ \mathtt{q}\ \mathtt{p}$
$\mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * C_2 \doteq [\,]$
if(q != t || p != s)
$\mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * C_2 \doteq [\,] * (\mathtt{q} \dot{\neq} \mathtt{t} \vee \mathtt{p} \dot{\neq} \mathtt{s})$
Propagate contradiction across entire interface
$false$

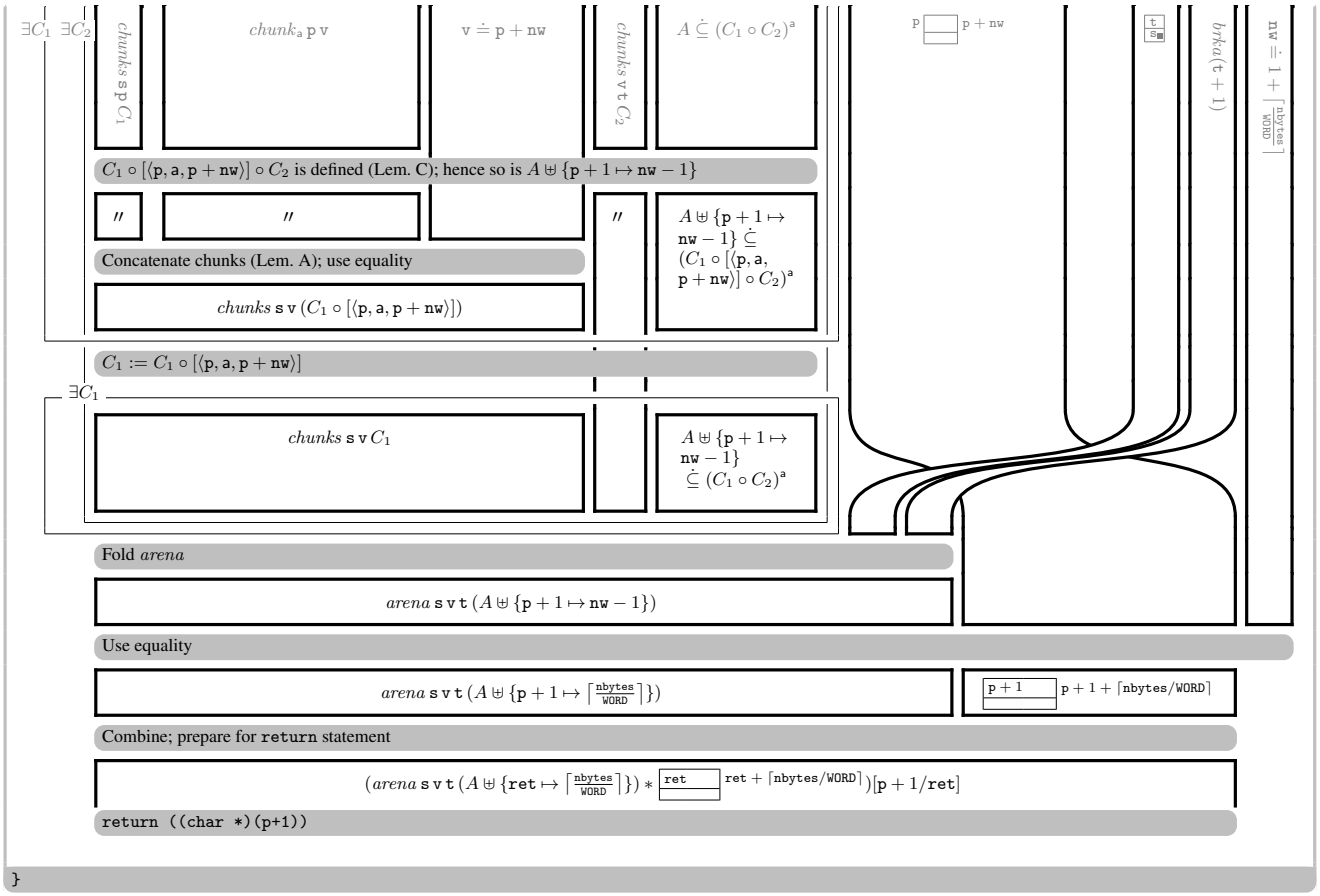

false
return 0
false
else
∃C1 ∃C2
chunks s q C1
q ≐ t ∗ p ≐ s ∗ C2 ≐ []
A ⊑̇ (C1 ∘ C2)a
brka(t + 1)
nw ≐ 1 + ⌈nbytes / WORD⌉
Use equalities; weaken
chunks p t C1
p ≐ s
A ⊑̇ (C1)a
if(++temp > 1)
C := C1
∃C
chunks p t C
A ⊑̇ Ca
break
false
∃C1 ∃C2
p ≐ s
chunks p t C1
A ⊑̇ (C1)a
Fold chunks
chunks s p []
C2 := C1; C1 := []
∃C1
chunks s p C1
∃C2
chunks p t C2
A ⊑̇ (C1 ∘ C2)a
}
false

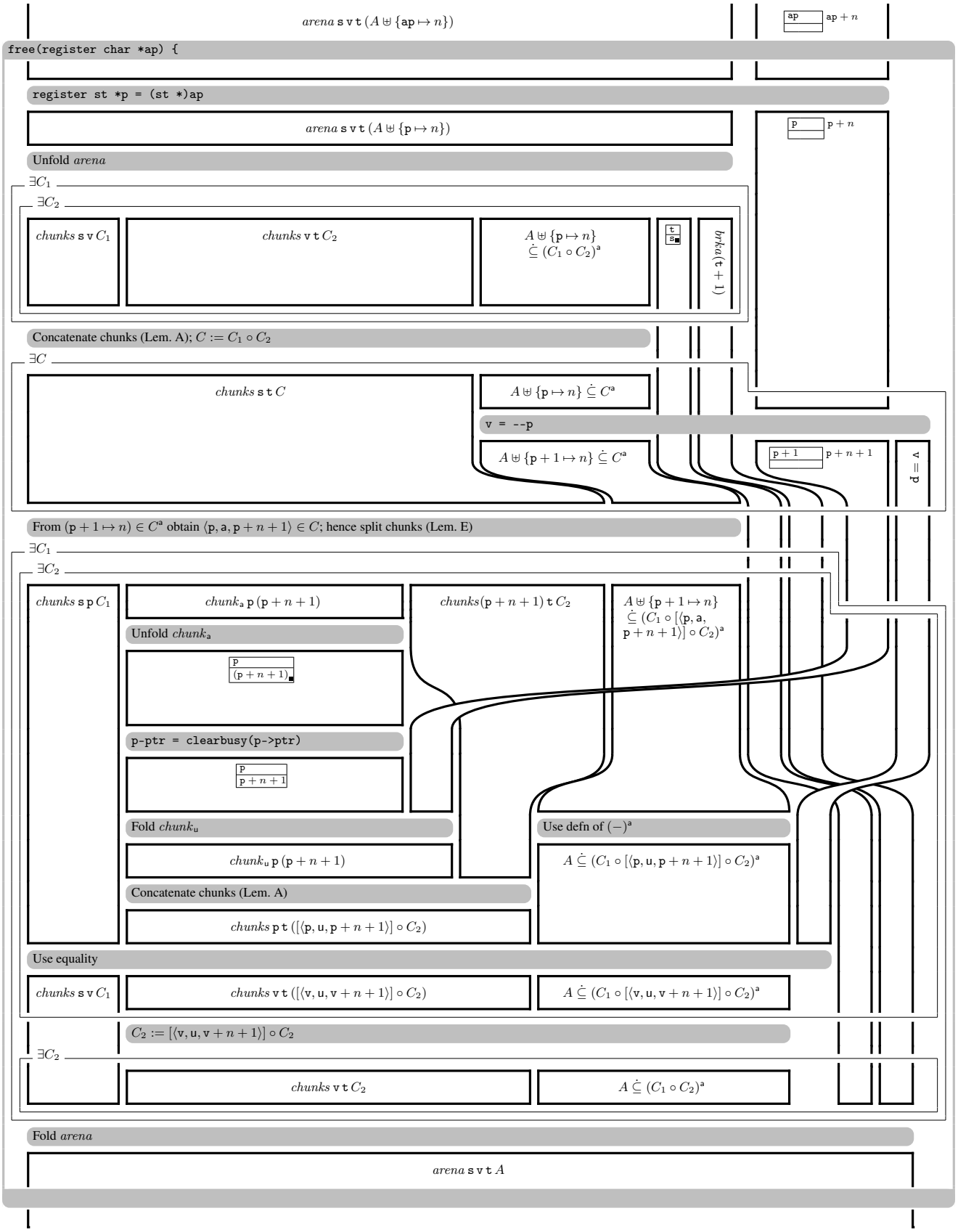

Continue from break
$\exists C$
$\mathtt{p} \doteq \mathtt{s}$
$chunks\ \mathtt{p}\ \mathtt{t}\ C$
$A \dot{\subseteq} C^{\mathsf{a}}$
$\boxed{\mathtt{t}}\ \boxed{\mathtt{s}\blacksquare}$
$brka(\mathtt{t}+1)$
$\mathtt{nw} \doteq 1 + \lfloor \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rfloor$
temp = ((nw+BLOCK/WORD)/ (BLOCK/WORD))*(BLOCK/WORD)
$\mathtt{temp} \dot{>} \mathtt{nw}$
if(q+temp < q)
$\mathtt{q} + \mathtt{temp} \dot{<} \mathtt{q}$
Integer overflows not modelled
false
return 0
false
$\mathtt{temp} \dot{>} \mathtt{nw}$
q = (st *)sbrk(0)
$brka(\mathtt{q})$
$\mathtt{q} \dot{\geq} \mathtt{t}+1$
Use Lem. F
$brka(\mathtt{t}+1)$
q = (st *)sbrk(temp*WORD)
$((brka(\mathtt{t}+1) * \mathtt{q} \doteq -1/\mathtt{WORD})$
$\vee (brka(\mathtt{q}+\mathtt{temp}) * \mathtt{t}+1 \dot{\leq} \mathtt{q}$
$* \boxed{\mathtt{q}}^{\mathtt{q}+\mathtt{temp}}))$
if((int)q == -1) {
Use equality
Kill
Fold $chunks$
$chunks\ \mathtt{s}\ \mathtt{t}\ C$
$chunks\ \mathtt{s}\ \mathtt{s}\ []$
v = s //line added to fix bug
$chunks\ \mathtt{s}\ \mathtt{v}\ []$
$chunks\ \mathtt{v}\ \mathtt{t}\ C$
$brka(\mathtt{t}+1)$
$C_1 := []; C_2 := C$
$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{v}\ C_1$
$\exists C_2$
$chunks\ \mathtt{v}\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathsf{a}}$
Fold $arena$
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A$
Prepare for return
$(arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A * \mathtt{ret} \doteq 0)[0/\mathtt{ret}]$
return 0
false
}
$\exists C$
$\mathtt{p} \doteq \mathtt{s}$
$chunks\ \mathtt{p}\ \mathtt{t}\ C$
$\mathtt{temp} \dot{>} \mathtt{nw}$
$A \dot{\subseteq} C^{\mathsf{a}}$
$\boxed{\mathtt{t}}\ \boxed{\mathtt{s}\blacksquare}$
$brka(\mathtt{q}+\mathtt{temp}) * \mathtt{t}+1 \dot{\leq} \mathtt{q}$
$* \boxed{\mathtt{q}}^{\mathtt{q}+\mathtt{temp}}$


∃C
p ≐ s
chunks p t C
temp ≥̇ nw
A ⊆̇ C^a
brka(q + temp) ∗ t + 1 ≤̇ q
∗ q | q + temp
nw ≐ 1 + ⌈ nbytes / WORD ⌉
t->ptr = q
t | q
Split
t + 1 ≤̇ q
q | q + temp
brka(q + temp)
if(q != t+1)
t + 1 <̇ q
t->ptr = setbusy(t->ptr)
t | q
Fold chunk_a
chunk_a t q
Weaken
∃τ. chunk_τ t q
t + 1 ≐ q
Fold chunk_u
chunk_u t q
Weaken
∃τ. chunk_τ t q
Note temp ≥ 2, so array contains at least two cells
q
q | q + temp − 1
q + temp − 1
q + temp − 1
″
∃τ
chunk_τ t q
C ∘ [⟨t, τ, q⟩] is defined (Lem. C)
″
″
A ⊆̇ (C ∘ [⟨t, τ, q⟩])^a
Concatenate chunks (Lem. A)
chunks p q (C ∘ [⟨t, τ, q⟩])
t = q->ptr = q+temp-1
q | t
q | t | t
t
brka(t + 1)
Fold chunk_u
chunk_u q t
C := C ∘ [⟨t, τ, q⟩]
∃C
chunks p q C
A ⊆̇ C^a


p ≟ s
∃C
chunks p q C
A ⊑̇ C^a
chunk_u q t
t
brka(t + 1)
nw ≟ 1 + ⌈nbytes / WORD⌉
C ∘ [⟨q, u, t⟩] is defined (Lem. C)
〃
〃
A ⊑̇ (C ∘ [⟨q, u, t⟩])^a
Concatenate chunks (Lem. A)
chunks p t (C ∘ [⟨q, u, t⟩])
Fold chunks
C₂ := C ∘ [⟨q, u, t⟩]
t->ptr = setbusy(s)
chunks s p []
∃C₂
chunks p t C₂
A ⊑̇ (C₂)^a
C₁ := []
∃C₁
chunks s p C₁
A ⊑̇ (C₁ ∘ C₂)^a
}
false
found:
∃C₁
chunks s p C₁
chunk_u p q
∃C₂
chunks q t C₂
A ⊑̇ (C₁ ∘ C₂)^a
q ≥̇ p + nw
nw ≟ 1 + ⌈nbytes / WORD⌉
brka(t + 1)
Unfold chunk_u
v = p+nw
v ≟ p + nw
v ≤̇ q
p <̇ v
〃


if(q > v)
Split array
v->ptr = p->ptr
Make $chunk_u$
$chunk_u$ v q
$C_1 \circ [\langle v, u, q \rangle] \circ C_2$ is defined (Lem. C)
Concatenate chunks (Lem. A)
$C_2 := [\langle v, u, q \rangle] \circ C_2$
Kill
Use equality
p->ptr = setbusy(v)
Fold $chunk_a$; use equality
$chunk_a$ p v
v ≐ p + nw
p + nw
nw ≐ 1 + ⌈nbytes/WORD⌉
brka(t + 1)


$\exists C_1$ $\exists C_2$
$chunks\ \mathtt{s}\ \mathtt{p}\ C_1$
$chunk_{\mathtt{a}}\ \mathtt{p}\ \mathtt{v}$
$\mathtt{v} \doteq \mathtt{p} + \mathtt{nw}$
$chunks\ \mathtt{v}\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
$C_1 \circ [\langle \mathtt{p}, \mathtt{a}, \mathtt{p} + \mathtt{nw} \rangle] \circ C_2$ is defined (Lem. C); hence so is $A \uplus \{\mathtt{p} + 1 \mapsto \mathtt{nw} - 1\}$
〃
〃
〃
$A \uplus \{\mathtt{p} + 1 \mapsto \mathtt{nw} - 1\} \dot{\subseteq} (C_1 \circ [\langle \mathtt{p}, \mathtt{a}, \mathtt{p} + \mathtt{nw} \rangle] \circ C_2)^{\mathtt{a}}$
Concatenate chunks (Lem. A); use equality
$chunks\ \mathtt{s}\ \mathtt{v}\ (C_1 \circ [\langle \mathtt{p}, \mathtt{a}, \mathtt{p} + \mathtt{nw} \rangle])$
$C_1 := C_1 \circ [\langle \mathtt{p}, \mathtt{a}, \mathtt{p} + \mathtt{nw} \rangle]$
$\exists C_1$
$chunks\ \mathtt{s}\ \mathtt{v}\ C_1$
$A \uplus \{\mathtt{p} + 1 \mapsto \mathtt{nw} - 1\} \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
$\mathtt{p}$ $\mathtt{p} + \mathtt{nw}$
$\mathtt{t}$
$brka(\mathtt{t} + 1)$
$\mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil$
Fold $arena$
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{p} + 1 \mapsto \mathtt{nw} - 1\})$
Use equality
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{p} + 1 \mapsto \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil\})$
$\mathtt{p} + 1$ $\mathtt{p} + 1 + \lceil \mathtt{nbytes}/\mathtt{WORD} \rceil$
Combine; prepare for `return` statement
$(arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil\}) * \mathtt{ret}\ \mathtt{ret} + \lceil \mathtt{nbytes}/\mathtt{WORD} \rceil)[\mathtt{p} + 1/\mathtt{ret}]$
`return ((char *)(p+1))`
`}`
Establish postcondition
$(arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \lceil \frac{\mathtt{nbytes}}{\mathtt{WORD}} \rceil\}) * \mathtt{ret}\ \mathtt{ret} + \lceil \mathtt{nbytes}/\mathtt{WORD} \rceil) \vee (arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A * \mathtt{ret} \doteq 0)$


$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ap} \mapsto n\})$
ap
ap + n
free(register char *ap) {
register st *p = (st *)ap
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{p} \mapsto n\})$
p
p + n
Unfold arena
$\exists C_1$
$\exists C_2$
$chunks\ \mathtt{s}\ \mathtt{v}\ C_1$
$chunks\ \mathtt{v}\ \mathtt{t}\ C_2$
$A \uplus \{\mathtt{p} \mapsto n\} \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
t
s
$brka(\mathtt{t} + 1)$
Concatenate chunks (Lem. A); $C := C_1 \circ C_2$
$\exists C$
$chunks\ \mathtt{s}\ \mathtt{t}\ C$
$A \uplus \{\mathtt{p} \mapsto n\} \dot{\subseteq} C^{\mathtt{a}}$
v = --p
$A \uplus \{\mathtt{p} + 1 \mapsto n\} \dot{\subseteq} C^{\mathtt{a}}$
p + 1
p + n + 1
v = p
From $(\mathtt{p} + 1 \mapsto n) \in C^{\mathtt{a}}$ obtain $\langle \mathtt{p}, \mathtt{a}, \mathtt{p} + n + 1 \rangle \in C$; hence split chunks (Lem. E)
$\exists C_1$
$\exists C_2$
$chunks\ \mathtt{s}\ \mathtt{p}\ C_1$
$chunk_{\mathtt{a}}\ \mathtt{p}\ (\mathtt{p} + n + 1)$
$chunks(\mathtt{p} + n + 1)\ \mathtt{t}\ C_2$
$A \uplus \{\mathtt{p} + 1 \mapsto n\} \dot{\subseteq} (C_1 \circ [\langle \mathtt{p}, \mathtt{a}, \mathtt{p} + n + 1 \rangle] \circ C_2)^{\mathtt{a}}$
Unfold $chunk_{\mathtt{a}}$
p
(p + n + 1)
p-ptr = clearbusy(p->ptr)
p
p + n + 1
Fold $chunk_{\mathtt{u}}$
Use defn of $(-)^{\mathtt{a}}$
$chunk_{\mathtt{u}}\ \mathtt{p}\ (\mathtt{p} + n + 1)$
$A \dot{\subseteq} (C_1 \circ [\langle \mathtt{p}, \mathtt{u}, \mathtt{p} + n + 1 \rangle] \circ C_2)^{\mathtt{a}}$
Concatenate chunks (Lem. A)
$chunks\ \mathtt{p}\ \mathtt{t}\ ([\langle \mathtt{p}, \mathtt{u}, \mathtt{p} + n + 1 \rangle] \circ C_2)$
Use equality
$chunks\ \mathtt{s}\ \mathtt{v}\ C_1$
$chunks\ \mathtt{v}\ \mathtt{t}\ ([\langle \mathtt{v}, \mathtt{u}, \mathtt{v} + n + 1 \rangle] \circ C_2)$
$A \dot{\subseteq} (C_1 \circ [\langle \mathtt{v}, \mathtt{u}, \mathtt{v} + n + 1 \rangle] \circ C_2)^{\mathtt{a}}$
$C_2 := [\langle \mathtt{v}, \mathtt{u}, \mathtt{v} + n + 1 \rangle] \circ C_2$
$\exists C_2$
$chunks\ \mathtt{v}\ \mathtt{t}\ C_2$
$A \dot{\subseteq} (C_1 \circ C_2)^{\mathtt{a}}$
Fold arena
$arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A$