

Ribbon Proofs for Separation Logic

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

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

Matthew Parkinson
Microsoft Research Cambridge
mattpark@microsoft.com

Abstract—We present a diagrammatic system for constructing and presenting readable program proofs in separation logic.

I. INTRODUCTION

A program proof should not merely certify that a program is correct; it should explain *why* it is correct. By examining a proof, one should gain understanding of both the program being considered and the proof technique being used. To achieve this, the proof must first be *intelligible*. This paper presents a system that produces intelligible program proofs.

A proof in Hoare logic in general [1], or separation logic in particular [2], [3], is formally a derivation tree that uses the logic’s axioms and rules. As a visual representation of the proof, however, derivation trees are highly repetitious, spatially inefficient and laborious to read. Hence, the de facto standard for presenting program proofs is the *proof outline* (e.g. Fig. 1a), in which the program’s instructions are interspersed with ‘enough’ assertions to allow the reader to reconstruct the derivation tree.

The proof outline exhibits two problems of its own. Firstly, there remains much repetition. The assertion ‘ $x \mapsto 1$ ’, for instance, appears three times. Secondly, there is no distinction between those parts of an assertion that are affected by an instruction and those that are merely in what separation logic calls the *frame*. For instance, in the second assertion, the first and third conjuncts are not affected by the upcoming ‘ $[y] := 1$ ’, yet they remain visually indistinct from the second, which is. Hence, it is difficult to deduce the effect of each instruction: one must parse the assertion before it, parse the assertion after it, and then compute the difference.

Of course, these are only minor problems in our toy example, but they quickly become devastating when scaled to large programs. Indeed, the authors have constructed proof outlines with assertions spanning six or seven lines, and found the process prohibitively difficult.

Note that separation logic’s *frame rule* does little to ameliorate the situation. We could depict how it is invoked at each instruction; the second assignment would then be written:

$$\text{frame} \quad \left[\begin{array}{l} \{x \mapsto 1 * y \mapsto 0 * z \mapsto 0\} \\ \{y \mapsto 0\} \\ [y] := 1; \\ \{y \mapsto 1\} \\ \{x \mapsto 1 * y \mapsto 1 * z \mapsto 0\}. \end{array} \right] \quad (\dagger)$$

This clarifies the effect of each instruction, but the extra ink brings more repetition, more spatial inefficiency and more

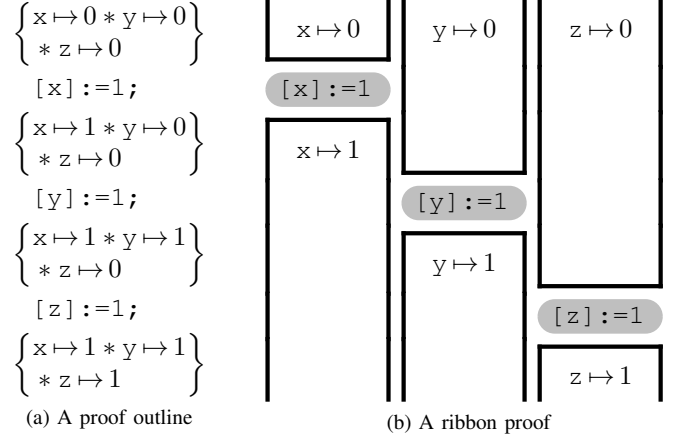


Fig. 1. A toy example

proof text to be verified by the reader. The frame rule is better applied across several instructions. For instance, we might frame away z during the first two assignments, or x during the final two. But we cannot do both, because applications of the frame rule cannot overlap; they must be well-nested.

Our solution is to recast the proof as a *ribbon proof* (Fig. 1b). The repetition has disappeared, and each instruction’s effect is clear: it affects exactly those assertions directly above and below it, while framed assertions bypass to the left or right. As in (\dagger), we invoke the frame rule at each instruction, but having distributed the resources (memory cells) along the horizontal axis of the diagram, doing so increases neither the size of the proof nor the burden on the reader. That the frame rule has been correctly applied is visually evident.

A bonus of this particular ribbon proof is that it emphasises the fact that the three assignments update separate memory cells. They are thus independent, and amenable to reordering or parallelisation. Indeed, one can imagine obtaining a proof of the transformed program by simply sliding the first column downward and the third column upward. The traditional proof outline neither suggests nor supports such manoeuvres. Section IV explains how a shift to the *variables-as-resource* paradigm might enable ribbon proofs to be used to study such parallelisation opportunities.

In Fig. 1 and throughout this paper, the ribbon proof is larger than the corresponding proof outline. Nonetheless, because ribbon proofs avoid the redundancy endemic in proof outlines, they are, in fact, a more scalable proof representation. Appendix B showcases the ability of our diagrams to present

```

{ls x 0 * ls y 0}
if (x==0) x:=y;
else {
  t:=x; u:=[t];
  while {ls x t * t ↦ u * ls u 0} (u!=0) {
    t:=u; u:=[t];
  }
  [t]:=y;
}
{ls x 0}

```

Fig. 2. Code, specification and loop invariant for list append

readable proofs of programs of non-trivial size and complexity by presenting a ribbon proof of the memory manager from Version 7 Unix. Although the proof is large, covering thirteen pages, it remains easy to read. A proof outline providing the same level of detail would be extremely tedious. The ribbon proof is checked ‘locally’; that is, by focusing on each basic step in turn, and verifying: (i) that the ribbons above the command, the command itself, and the ribbons below the command comprise a valid Hoare triple, and (ii) that no ribbon or command directly to the left or right reads a variable that is written in the basic step.

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

Background: Ribbon proofs were introduced by Bean [4] as a proof system for the propositional fragment of bunched implications logic (BI) [5]. Because BI is the basis of the assertion language used in separation logic [2], 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. Both entailments, $p \Rightarrow q$, and Hoare triples, $\{p\}c\{q\}$, can now be proved within the same system. Bean’s ribbon proof system extends, in turn, Fitch’s *box proofs* for first-order logic [6].

II. INFORMAL DEVELOPMENT

We describe our ribbon proof system using two examples.

Example 1: List append

Figure 2 presents a program that appends two lists, a (rather weak) specification, and a loop invariant (all adapted from [7]). A ribbon proof of this program is shown in Fig. 3.

The proof advances down the vertical dimension of the diagram, and the resources being operated upon (that is, the memory cells) are distributed horizontally. The precondition $ls\ x\ 0 * ls\ y\ 0$ 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

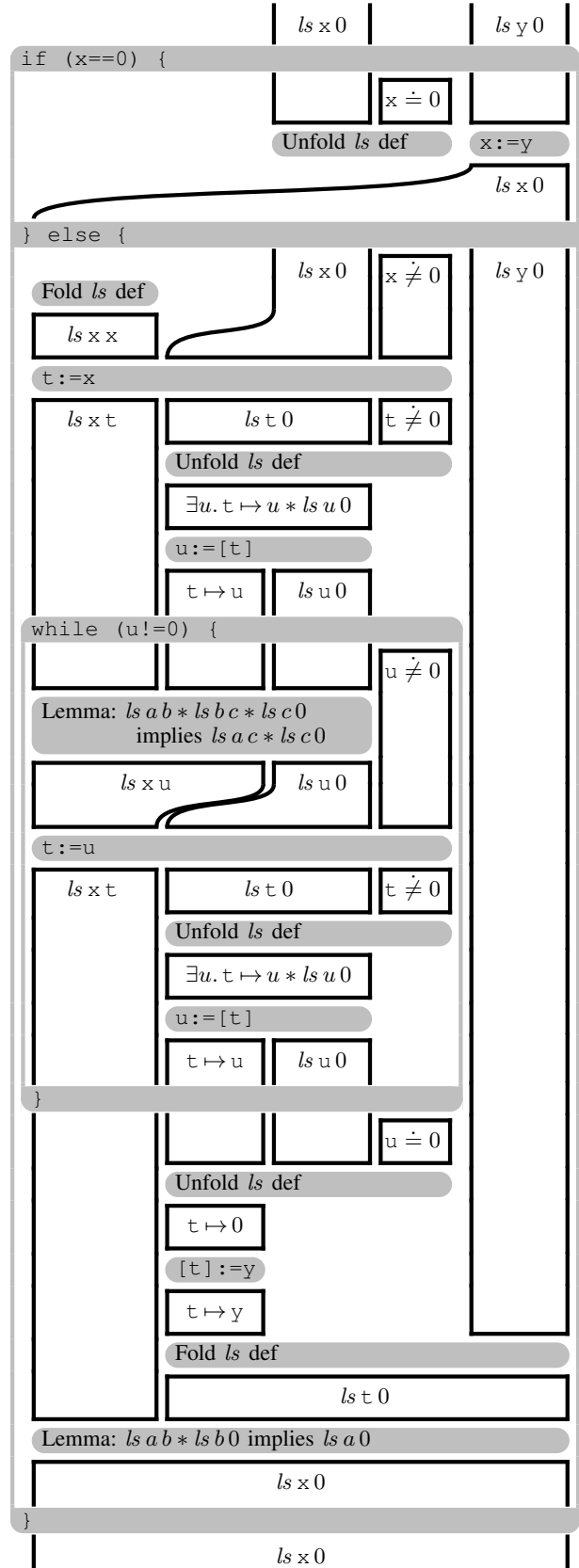


Fig. 3. Ribbon proof of list append

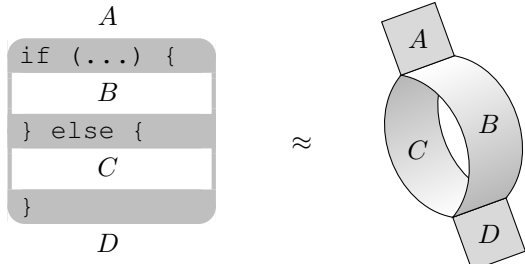


Fig. 4. If-statements, pictorially

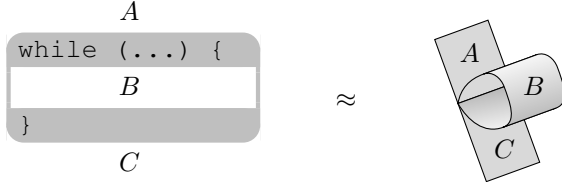


Fig. 5. While-loops, pictorially

between the ribbons. The resource distribution is not uniform, so the width of a ribbon is not proportional to the amount of resource it describes. In particular, the assertion $x \doteq 0$, obtained as an assumption upon entering the first branch of the if-statement, describes no memory cells at all and is merely a fact about variables. (For each relational symbol r , we define $x \dot{r} y \stackrel{\text{def}}{=} x r y \wedge \text{emp}$.) In general, we are free to stretch ribbons horizontally as necessitated by the layout, and also to reorder ribbons via a ‘twist’ operation (see Fig. 7), which is justified by the commutativity of $*$.

The next step in the proof is to unfold the definition of the ls predicate, which is the smallest satisfying:

$$ls \ x \ y \Leftrightarrow (x \doteq y \vee x \neq y * \exists x'. x \mapsto x' * ls \ x' \ y).$$

The combination of $ls \ x \ 0$ and $x \doteq 0$ implies emp . A ribbon labelled ‘ emp ’ is equivalent to a gap in the diagram. Simultaneously with this reasoning, we perform the assignment to x , which transforms $ls \ y \ 0$ into $ls \ x \ 0$. We then expand this ribbon to match the width of the identically-labelled ribbon that leaves the second branch of the if-statement. The general rule here is that the collections of ribbons entering the first and second branches of an if-statement must match, as must those at the two exits, so that the proof could be cut and folded into the three-dimensional shape suggested by Fig. 4.

The second-branch of the if-statement begins by assuming that the test condition fails. We repeat the labels on the other two ribbons, $ls \ x \ 0$ and $ls \ y \ 0$, to aid readability. The assertion $ls \ y \ 0$ is not needed until nearly the end of this branch, when it is merged with $t \mapsto y$. A proof outline would either temporarily remove this assertion via an explicit application of the frame rule, or else redundantly repeat it 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*.

The while-loop has a similar proof structure to the if-statement. Inside the loop body we assume that the test succeeds ($u \neq 0$); the complementary assumption appears after

```

1 {list δ x}
2 y:=0;
3 while {∃α, β. list α x * list β y * δ ≐ β† · α}
4 (x!=0) {
5   {∃α, β, i, Z. x ↦ i, Z * list α Z * list β y * δ ≐ β† · (i · α)}
6   z:=[x+1];
7   {∃α, β, i. x ↦ i, z * list α z * list β y * δ ≐ (i · β)† · α}
8   [x+1]:=y;
9   {∃α, β, i. x ↦ i, y * list α z * list β y * δ ≐ (i · β)† · α}
10  {∃α, β. list α z * list β x * δ ≐ β† · α}
11  y:=x; x:=z;
12  {∃α, β. list α x * list β y * δ ≐ β† · α}
13 }
14 {list δ† y}

```

Fig. 6. Proof outline for list reverse

exiting the loop. The loop invariant is the collection of ribbons entering the top of the loop: $ls \ x \ t$, $t \mapsto u$ and $ls \ u \ 0$. This collection must be recreated at the end of the loop body, so that one could pinch the proof into the shape drawn in Fig. 5.

Example 2: List reverse

Figure 6 presents a program for in-place reversal of a list. It gives a proof outline (adapted from [3]) comprising a precondition, postcondition and loop invariant, together with several intermediate assertions that guide the reader through the proof. We write \cdot for sequence concatenation, $(-)^{\dagger}$ for sequence reversal and ϵ for the empty sequence, and define $list$ as the smallest predicate satisfying

$$list \ \alpha \ x \Leftrightarrow x \doteq 0 * \alpha \doteq \epsilon \vee x \neq 0 * \exists \alpha', i, x'. x \mapsto i, x' * \alpha \doteq i \cdot \alpha' * list \ \alpha' \ x'.$$

Despite the abundance of assertions, the proof outline still obscures several features of the proof. It is, for instance, hard to deduce the effect of ‘ $z:=[x+1]$ ’, because there are multiple differences between its precondition and its postcondition: the logical variable Z has been instantiated to the program variable z , and ‘ $\beta^{\dagger} \cdot (i \cdot \alpha)$ ’ has become ‘ $(i \cdot \beta)^{\dagger} \cdot \alpha$ ’. Only the former results from the assignment; the latter step has been performed simultaneously so as to avoid writing out the whole assertion yet again. In ribbon proofs, simultaneous proof steps (on disjoint assertions) are unproblematic – even idiomatic.

Another inadequacy in the proof outline is that it obscures the usage of the logical variables α and β . For instance, the α on line 3 is not the same as the α on line 5, though visually it seems to be. In fact, the latter is witnessed as the *tail* of the former. The witness for β is constant through lines 5, 7 and 9, but on line 10 it becomes the previous β prepended with i . These subtle changes can only be spotted through careful examination of the proof outline.

The handling of logical variables in the ribbon proof shown in Fig. 7 is far more satisfactory. The scope of a logical variable is delimited by a thin *existential box*, in a manner similar to Fitch’s [6, Chap. 5]. Boxes extend horizontally across several ribbons, but also vertically to indicate the range

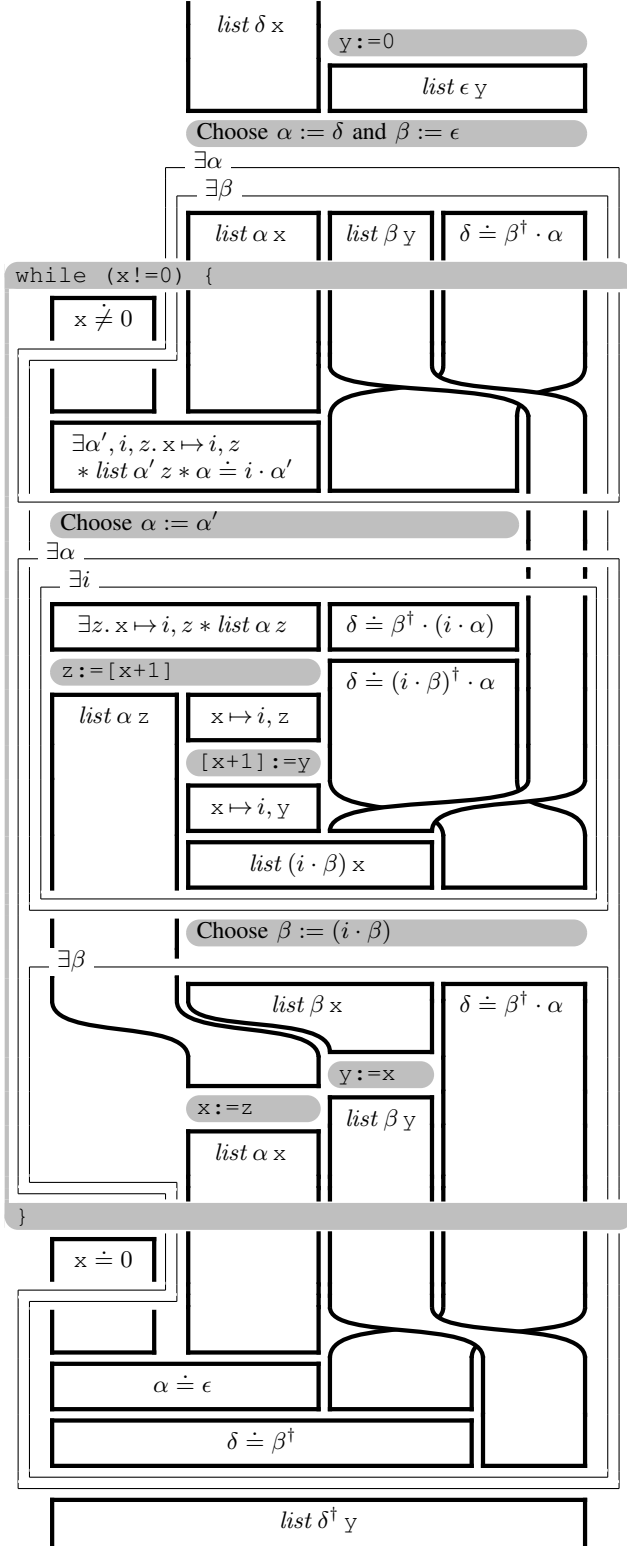


Fig. 7. Ribbon proof of list reverse. Note that most justifications of entailments are omitted to save space.

of steps over which the same witness is used. Within any single row projected from the proof, boxes, like existential quantifiers, must be well-nested. Vertically, however, boxes may overlap. For instance, in Fig. 7, α 's box contains β 's before the loop, but inside the loop, β 's box extends beyond the end of α 's. This reversal of the nesting order corresponds to the implication $\exists x. \exists y. p \Rightarrow \exists y. \exists x. p$. At the top of the loop body, we extend the scopes of α and β : this corresponds to the implication $p * \exists x. q \Rightarrow \exists x. p * q$ (where x is not in p).

A slight refinement to the previous subsection is due: a loop invariant is not just a collection of ribbons, but of boxes too. The one in Fig. 7 comprises three ribbons contained within two boxes. The loop invariant is reestablished at the end of the loop body, but note that the boxes are different: α 's box was replaced about one-third of the way through the loop body, and β 's box about two-thirds through. 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 syntactic structure of the program; instead, they appear to be *dynamically* scoped.

III. FORMALISATION

We now formalise the concepts introduced in the previous section. All of the proofs in this section and the next have been mechanically verified in Isabelle.¹

Definition 1 (Assertions). Let p range over the set of ordinary separation logic assertions, which we assume to contain at least the following constructions:

$$p \in \text{Assertion} ::= \text{emp} \mid p * p \mid \exists x. p \mid \dots$$

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

$$c \in \text{Command} ::= c \mid c \mid \text{skip} \mid c \text{ or } c \mid \text{loop } c \mid \dots$$

Since we are using partial correctness, standard if-statements and while-loops can be derived with the help of a primitive 'assume' command:

$$\begin{aligned} \text{if } b \ c_1 \ c_2 &\stackrel{\text{def}}{=} (\text{assume } b; c_1) \text{ or } (\text{assume } \neg b; c_2) \\ \text{while } b \ c &\stackrel{\text{def}}{=} \text{loop}(\text{assume } b; c); \text{assume } \neg b \end{aligned}$$

Definition 3 (Proof rules). We assume a separation logic over these commands and assertions, containing at least those rules given in Fig. 8. In that figure and elsewhere, we write rd/wr for the sets of program variables read/written by a command or an assertion.

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.

¹Proof script online at: <http://www.cl.cam.ac.uk/~jpw48/ribbons.thy.html>

$$\begin{array}{c}
\frac{}{\vdash_{\text{SL}} \{p\} \text{skip} \{p\}} \quad \frac{\vdash_{\text{SL}} \{p\} c_1 \{q\} \quad \vdash_{\text{SL}} \{q\} c_2 \{r\}}{\vdash_{\text{SL}} \{p\} c_1 ; c_2 \{r\}} \\
\frac{\vdash_{\text{SL}} \{p\} c \{q\} \quad wr(c) \cap rd(r) = \emptyset}{\vdash_{\text{SL}} \{p * r\} c \{p * r\}} \quad \frac{\vdash_{\text{SL}} \{p\} c \{q\}}{\vdash_{\text{SL}} \{\exists x. p\} c \{\exists x. q\}} \\
\frac{\vdash_{\text{SL}} \{p\} c_1 \{q\} \quad \vdash_{\text{SL}} \{p\} c_2 \{q\}}{\vdash_{\text{SL}} \{p\} c_1 \text{ or } c_2 \{q\}} \quad \frac{\vdash_{\text{SL}} \{p\} c \{p\}}{\vdash_{\text{SL}} \{p\} \text{loop} c \{p\}}
\end{array}$$

Fig. 8. Proof rules for commands

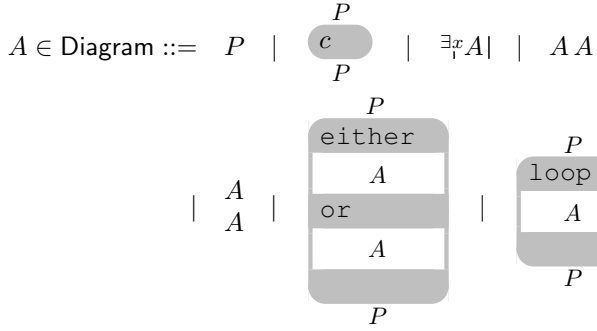


Fig. 9. Abstract syntax of diagrams

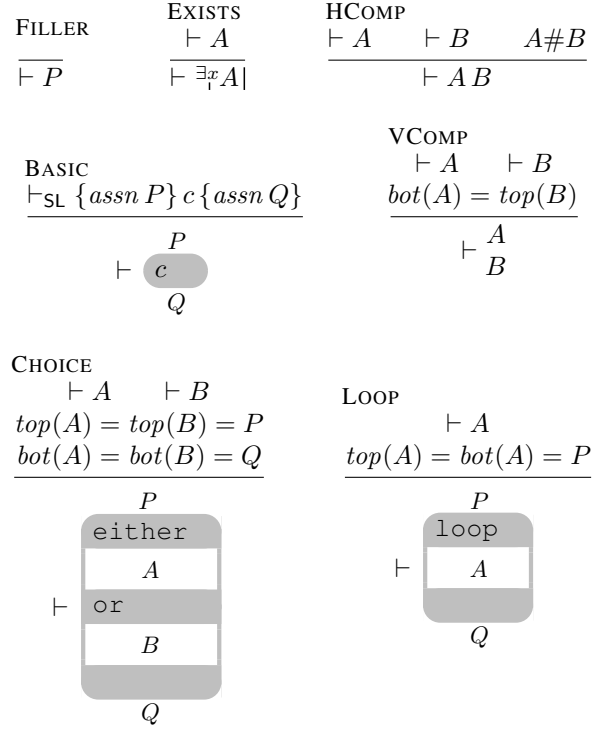


Fig. 10. Provability of diagrams

A. Abstract ribbon diagrams

In this subsection we define a syntax, provability relation, and semantics for an abstract version of our ribbon diagrams. Abstract diagrams contain neither size nor absolute positional information for their components, only their position relative to each other. The next subsections shall bridge the gap between these abstract diagrams and the concrete pictures we have already seen.

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

$$P \in \text{Interface} ::= \boxed{p} \mid \epsilon \mid PP \mid \exists x P.$$

Roughly, an interface is a single row of a ribbon diagram. We shall be particularly interested in those interfaces at the top and bottom of a diagram, as it is these which must match when diagrams are vertically composed. Interfaces contain no size information, so can be horizontally stretched as appropriate. Since the concrete pictures elide parentheses and show the empty interface as whitespace, we shall identify interfaces up to the equivalences $(PQ)R = P(QR)$ and $P\epsilon = \epsilon P = P$.

Definition 5 (Semantics of interfaces). There exists a straightforward mapping from interfaces to assertions, given by: $assn(\boxed{p}) = p$, $assn(\epsilon) = emp$, $assn(PQ) = assn(P) * assn(Q)$, and $assn(\exists x P) = \exists x. assn(P)$.

Let A range over the set of *diagrams* presented in Fig. 9. (We avoid the term ‘proof’ at this point since only *provable*

diagrams constitute a proof.) The first kind of diagram is a *filler*: an interface that can be vertically stretched as necessary. The second is a *basic step*: a proof that command c has the given pre- and postcondition. The third encloses a diagram in an *existential box*. The fourth and fifth are horizontal and vertical composition. The final two are *choice diagrams* and *loop diagrams*, from which diagrams that handle if-statements and while-loops can be derived by adding *assume* steps.

There are two pertinent questions to be asked of a given ribbon diagram. The first is: is it a valid proof? Figure 10 defines a *provability* predicate to answer this. We use top/bot to project the top/bottom interface from a diagram, and define $A \# B \stackrel{\text{def}}{=} rd(A) \cap wr(B) = rd(B) \cap wr(A) = \emptyset$. Note that HCOMP resembles separation logic’s rule for disjoint concurrency [3]. If B is a filler, we obtain a version of the frame rule. Likewise, VCOMP resembles the sequencing rule.

Checking provability is a predominantly visual procedure, easily performed by a human. The side-conditions on VCOMP, CHOICE and LOOP require various interfaces to ‘match up’; this is visually evident once the diagram is helpfully laid out. HCOMP’s side-condition forbids written variables to appear in adjacent diagrams. Checking the BASIC rule reduces to checking a separation logic judgement $\vdash_{\text{SL}} \{p\} c \{q\}$. This could be arbitrarily complex. In practice, p and q are usually small, since HCOMP allows irrelevant assertions to be framed away. Moreover, c is usually `skip` or some primitive command, since sequential composition, choices and loops can be handled using the diagrams in the second row of Fig. 9. Hence, the separation logic judgement is typically simple. (That said,

$$\begin{aligned}
\text{coms} &: \text{Diagram} \rightarrow \mathcal{P}(\text{Command}) \\
\text{coms}' &: \text{Diagram} \rightarrow \mathcal{P}(\text{Command list}) \\
\text{coms}(A) &= \{c_1; \dots; c_n; \text{skip} \mid [c_1, \dots, c_n] \in \text{coms}'(A)\} \\
\text{coms}'(P) &= \{\{\}\} \\
\text{coms}'\left(\begin{array}{c} P \\ \text{c} \\ Q \end{array}\right) &= \{[c]\} \\
\text{coms}'(\exists x A) &= \text{coms}'(A) \\
\text{coms}'(A B) &= \bigcup \{ \text{interleave}(C, C') \mid C \in \text{coms}'(A) \wedge \\
&\quad C' \in \text{coms}'(B) \} \\
\text{coms}'\left(\begin{array}{c} A \\ B \end{array}\right) &= \{C @ C' \mid C \in \text{coms}'(A) \wedge C' \in \text{coms}'(B)\} \\
\text{coms}'\left(\begin{array}{c} P \\ \text{either} \\ A \\ \text{or} \\ B \\ Q \end{array}\right) &= \{[c \text{ or } c'] \mid c \in \text{coms}(A) \wedge \\
&\quad c' \in \text{coms}(B)\} \\
\text{coms}'\left(\begin{array}{c} P \\ \text{loop} \\ A \\ Q \end{array}\right) &= \{[\text{loop } c] \mid c \in \text{coms}(A)\}
\end{aligned}$$

Fig. 11. Extracting commands from diagrams

since ribbon proofs are intended for humans, we do let basic steps contain compound commands. For instance, although ‘while true do skip’ could be proved using a full-blown loop diagram, one basic step may provide sufficient detail.)

The second question is: if this ribbon diagram *is* deemed valid, what does it prove? Since our diagrams can be composed ‘in parallel’, 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 in Fig. 11 is responsible for extracting this set from a given diagram. It first uses a helper function, *coms'*, to extract the set of lists of commands, and then it flattens each list. The *interleave* function returns all interleavings of a pair of lists.

Definition 6 (Semantics of diagrams). Diagram *A* denotes a set of Hoare triples, each with its precondition/postcondition as *A*’s top/bottom interface, and its command in *coms*(*A*):

$$\llbracket A \rrbracket \stackrel{\text{def}}{=} \{ \{ \text{assn}(\text{top } A) \} c \{ \text{assn}(\text{bot } A) \} \mid c \in \text{coms}(A) \}.$$

Theorem 1 (Soundness). *If there exists a provable ribbon diagram of a triple $\{p\} c \{q\}$, then that triple is provable in separation logic. That is:*

$$\forall p, c, q. (\exists A. \vdash A \wedge \{p\} c \{q\} \in \llbracket A \rrbracket) \Rightarrow \vdash_{\text{SL}} \{p\} c \{q\}.$$

Proof: See Appendix A. ■

Theorem 2 (Relative completeness). *Any separation logic proof can be recreated as a ribbon diagram. That is:*

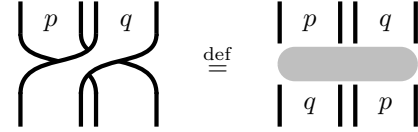
$$\forall p, c, q. \vdash_{\text{SL}} \{p\} c \{q\} \Rightarrow (\exists A. \vdash A \wedge \{p\} c \{q\} \in \llbracket A \rrbracket).$$

Proof: Choose *A* as the basic step comprising command *c*, top interface $\{p\}$ and bottom interface $\{q\}$. ■

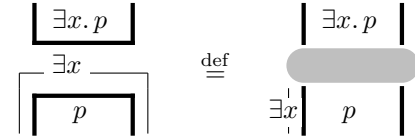
B. From abstract diagrams to concrete pictures

The following steps are taken to transform abstract diagrams into concrete pictures of the kind presented in Sect. II.

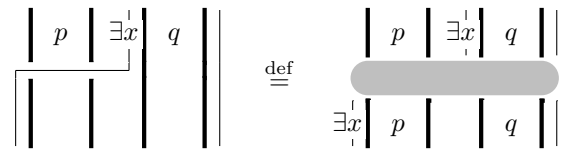
- Entailments are performed using basic steps whose command is *skip*. Rather than write ‘skip’, we label the step with a formal or informal justification of the entailment.
- Wherever a ribbon meets a basic step, we close the ribbon with a thick horizontal ‘lid’.
- The empty interface ϵ is displayed as whitespace.
- Where the VCOMP rule is employed, the ribbons and boxes being joined are shifted and stretched to make the diagram visually continuous. The labels on the top interface of the lower diagram become redundant, and are removed.
- The ‘twist’ operation is an abbreviation for the following entailment (the extension to extra ribbons is similar):



- The creation of an existential box is an entailment (deletion is similar):



- The extension of an existential box is an entailment, conditional on *x* not being free in *p* (contraction is similar):



C. From concrete pictures to abstract diagrams

A single concrete picture can be parsed as several different abstract diagrams. We now discuss whether every interpretation of a concrete picture is equally provable and semantically equivalent.

Definition 7 (Visual congruence). Visual congruence (written \sim) is the smallest congruence on diagrams satisfying the following properties. Firstly, diagrams that are identical up to associativity of horizontal and vertical composition are

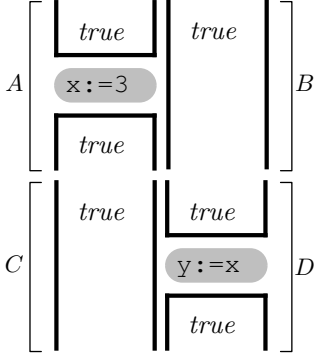


Fig. 12. Counterexample to the exchange law

visually congruent, because parentheses are not shown in the concrete pictures:

$$A(BC) \sim (AB)C \quad \begin{pmatrix} A \\ B \\ C \end{pmatrix} \sim \begin{pmatrix} A \\ B \\ C \end{pmatrix}$$

Secondly, a diagram is visually unchanged by adding a gap to its left or its right, because concrete pictures show gaps merely as whitespace:

$$A \epsilon \sim A \quad \epsilon A \sim A$$

Thirdly, a diagram can be extended upwards/downwards with a copy of its top/bottom interface without any visual effect, because concrete pictures obscure the ‘joins’ where the VCOMP rule has been applied:

$$\begin{matrix} A \\ \text{bot}(A) \end{matrix} \sim A \quad \begin{matrix} \text{top}(A) \\ A \end{matrix} \sim A$$

Theorem 3. *Visual congruence implies equiprovability and semantic equivalence:*

$$A \sim B \Rightarrow (\vdash A = \vdash B \wedge \llbracket A \rrbracket = \llbracket B \rrbracket)$$

Proof: By rule induction on \sim using the following strengthened induction hypothesis: $A \sim B \Rightarrow (\vdash A = \vdash B \wedge \text{top}(A) = \text{top}(B) \wedge \text{bot}(A) = \text{bot}(B) \wedge \text{rd}(A) = \text{rd}(B) \wedge \text{wr}(A) = \text{wr}(B) \wedge \text{coms}'(A) = \text{coms}'(B))$. ■

There is a fourth source of ambiguity in a concrete picture, which has been omitted from Defn. 7. A picture taking the visual form of a grid might be parsed by column, or by row, or by an alternation thereof. Unfortunately, it is not the case that each of these parsings is equiprovable and semantically equivalent. For example, let A , B , C and D be the four diagrams shown in Fig. 12. The term $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ is provable but $\begin{pmatrix} A \\ C \end{pmatrix} \begin{pmatrix} B \\ D \end{pmatrix}$ is not (the left-hand column writes what the right-hand one reads). Moreover, the semantics of $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ contains just $\{\text{true}\} x:=3; y:=x \{\text{true}\}$, but that of $\begin{pmatrix} A \\ C \end{pmatrix} \begin{pmatrix} B \\ D \end{pmatrix}$ contains additionally $\{\text{true}\} y:=x; x:=3 \{\text{true}\}$.

We show how to rectify this situation in the next section, but in the meantime, we adopt the following ‘row first’ or

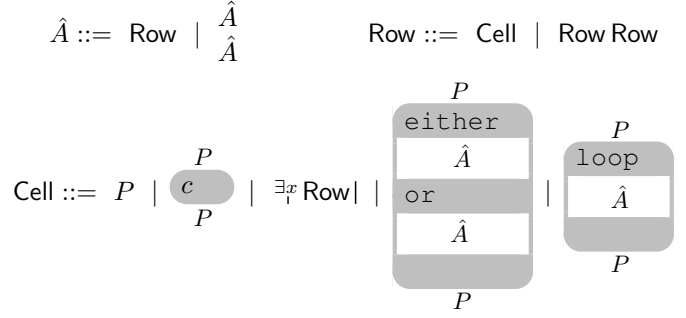


Fig. 13. Rasterised diagrams

‘rasterising’ strategy when parsing a concrete picture into an abstract diagram.

Definition 8 (‘Rasterising’ parsing strategy).

- 1) Trim the interfaces above and below each basic step, choice diagram and loop diagram to a small non-zero height. (The appearance of extra height can be achieved using filler.)
- 2) Parse the picture into a term \hat{A} of the fragment of *rasterised* diagrams presented in Fig. 13.

Essentially: if two diagrams are to be interpreted as horizontally composed, they must be drawn aligned; otherwise, they must be spaced out and will be interpreted as vertically composed. For example, Fig. 1b must be read as a vertical composition of three applications of the BASIC rule, each with two ribbons framed on, interspersed with rows of fillers to provide height. Without this parsing strategy, the picture might validly be interpreted as three assignment steps composed *horizontally*. Note that pictures containing partially vertically-overlapping components cannot be rasterised, but we dismiss such pictures as ‘badly drawn’.

IV. THE EXCHANGE LAW, VARIABLES-AS-RESOURCE AND PARALLELISATION

This section describes how the counterexample in Fig. 12 can be resolved more satisfactorily by shifting to the *variables-as-resource* paradigm [8]. We extend our ribbon proofs to use this paradigm, and suggest how they can be used to justify compiler optimisations such as parallelisation.

The two interpretations of Fig. 12 are not equiprovable because of the reading and writing of program variables. We can obtain an equiprovability result by imposing a condition on variable usage. We name this the ‘exchange law’ by analogy with the one-sided version studied in [9].

Theorem 4 (Exchange). *If $\text{bot}(A) = \text{top}(C)$, $\text{bot}(B) = \text{top}(D)$, $A \# D$ and $B \# C$ then*

$$\vdash \begin{pmatrix} A \\ C \end{pmatrix} \begin{pmatrix} B \\ D \end{pmatrix} = \vdash \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Note that the exchange law holds only for equiprovability: semantic equivalence does not hold because a grid parsed

by column may admit more command interleavings than one parsed by row.

The first two conditions of Thm. 4 hold whenever the four diagrams line up in a two-by-two grid. (The second is redundant, but included for symmetry.) To discharge the remaining conditions, we look to variables-as-resource. The variables-as-resource paradigm treats program variables much 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 $\pi \in (0..1]$) must be held to read it. This treatment supplants the use of rd and wr sets, which can henceforth be assumed to be empty, thus validating the third and fourth conditions of Thm. 4. Note that both interpretations of Fig. 12 are now equally unprovable: neither assignment 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 must simply choose appropriate separation logic axioms and primitive commands.

Figure 14 exhibits a ribbon proof of a tree-copying routine (adapted from [7]) conducted using variables-as-resource. If a binary tree is rooted at location p , $copy_tree(q;p)$ creates a copy at location q . We define the *tree* predicate as the smallest satisfying

$$\begin{aligned} tree \tau x \Leftrightarrow x \dot{=} 0 * \tau \dot{=} \bullet \vee \\ x \neq 0 * \exists v, i, j, \tau_1, \tau_2. x \mapsto v, i, j \\ * \tau \dot{=} \tau_1 \overset{v}{*} \tau_2 * tree \tau_1 i * tree \tau_2 j, \end{aligned}$$

where τ ranges over abstract binary trees, whose leaves are written ‘ \bullet ’ and whose other nodes store values.

Variables-as-resource dictates that each assertion is accompanied by one *Own* predicate per program variable mentioned. For instance, the function’s first precondition, $tree \tau p$, is paired with some fraction $\pi \in (0..1]$ of p ’s resource. That this fraction is non-zero allows p to be mentioned in the assertion, and later read by the function. The other precondition, the entirety of q ’s resource, entitles the function to write to q . The shading is merely syntactic sugar: $\lfloor q, \frac{1}{2}i', \frac{1}{2}j' \rfloor q \mapsto v, i', j' \rfloor$, for instance, means $\lfloor Own_1(q) * Own_{.5}(i') * Own_{.5}(j') * q \mapsto v, i', j' \rfloor$.

Figure 14 introduces a few novel features. ‘Jigsaw puzzle pieces’ help the reader connect the ribbons correctly at the boundaries of if and while blocks (in accordance with Figs. 4 and 5). Ribbons may pass ‘underneath’ basic steps to reduce the amount of twisting required (see e.g. ‘ $i := [p+1]$; $j := [p+2]$ ’). Horizontal space is conserved by writing some assertions sideways. We also introduce a new construction for block-scoped variables: at the top of the block we create a ribbon containing the entire resource for the declared variable, which is split in order to distribute access throughout the block, then reassembled and destroyed at the end of the block.

Thanks to Thm. 4, Fig. 14 can be parsed freely: we are unconstrained by the rasterisation strategy of Defn. 8, and can be confident that any parsing is equally provable. Rasterisation would place a vertical composition between the recursive calls

and ‘ $q := cons(3)$ ’. Suppose the sub-diagram containing the recursive calls were instead *horizontally* composed with that containing q ’s allocation: then any interleaving of their commands is admitted. We can therefore use the same picture to justify several variations of a program. For instance: a compiler optimisation that reorders ‘ $q := cons(3)$ ’ to the very start of the else-branch can be justified by stretching the right-hand column of the picture appropriately. The abstract diagram is unchanged by this purely visual transformation. Conversely, an (unsafe) optimisation that switches the recursive calls with the assignments to i and j cannot be justified by this picture because the commands involved are not horizontally separated.

We conclude this section with two ‘pieces of advice’.

- If seeking merely to present a proof of a particular program, one need not use variables-as-resource. The splitting, distributing, and re-combining of the resource associated with each variable is an unnecessary burden. (Figure 14 is several times larger than it would have been without variables-as-resource.) Concrete pictures should be drawn carefully so they can be successfully rasterised.
- If seeking to explore potential optimisations, or to analyse the dependencies between various components of a program, one should employ variables-as-resource. Concrete pictures should be parsed ‘column-first’, to enable as many visual manoeuvres as possible.

V. FUTURE DIRECTIONS

We have so far considered only sequential programs, even though the proofs themselves have a concurrent nature. It may be possible to extend our ribbon proof system to handle *concurrent separation logic* [10] as follows. Consider a program (adapted from [10]) in which two threads communicate through a shared single-cell buffer at location c :

```

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

```

<pre> while (true) { with buff when full { full:=false; y:=c; } dispose(y); } </pre>
--

The *resource invariant* protected by the lock `buff` is $(full \wedge c \mapsto _) \vee (\neg full \wedge emp)$, which means that the location c is shared exactly when the `full` flag is set.

Figure 15 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, that ribbon is made available to the thread. The resource invariant is re-established at the end of the critical region, at which point it becomes protected and inaccessible once more.

Beyond concurrent separation logic, it is hoped that our proof system can be applied fruitfully to more advanced separation logics for concurrency, such as RGSep [11], SAGL [12], Local Rely-Guarantee [13], Deny-Guarantee [14] and Concurrent Abstract Predicates [15]. Increasingly complicated logics make techniques for intuitive construction and clear presentation ever more crucial.

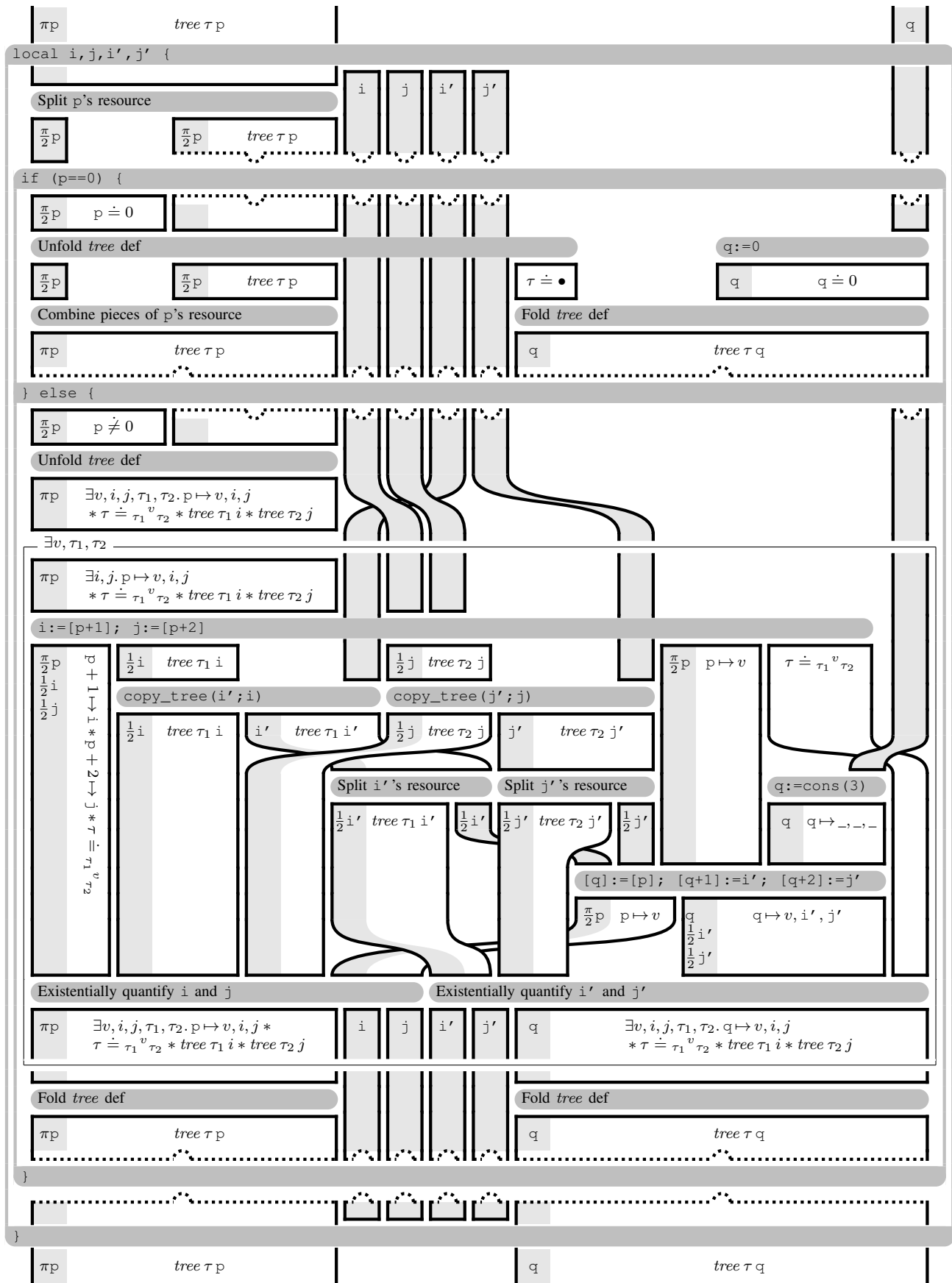


Fig. 14. Ribbon proof of `copy_tree`, using variables-as-resource and jigsaw puzzle pieces.

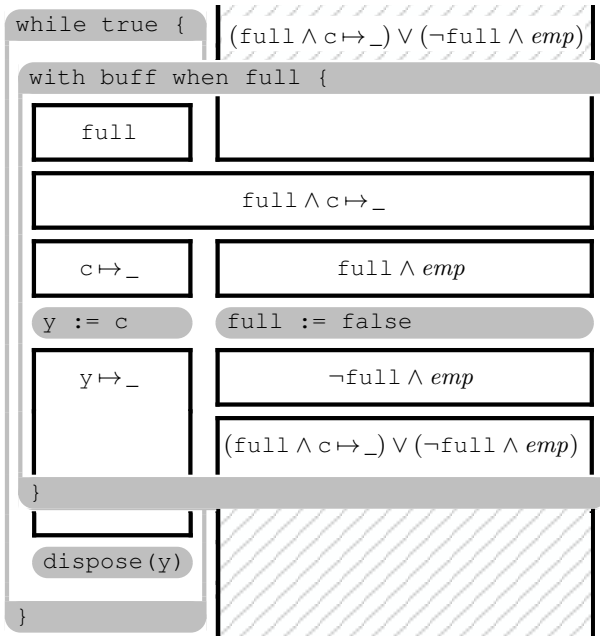


Fig. 15. Ribbon proof of single-cell buffer.

Another avenue for future work is to investigate the connection between our ribbon proofs and Raza’s *labelled separation logic* [16]. 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 are currently working on an improved semantics that models a ribbon diagram as a hierarchical directed acyclic graph. This semantics would be ignorant of the various twisting operations occurring in a concrete picture that we currently treat as syntactic sugar, and would clarify the connection between ribbon proofs and dependency graphs. Moreover, it would support further investigation of the seemingly ‘dynamic’ scoping of logical variables that was exhibited in Fig. 7. Each basic step would correspond to a vertex, each ribbon an edge, and the components of choice and loop diagrams would form subgraphs. The scoping of logical variables would be modelled by a separate forest structure over the edges, in a manner recalling Milner’s bigraphs [17]. Such a graphical semantics recalls Girard’s *proof nets* for linear logic [18], which, like our own system, eliminate visual redundancy from proofs.

The ribbon proofs in this paper have all been produced manually (with the help of several \LaTeX macros), but there is clear scope for tool support here, both to help to find a

layout that minimises twists and to validate each proof step. In the future, 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 logical rule and axiom, or a coarse birds-eye view of the overall structure of the proof.

VI. CONCLUSION

Ribbon proofs are an attractive and practical approach for constructing and presenting proofs in separation logic or any derivative thereof. They are free from redundancy, and express the intent of the proof more clearly than a proof outline. They can be checked locally, by focusing on each step of the proof in turn. They are useful pedagogically for explaining how a simple proof is constructed, but also scale to large and complex programs (as demonstrated in Appx. B). They show graphically the distribution of resource in a program, and in particular, which parts of a program operate on disjoint resources. This, with the addition of variables-as-resource, may prove useful for exploring parallelisation opportunities.

REFERENCES

- [1] C. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, October 1969.
- [2] S. Ishtiaq and P. W. O’Hearn, “BI as an assertion language for mutable data structures,” in *POPL*, 2001.
- [3] J. C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *LICS*, 2002.
- [4] J. Bean, “Ribbon proofs - a proof system for the logic of bunched implications,” Ph.D. dissertation, Queen Mary University of London, 2006.
- [5] P. W. O’Hearn and D. J. Pym, “The logic of bunched implications,” *Bulletin of Symbolic Logic*, 1999.
- [6] F. Fitch, *Symbolic logic: an introduction*. Ronald Press Co., 1952.
- [7] J. Berdine, C. Calcagno, and P. W. O’Hearn, “Symbolic execution with separation logic,” in *APLAS*, 2005.
- [8] M. J. Parkinson, R. Bornat, and C. Calcagno, “Variables as resource in Hoare logics,” in *LICS*, 2006.
- [9] C. Hoare, A. Hussain, B. Möller, P. W. O’Hearn, R. L. Petersen, and G. Struth, “On locality and the exchange law for concurrent processes,” in *CONCUR*, 2011.
- [10] P. W. O’Hearn, *Resources, Concurrency and Local Reasoning*, Queen Mary, University of London, 2007.
- [11] V. Vafeiadis and M. J. Parkinson, “A marriage of rely/guarantee and separation logic,” *CONCUR*, 2007.
- [12] X. Feng, R. Ferreira, and Z. Shao, “On the relationship between concurrent separation logic and assume-guarantee reasoning,” in *ESOP*, 2007.
- [13] X. Feng, “Local rely-guarantee reasoning,” Toyota Technological Institute at Chicago, Tech. Rep., 2008.
- [14] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis, “Deny-guarantee reasoning,” in *ESOP*, 2009.
- [15] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, “Concurrent abstract predicates,” in *ECOOP*, 2010.
- [16] M. Raza, “Resource reasoning and labelled separation logic,” Ph.D. dissertation, Imperial College, 2010.
- [17] R. Milner, *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
- [18] J.-Y. Girard, “Linear logic,” *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–102, 1987.

This proof has been formalised in Isabelle.²

It is difficult to prove this theorem directly by rule induction on \vdash , because from a proof of $\{p_0\} c; c' \{p_2\}$, it is possible to deduce the existence of an intermediate assertion p_1 , but not that p_1 reads only program variables already mentioned. This is necessary to complete the HCOMP case. To address this, we introduce the following generalisation of a Hoare triple.

Definition 9 (Hoare chains). A *Hoare chain* is an alternating sequence of assertions (delimited by angled brackets to avoid a clash with the notation for sets) and commands, beginning and ending with an assertion:

$$H \in \text{HoareChain} ::= \langle p \rangle \\ | \langle p \rangle c H.$$

Definition 10 (Provability of Hoare chains). For a Hoare chain $H = \langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle$, define:

$$\vdash_{\text{SL}} H \stackrel{\text{def}}{=} \forall i \in (0..n). \vdash_{\text{SL}} \{p_{i-1}\} c_i \{p_i\}.$$

Definition 11. For $H = \langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle$, define:

$$\text{comlist}(H) \stackrel{\text{def}}{=} [c_1, \dots, c_n].$$

Definition 12. The *chains* function, defined recursively in Fig. 16, extracts all the Hoare chains from a given diagram. It operates similarly to the *coms* and *coms'* functions of Fig. 11.

Lemma 13. Given a command list C extracted from a provable diagram, we can extract a Hoare chain from the same diagram whose command list is C . That is,

$$\forall A. \vdash A \Rightarrow \\ (\forall C \in \text{coms}'(A). \exists H \in \text{chains}(A). C = \text{comlist}(H)).$$

Proof: By induction on the structure of diagrams. ■

Lemma 14. If every Hoare chain extracted from a provable diagram A is provable, then every Hoare triple in $\llbracket A \rrbracket$ is provable. That is,

$$\forall A. \vdash A \wedge (\forall H \in \text{chains}(A). \vdash_{\text{SL}} H) \Rightarrow \\ (\forall p, c, q. \{p\} c \{q\} \in \llbracket A \rrbracket \Rightarrow \vdash_{\text{SL}} \{p\} c \{q\}).$$

Proof: If $\{p\} c \{q\}$ is in $\llbracket A \rrbracket$ then c is in $\text{coms}(A)$, $p = \text{assn}(\text{top } A)$ and $q = \text{assn}(\text{bot } A)$. Hence obtain a command list $[c_1, \dots, c_n]$ in $\text{coms}'(A)$ such that $c = c_1; \dots; c_n; \text{skip}$. Through Lem. 13, obtain a Hoare chain $H = \langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle$ in $\text{chains}(A)$ such that $p_0 = p$ and $p_n = q$. Use the second assumption to deduce $\vdash_{\text{SL}} H$. Erasing H 's intermediate assertions leads to $\vdash_{\text{SL}} \{p\} c \{q\}$. ■

Lemma 15. Every Hoare chain extracted from a provable diagram is provable. That is,

$$\forall A. \vdash A \Rightarrow (\forall H \in \text{chains}(A). \vdash_{\text{SL}} H).$$

Proof: By rule induction on \vdash . ■

The proof of Thm. 1 follows directly from Lems. 14 and 15.

²Proof script online at: <http://www.cl.cam.ac.uk/~jpw48/ribbons.thy.html>

We illustrate the ability of our system to produce readable proofs for large and 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. 17.

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. 18: 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 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:

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

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:

$$s, h \models \boxed{\frac{e_1}{e_2}} \stackrel{\text{def}}{=} \text{let } l_1 = \llbracket e_1 \rrbracket(s) \text{ and } l_2 = \llbracket e_2 \rrbracket(s) \\ \text{in } l_1 \in \mathbb{N}^+ \wedge h = \{l_1 \mapsto l_2\}.$$

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 $\boxed{\frac{e_1}{e_2}}$. 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

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

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

$$\begin{aligned}
\text{chains} & : \text{Diagram} \rightarrow \mathcal{P}(\text{HoareChain}) \\
\text{chains}(P) & = \{\langle \text{assn } P \rangle\} \\
\text{chains} \left(\begin{array}{c} P \\ \text{c} \\ Q \end{array} \right) & = \{\langle \text{assn } P \rangle \text{c} \langle \text{assn } Q \rangle\} \\
\text{chains}(\exists x. A) & = \{\langle \exists x. p_0 \rangle c_1 \langle \exists x. p_1 \rangle \dots c_n \langle \exists x. p_n \rangle \mid \langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle \in \text{chains}(A)\} \\
\text{chains}(A \ B) & = \bigcup \{\text{interleavechains}(H, H') \mid H \in \text{chains}(A) \wedge H' \in \text{chains}(B)\} \\
\text{chains} \left(\begin{array}{c} A \\ B \end{array} \right) & = \{\langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle c'_1 \langle q_1 \rangle \dots c'_m \langle q_m \rangle \\ & \quad \mid \langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle \in \text{chains}(A) \wedge \langle p_n \rangle c'_1 \langle q_1 \rangle \dots c'_m \langle q_m \rangle \in \text{chains}(B)\} \\
\text{chains} \left(\begin{array}{c} P \\ \text{either} \\ A \\ \text{or} \\ B \\ Q \end{array} \right) & = \{\langle \text{assn } P \rangle (c_1; \dots; c_n; \text{skip}) \text{ or } (c'_1; \dots; c'_m; \text{skip}) \langle \text{assn } Q \rangle \\ & \quad \mid \langle \text{assn } P \rangle c_1 \langle p_1 \rangle \dots c_n \langle \text{assn } Q \rangle \in \text{chains}(A) \wedge \\ & \quad \langle \text{assn } P \rangle c'_1 \langle q_1 \rangle \dots c'_m \langle \text{assn } Q \rangle \in \text{chains}(B)\} \\
\text{chains} \left(\begin{array}{c} P \\ \text{loop} \\ A \\ Q \end{array} \right) & = \{\langle \text{assn } P \rangle \text{loop} (c_1; \dots; c_n; \text{skip}) \langle \text{assn } Q \rangle \\ & \quad \mid \langle \text{assn } P \rangle c_1 \langle p_1 \rangle \dots c_n \langle \text{assn } Q \rangle \in \text{chains}(A) \wedge \\ & \quad \text{assn } Q = \text{assn } P\}
\end{aligned}$$

where:

$$\begin{aligned}
\text{interleavechains} & : \text{HoareChain} \times \text{HoareChain} \rightarrow \mathcal{P}(\text{HoareChain}) \\
\text{interleavechains}(\langle p \rangle, \langle q_0 \rangle c_1 \langle q_1 \rangle \dots c_n \langle q_n \rangle) & = \{\langle p * q_0 \rangle c_1 \langle p * q_1 \rangle \dots c_n \langle p * q_n \rangle\} \\
\text{interleavechains}(\langle p_0 \rangle c_1 \langle p_1 \rangle \dots c_n \langle p_n \rangle, \langle q \rangle) & = \{\langle p_0 * q \rangle c_1 \langle p_1 * q \rangle \dots c_n \langle p_n * q \rangle\} \\
\text{interleavechains}(\langle p \rangle \text{c } H, \langle q \rangle \text{c}' H') & = \{\langle p * q \rangle \text{c } H'' \mid H'' \in \text{interleavechains}(H, \langle q \rangle \text{c}' H')\} \cup \\ & \quad \{\langle p * q \rangle \text{c}' H'' \mid H'' \in \text{interleavechains}(\langle p \rangle \text{c } H, H')\}
\end{aligned}$$

Fig. 16. Extracting Hoare chains from diagrams

desirable. We use only what Reynolds calls ‘regular’ ranges, whose upper bound is greater than or equal to its lower bound:

$$s, h \models \boxed{e_1}^{e_2} \stackrel{\text{def}}{=} \text{let } l_1 = \llbracket e_1 \rrbracket(s) \text{ and } l_2 = \llbracket e_2 \rrbracket(s) \\ \text{in } l_1, l_2 \in \mathbb{N}^+ \wedge l_1 \leq l_2 \wedge \\ \text{dom}(h) = \{i \in \mathbb{N}^+ \mid l_1 \leq i < l_2\}.$$

The default diagram has an inclusive lower bound and an exclusive upper bound, but alternatives can be obtained by switching between $\boxed{e_1}^{e_2}$ and $\boxed{e_1}^{e_2+1}$. Two ranges may be concatenated; that is, $\boxed{e_1}^{e_2} * \boxed{e_2}^{e_3}$ implies $\boxed{e_1}^{e_3}$. We can write \boxed{e}^e as \boxed{e} . As an example: an unallocated chunk in the arena is written $\boxed{\frac{x}{y}}$. This can be rewritten in terms of the primitives defined above as $\boxed{\frac{x}{y}} * \boxed{x+1}^y$. 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. 18. 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 τ ; 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 ν), 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

```

#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);
}

```

Fig. 17. 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.

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 $A \subseteq (C_1 \circ C_2)^a$ 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. 18 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,⁶ 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 two further conventions adopted in this proof. Firstly, 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. Secondly, 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.

⁵M. J. Parkinson and G. M. Bierman, “Separation logic and abstraction,” in *POPL*, 2005

⁶J. Wickerson, M. Dodds, and M. J. Parkinson, “Explicit stabilisation for modular rely-guarantee reasoning,” in *ESOP*, 2010

Specifications of main routines

$$\left\{ \begin{array}{l} \text{uninit } s \ A \vee \\ \text{arena } s \ v \ t \ A \end{array} \right\} \text{malloc}(n) \left\{ \begin{array}{l} (\text{arena } s \ v \ t \ (A \uplus \{\text{ret} \mapsto \lceil n/\text{WORD} \rceil\})) * \boxed{\text{ret}}_{\text{ret} + \lceil n/\text{WORD} \rceil} \vee \\ (\text{arena } s \ v \ t \ A * \text{ret} \doteq 0) \end{array} \right\} \\ \left\{ \text{arena } s \ v \ t \ (A \uplus \{x \mapsto n\}) * \boxed{x}_{x+n} \right\} \text{free}(x) \left\{ \text{arena } s \ v \ t \ A \right\}$$

Specifications of sub-routines

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

Types

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

Operators

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

Predicates

$$\begin{array}{ll} \text{chunk}_u(x : \text{ptr})(y : \text{ptr}) & \stackrel{\text{def}}{=} \boxed{\begin{array}{l} x \\ y \end{array}}_y \\ \text{chunk}_a(x : \text{ptr})(y : \text{ptr}) & \stackrel{\text{def}}{=} x < y * \boxed{\begin{array}{l} x \\ y_{\blacksquare} \end{array}} \\ \text{chunks}(x : \text{ptr})(y : \text{ptr})(C : \text{chunks_int}) & \stackrel{\text{def}}{=} (x \doteq y * C \doteq []) \vee \exists z : \text{ptr}. \exists \tau : \text{tag}. \\ & \quad \text{chunk}_\tau x z * \text{chunks } z y (C \circ - [\langle x, \tau, z \rangle]) \\ \text{uninit}(s : \text{ptr})(A : \text{chunks_ext}) & \stackrel{\text{def}}{=} \boxed{\begin{array}{l} s \\ 0 \end{array}} * A \doteq \emptyset * \text{brka}(s+2) \\ \text{arena}(s : \text{ptr})(v : \text{ptr})(t : \text{ptr})(A : \text{chunks_ext}) & \stackrel{\text{def}}{=} \exists C_1, C_2 : \text{chunks_int}. \text{chunks } s \ v \ C_1 * \text{chunks } v \ t \ C_2 \\ & \quad * A \subseteq (C_1 \circ C_2)^a * \boxed{\begin{array}{l} t \\ s_{\blacksquare} \end{array}} * \text{brka}(t+1) \end{array}$$

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

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

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

Lemma D. $\text{arena } s \ v \ t \ A \implies \exists n > 0. \boxed{\begin{array}{l} s \\ n \end{array}} * (\boxed{\begin{array}{l} s \\ n \end{array}} \text{ } * \text{arena } s \ v \ t \ A)$

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

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

Fig. 18. Glossary of definitions and lemmas used in the specifications and proofs of malloc and free

$uninit\ s\ A \vee arena\ s\ v\ t\ A$

`char *malloc(unsigned int nbytes) {`

Case split, first disjunct

$uninit\ s\ A$

Unfold $uninit$

$\frac{s+1}{0} * A \doteq \emptyset * brka(s+2)$

$\exists n$

$\frac{s}{n}$

$n \doteq 0 * \frac{s+1}{0} * A \doteq \emptyset * brka(s+2)$

Weaken

$(n \doteq 0 * \frac{s+1}{0} * A \doteq \emptyset * brka(s+2)) \vee (n > 0 * (\frac{s}{n} \dashv arena\ s\ v\ t\ A))$

Case split, second disjunct

$arena\ s\ v\ t\ A$

$arena\ s\ v\ t\ A$ implies $\exists n > 0. \frac{s}{n} * (\frac{s}{n} \dashv arena\ s\ v\ t\ A)$ (Lem. D)

$\exists n$

$\frac{s}{n}$

$n > 0 * (\frac{s}{n} \dashv arena\ s\ v\ t\ A)$

Weaken

$(n \doteq 0 * \frac{s+1}{0} * A \doteq \emptyset * brka(s+2)) \vee (n > 0 * (\frac{s}{n} \dashv arena\ s\ v\ t\ A))$

`if(s[0].ptr == 0) {`

"

$n \doteq 0 * \frac{s+1}{0} * A \doteq \emptyset * brka(s+2)$

Split

$n \doteq 0$

$A \doteq \emptyset$

$\frac{s+1}{0}$

$brka(s+2)$

Combine

$\frac{s}{0}$

`s[0].ptr = setbusy(&s[1])`

$\frac{s}{s+1}$

`s[1].ptr = setbusy(&s[0])`

$\frac{s+1}{s}$

`t = &s[1]`

$s < t * \frac{s}{t}$

$\frac{t}{s}$

$brka(t+1)$

`v = &s[0]`

$v \doteq s$

$v < t * \frac{v}{t}$

Fold $chunk_a$

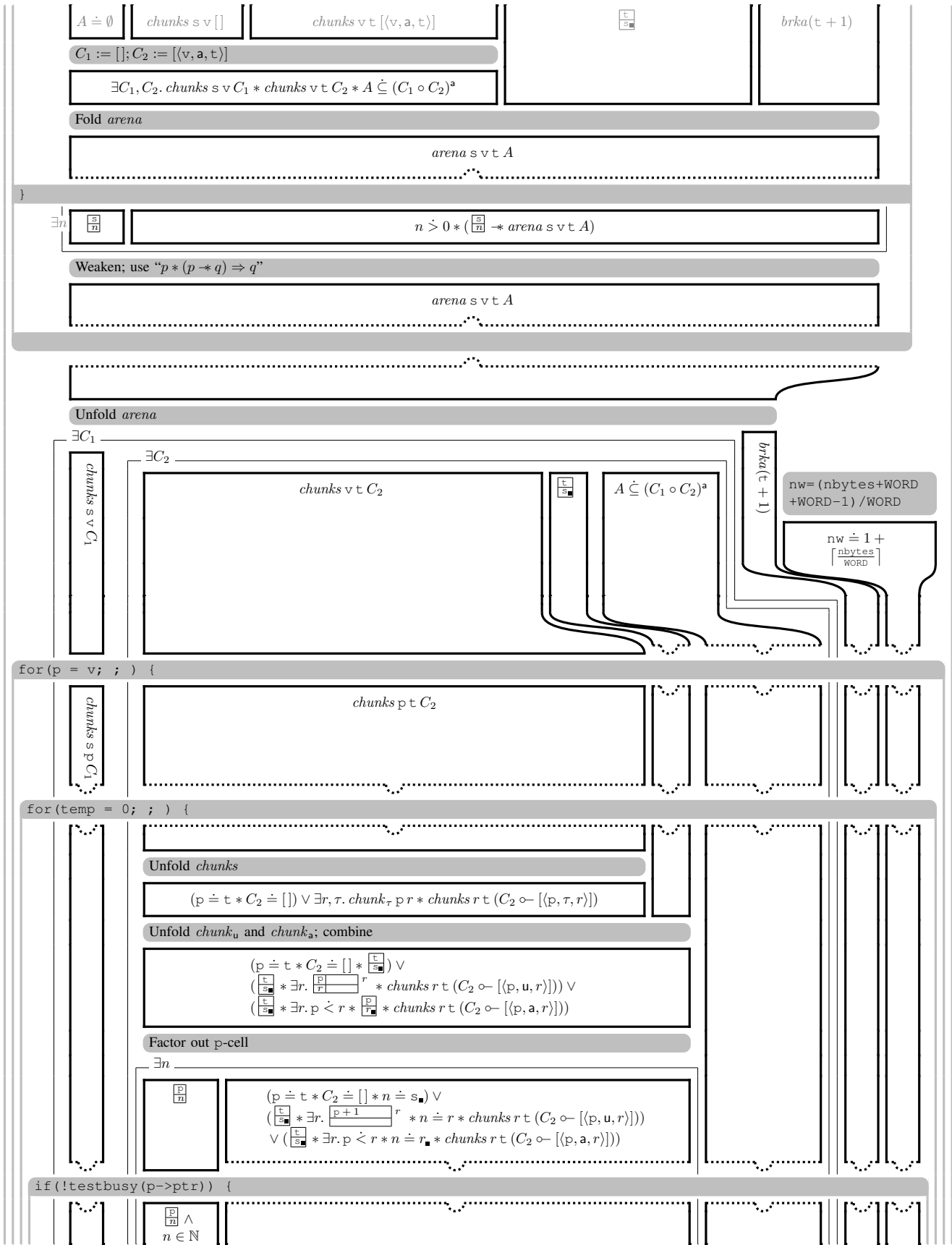
$chunk_a\ v\ t$

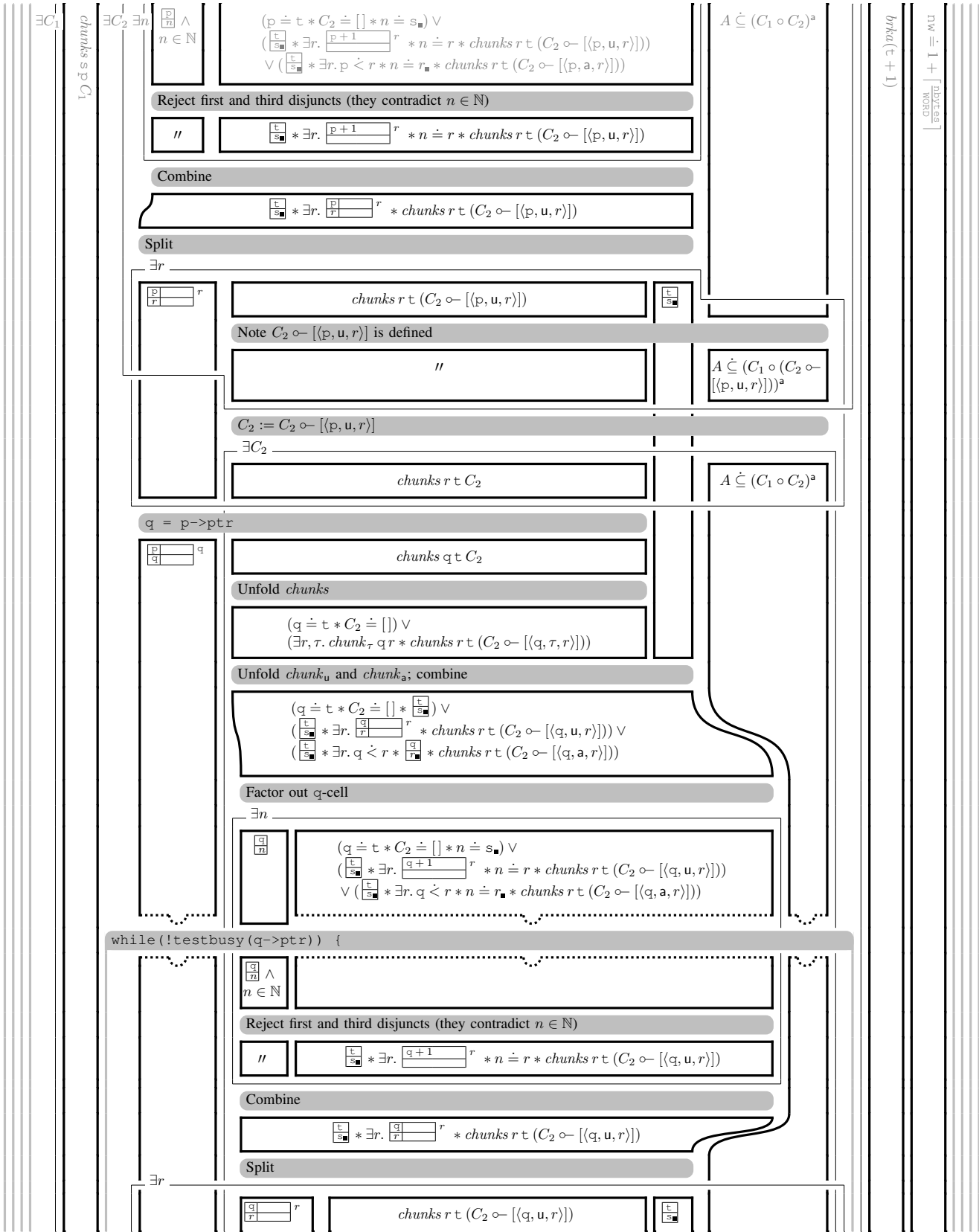
Fold $chunks$

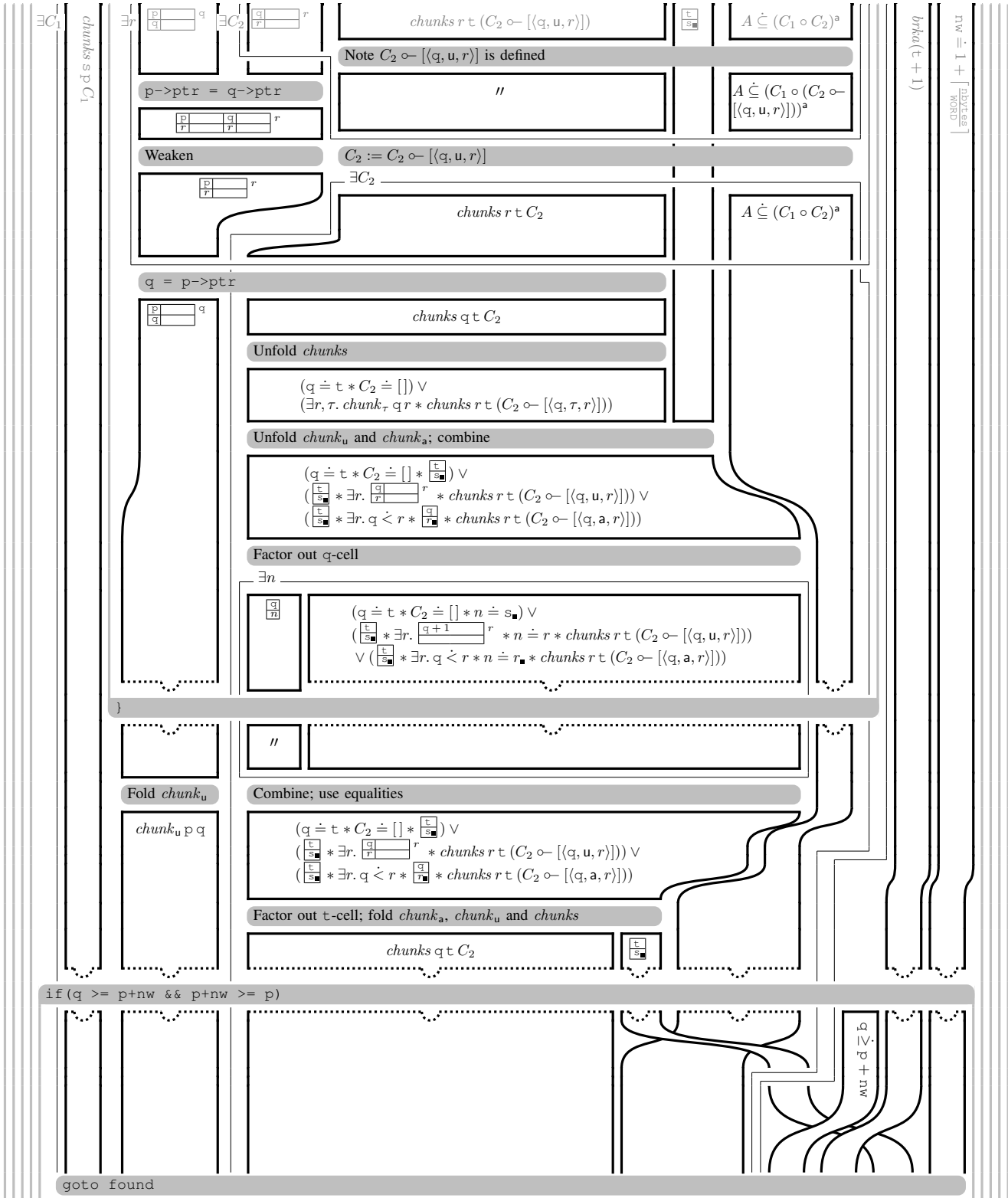
Fold $chunks$ (Lem. B)

$chunks\ s\ v\ []$

$chunks\ v\ t\ [(v, a, t)]$



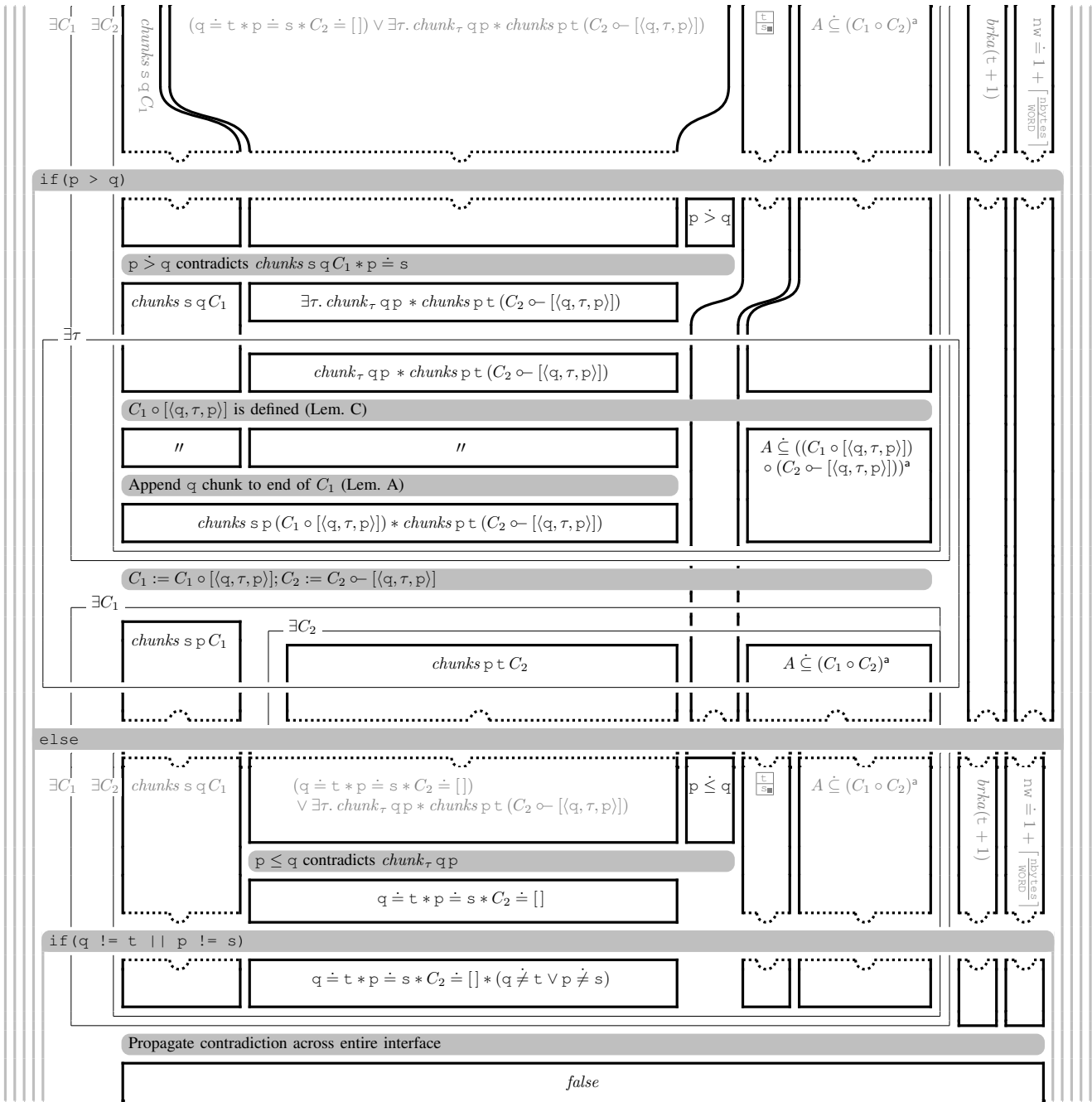




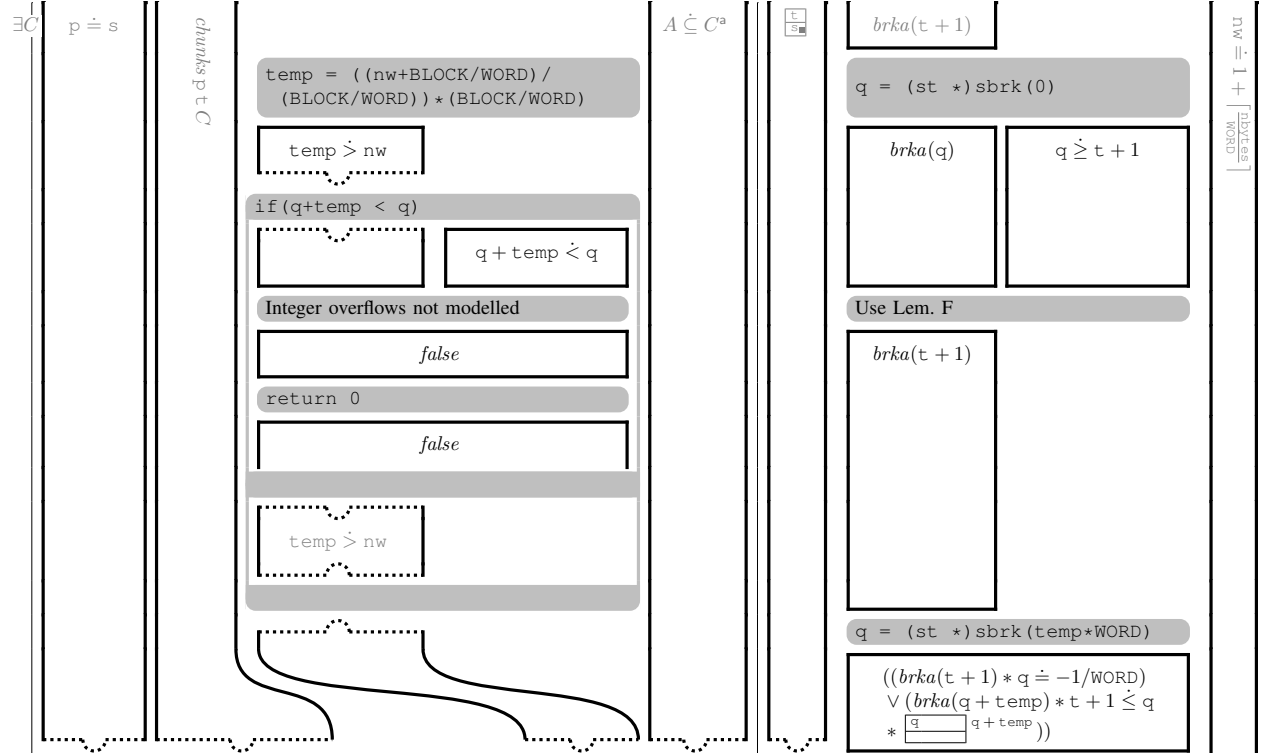
$\exists C_1$	$chunks\ s\ p\ C_1$	$chunk_u\ p\ q$	$\exists C_2$	$chunks\ q\ t\ C_2$	$\frac{t}{s}$	$A \dot{\subseteq} (C_1 \circ C_2)^a$	$brka(t+1)$	$nw = 1 + \lceil \text{bytes} \rceil_{\text{word}}$
$C_1 \circ [(p, u, q)] \circ C_2$ is defined (Lem. C)								
"		"		"		$A \dot{\subseteq} (C_1 \circ [(p, u, q)] \circ C_2)^a$		
Concatenate chunks (Lem. A)								
$chunks\ p\ t\ (([p, u, q]) \circ C_2)$								
$C_2 := [(p, u, q)] \circ C_2$								
$\exists C_2$								
$chunks\ p\ t\ C_2$								
$A \dot{\subseteq} (C_1 \circ C_2)^a$								

$\exists C_1$	$chunks\ s\ p\ C_1$	$\frac{p}{n}$	$\exists n$	$(p \dot{=} t * C_2 \dot{=} [] * n \dot{=} s_a) \vee$ $(\frac{t}{s} * \exists r. \frac{p+1}{r} * n \dot{=} r * chunks\ r\ t\ (C_2 \circ - [(p, u, r)]))$ $\vee (\frac{t}{s} * \exists r. p < r * n \dot{=} r * chunks\ r\ t\ (C_2 \circ - [(p, a, r)]))$	$\frac{t}{s}$	$A \dot{\subseteq} (C_1 \circ C_2)^a$	$brka(t+1)$	$nw = 1 + \lceil \text{bytes} \rceil_{\text{word}}$
Combine; use equalities								
$(p \dot{=} t * C_2 \dot{=} [] * \frac{t}{s}) \vee (\frac{t}{s} * \exists r. p < r * \frac{p}{r} * chunks\ r\ t\ (C_2 \circ - [(p, a, r)]))$								
Factor out t-cell; fold $chunk_a$ and $chunks$								
$chunks\ p\ t\ C_2$								

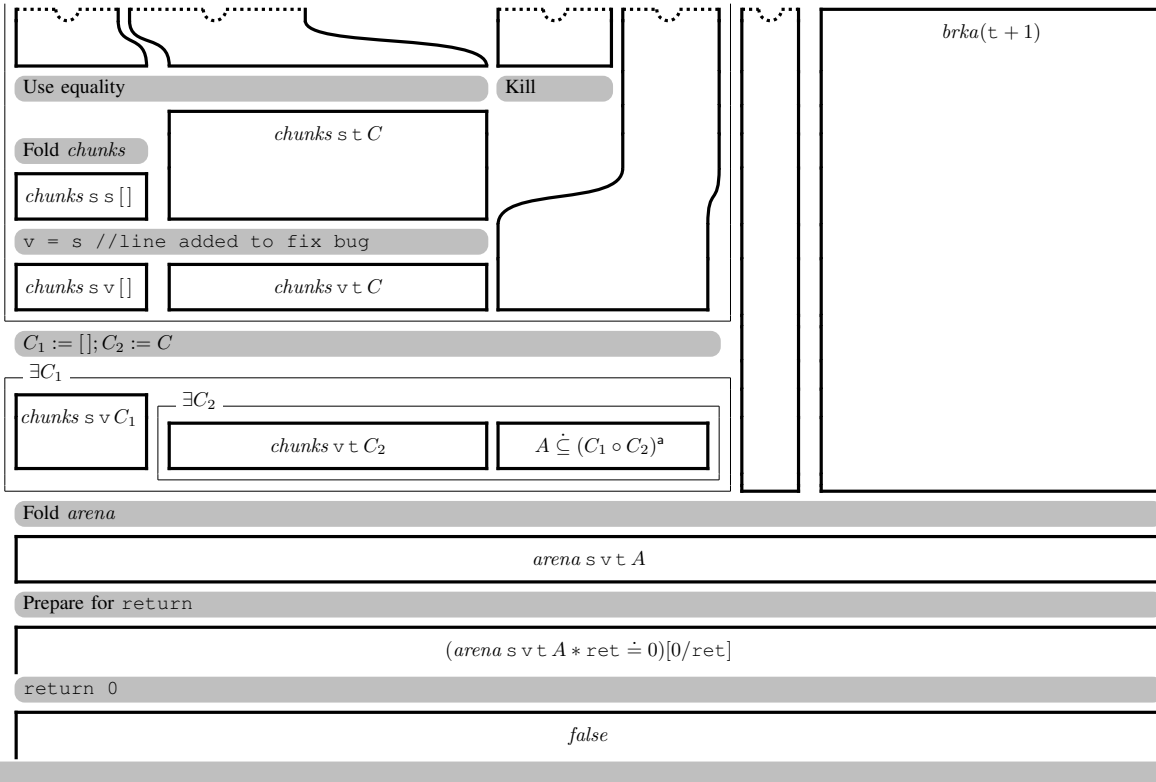
	$chunks\ s\ q\ C_1$	$q = p$	$chunks\ q\ t\ C_2$	$\frac{t}{s}$	$q \dot{=} p$			
Unfold $chunks$								
$(q \dot{=} t * C_2 \dot{=} []) \vee (\exists r, \tau. chunk_\tau\ q\ r * chunks\ r\ t\ (C_2 \circ - [(q, \tau, r)]))$								
Unfold $chunk_u$ and $chunk_a$; combine with t-cell; use equalities								
$(q \dot{=} t * C_2 \dot{=} [] * \frac{p}{s}) \vee$ $(\frac{t}{s} * \exists r. \frac{q}{r} * r * chunks\ r\ t\ (C_2 \circ - [(q, u, r)])) \vee$ $(\frac{t}{s} * \exists r. q < r * \frac{q}{r} * chunks\ r\ t\ (C_2 \circ - [(q, a, r)]))$								
$p = \text{clearbusy}(p \rightarrow ptr)$								
$(q \dot{=} t * p \dot{=} s * C_2 \dot{=} [] * \frac{t}{s}) \vee$ $(\frac{t}{s} * \frac{q}{p} * p * chunks\ p\ t\ (C_2 \circ - [(q, u, p)])) \vee$ $(\frac{t}{s} * q < p * \frac{q}{p} * chunks\ p\ t\ (C_2 \circ - [(q, a, p)]))$								
Factor out t-cell; fold $chunk_u$ and $chunk_a$								
$(q \dot{=} t * p \dot{=} s * C_2 \dot{=} []) \vee \exists r. chunk_\tau\ q\ p * chunks\ p\ t\ (C_2 \circ - [(q, \tau, p)])$								

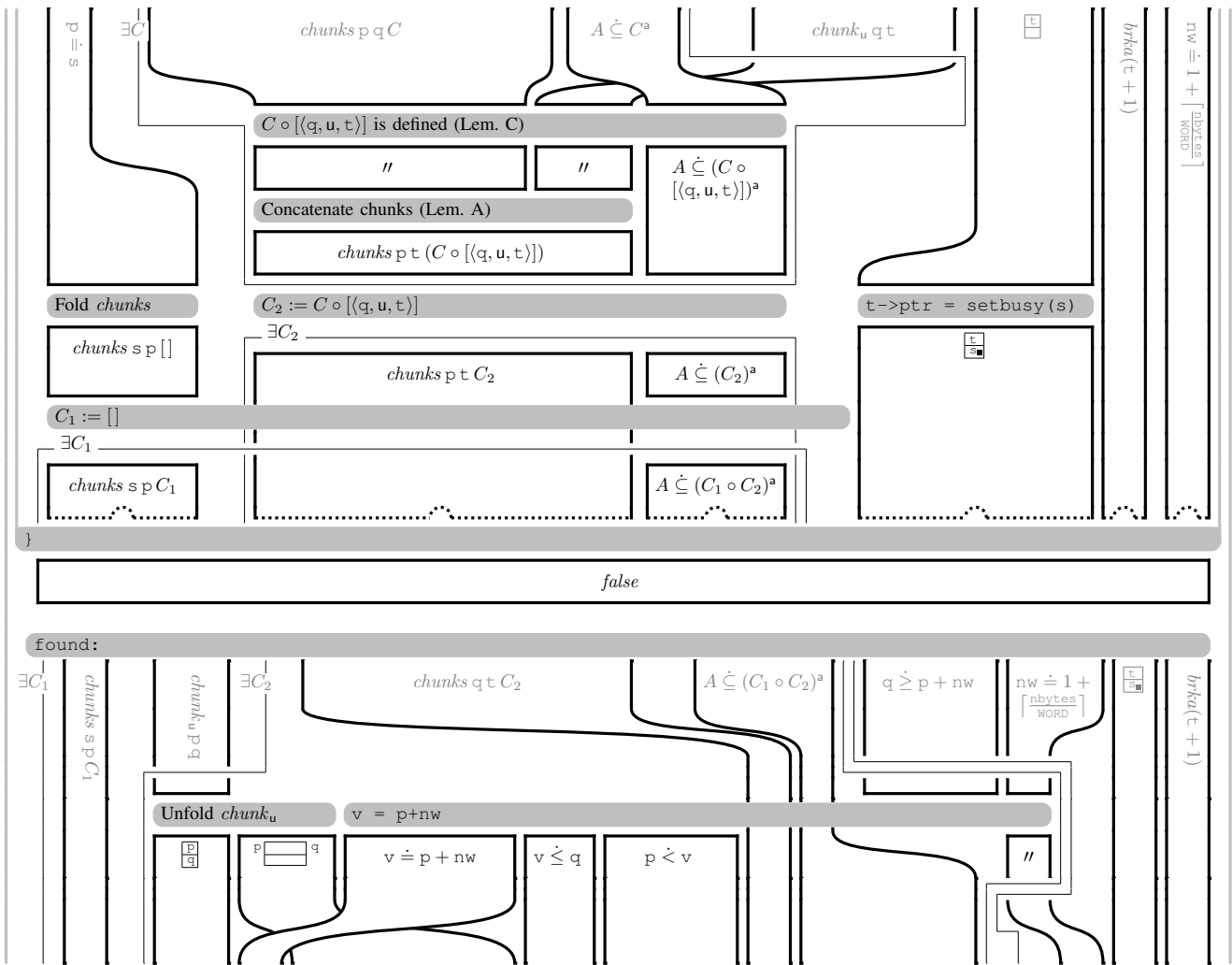


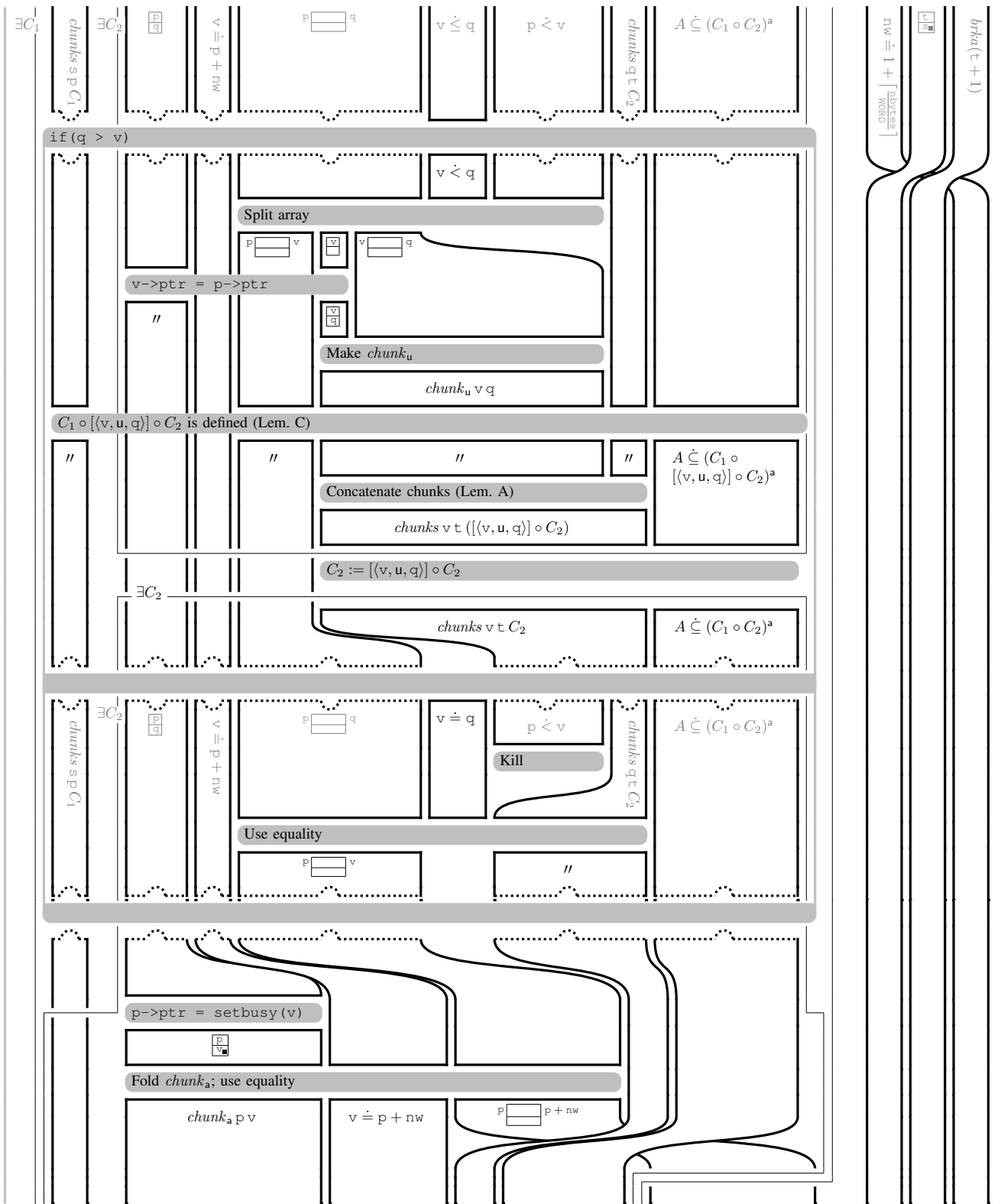
Continue from break

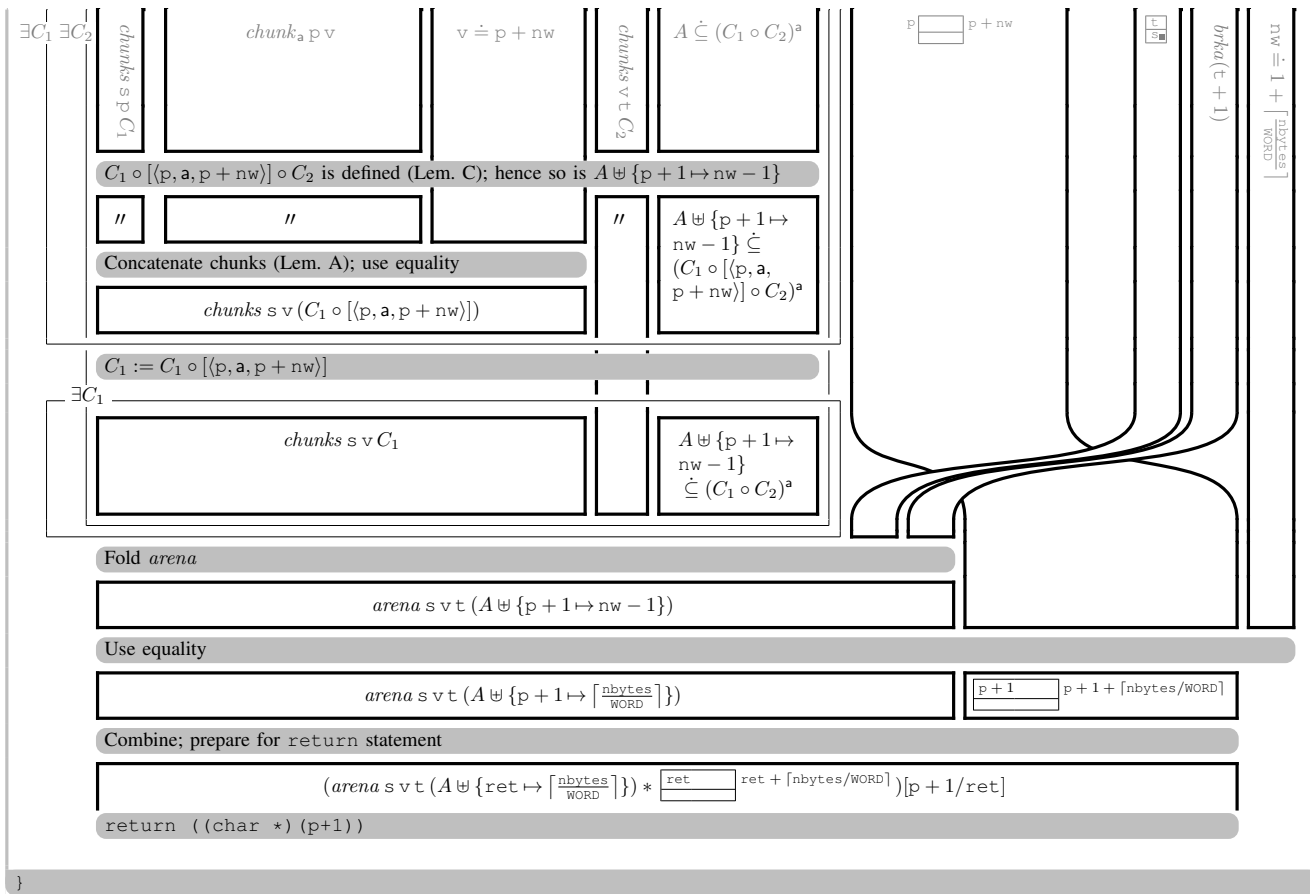


if ((int)q == -1) {









Establish postcondition

$(arena\ s\ v\ t\ (A \uplus \{ret \mapsto \lceil \frac{nbytes}{WORD} \rceil \})) * \boxed{ret} \quad ret + \lceil \frac{nbytes}{WORD} \rceil \vee (arena\ s\ v\ t\ A * ret \doteq 0)$

