Ribbon proofs and dynamically scoped quantifiers

John Wickerson

Technische Universität Berlin, Germany

1 Introduction

A graphical proof system, called *ribbon proofs*, for the propositional fragment of the logic of bunched implications was introduced by Bean [1]. Recently, Wickerson et al. upgraded ribbon proofs to a program logic by adding support for commands and existential quantifiers [10]. An existential quantifier is handled in a ribbon proof by drawing an 'existential box', labelled with the name of the variable being quantified, around a portion of the diagram. In their formalisation of ribbon proofs, Wickerson et al. assume that existential boxes are rectangular and well-nested. In practice, however, these boxes often partially overlap, or form more curious polygons than rectangles. Wickerson et al. mitigate this discrepancy between theory and practice by pointing out that overlapping polygons can be transformed into nested rectangles. Nevertheless, it is somewhat dissatisfying not to have a semantics for ribbon proofs directly.

This paper describes a direct semantics for ribbon proofs whose existential boxes may be non-rectangular and may overlap. In doing so, we hope to provide a deeper understanding of this 'intriguing proof structure' [10].

Outline Section 2 explains the problem with the current semantics for ribbon proofs. Section 3 explains the new semantics, which involves dynamically-scoped quantifiers. Section 4 debates the importance of this new semantics, and the need for existential boxes at all. Section 5 describes the idea of dynamically-scoped quantifiers in the more general setting of first-order logic. Section 6 proposes a semantics for dynamically-scoped quantifiers. Section 7 contains pointers to related work.

2 Starting point

Figure 1 shows a little ribbon proof.

The proof depicts a pattern that commonly arises when verifying a while-loop. Prior to the loop, we establish the loop invariant $\exists\alpha.\,\exists\beta.\,I$. Upon entering the loop body, we assume the test condition b. We then bring b into the scope of the α and β quantifiers; this operation is sensible since b (which is part of the program text) cannot mention the logical variables α or β (which exist only in the proof). We perform the loop body command C_1, which, in this case, produces three postconditions, p_1 and p_2 and p_3. We commute the scopes of the logical variables, to bring α inside the scope of β, and then pick a new witness for α. The 'Pick new α' step consumes α's existential box, and produces a new one below. We then commute the logical variable scopes again, to bring α back outside the scope of β, and pick a new witness for β. In doing so, we reestablish the loop invariant. After the loop, we assume the negation of the test condition, $\neg b$.

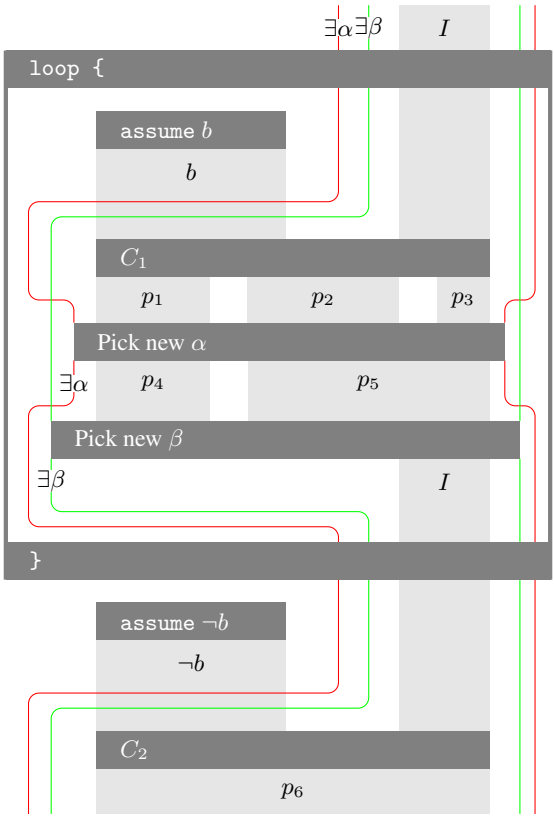
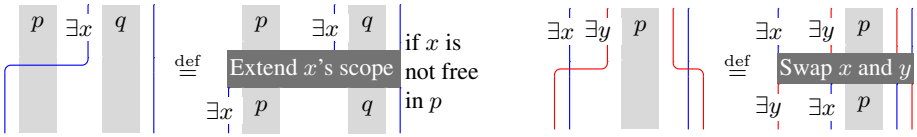


Fig. 1: A ribbon proof

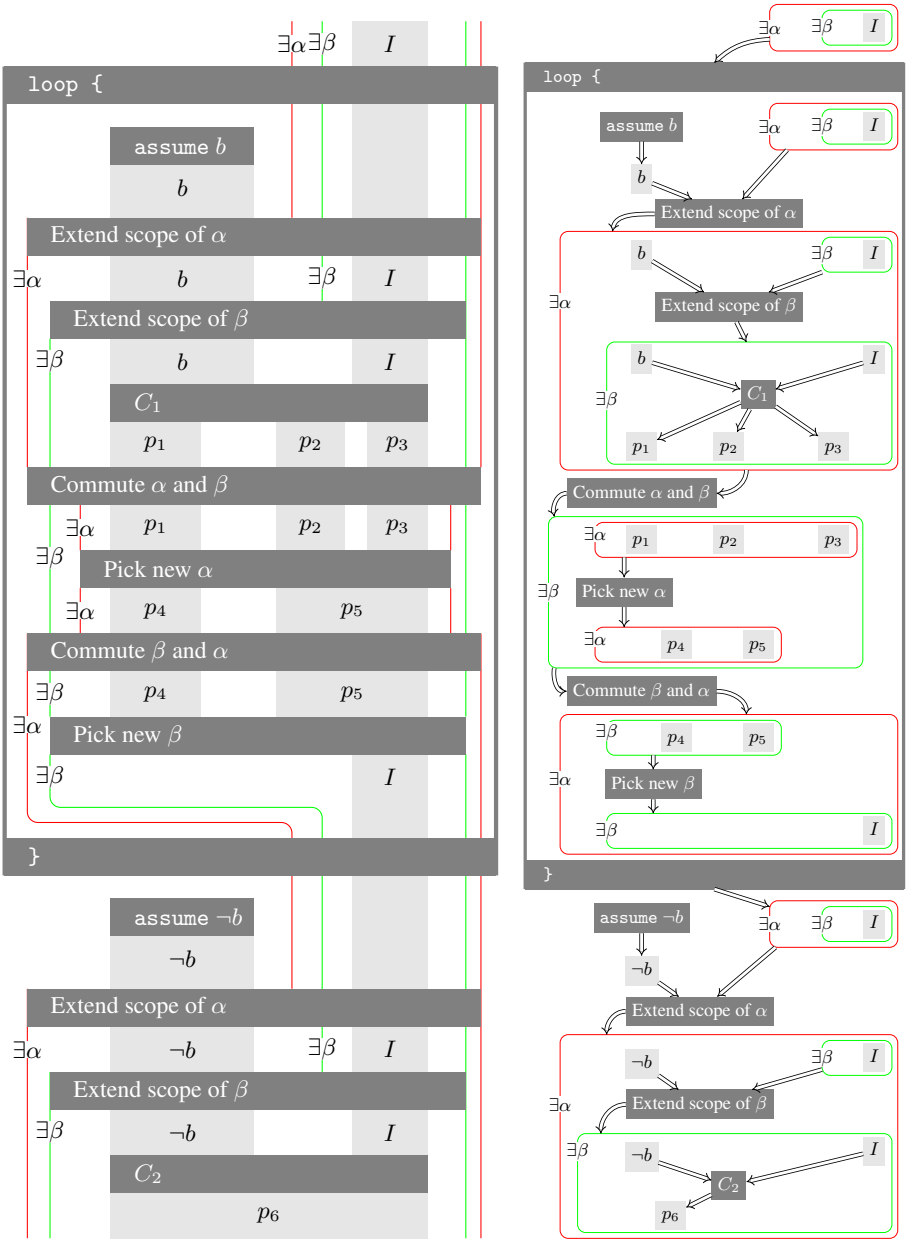
Let us now examine the semantics of such a picture. The formal language of ribbon proofs given in previous work [10] does not permit existential boxes to assume the exotic shapes seen in Fig. 1. The 'scope extension' steps and the 'commuting' steps are not allowed. They must be 'desugared' according to the following transformations:



We thus obtain the picture shown in Fig. 2a.

It is a shame that the picture has become significantly more complex, and that the interesting structure of overlapping quantifiers can no longer be easily seen.

Figure 3 depicts how Fig. 2a is then analysed as a graph. The graph has a hierarchical structure, in that loops and existential boxes are nodes that contain within them a complete graph. Besides this nesting hierarchy, we have a dag structure formed by the $\Rightarrow$-arrows. These arrows link commands to their pre- and postconditions.



(a) A ribbon proof, with scope-extending and scope-commuting shown as explicit proof steps

(b) The corresponding graph

Fig. 2

3 Main ideas

In this section, we shall describe how to obtain a graphical semantics for pictures such as Fig. 1, that results in much more concise graphs than that shown in Fig. 2b.

We shall proceed by a process of iterative refinement. Our first step is to flatten the existential boxes.

3.1 Flattening existential boxes

Figure 3 is obtained from Fig. 2b by replacing the existential boxes with single nodes, each of which is linked to the contents of the corresponding box by dotted arrows. We can think of this process as 'unpacking' or 'flattening' the existential boxes. Where previously the contents of each existential box was a completely separate graph, we now have nodes inside and outside the box co-existing in the same graph. (For now, we shall leave the loop-block as a special node that contains a complete graph.)

Note that the dotted arrows form a tree structure; that is, no node is a *direct* descendent of more than one existential quantifier. Previously, this 'tree' property was enforced by the fact that existential boxes must be well-nested.

We have only linked each existential node to the *assertion nodes* in its scope, not the *command nodes*. This is our first Main Idea.

> **Main Idea 1.** It is not necessary for command nodes to be located within the scope of existential quantifiers.

To clarify: observe that at the bottom of Fig. 2b, C_2 and p_6 are both visually inside β's existential box. It is important that p_6 is within β's scope, in case p_6 mentions β. But C_2, being a piece of program text, cannot mention logical variables, because these appear only in the proof. Therefore, at the bottom of Fig. 3, we *do not* have a dotted arrow from $\exists\beta$ to C_2, but we *do* have one from $\exists\beta$ to p_6.

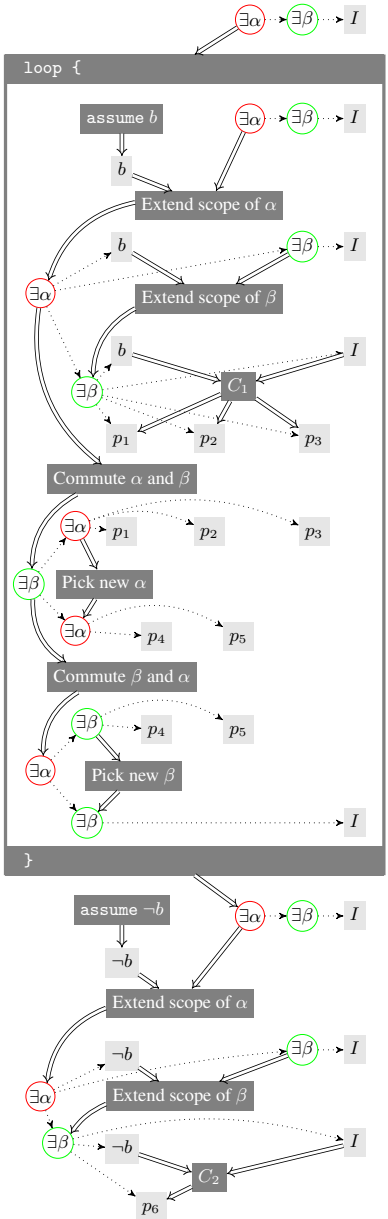
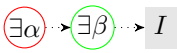


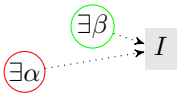
Fig. 3: First refinement: flattening existential boxes

3.2 Relaxing the tree property

In Fig. 3, the dotted arrows form a tree structure: each assertion node and each existential node is the target of at most one dotted arrow, and these arrows do not form cycles. Let us relax this constraint, and instead arrange that each assertion node receives an arrow *directly* from those quantifiers that bind its variables. As such, an assertion node may have more than one 'parent' in this dotted-arrow hierarchy. For instance, Fig. 3 begins with



while Fig. 4, which is obtained from Fig. 3 by relaxing the tree constraint, begins with



Observe that we have removed the ordering between $\exists\alpha$ and $\exists\beta$: they now both link to I, but not to each other. We have thus obtained a representation that is agnostic about the order of quantifiers. As such, the two 'Commute...' steps in Fig. 3 are no longer needed in Fig. 4. The 'Extend scope...' steps from Fig. 3 also become redundant, because we no longer have the concept of a variable scoping over a formula: instead the quantifiers link directly to those parts of the formula where the variable is used. In essence, we have moved from 'statically-scoped quantifiers' to 'dynamically-scoped quantifiers'.

> **Main Idea 2.** Dynamically-scoped quantifiers, where each variable is linked directly to its binder, are more appropriate than statically-scoped quantifiers here.

Figure 4 captures the intention of the picture in Fig. 1 quite well. However, there remains a little too much repetition. This repetition appears at the boundaries of the loop block, so we shall turn our attention to loops now.

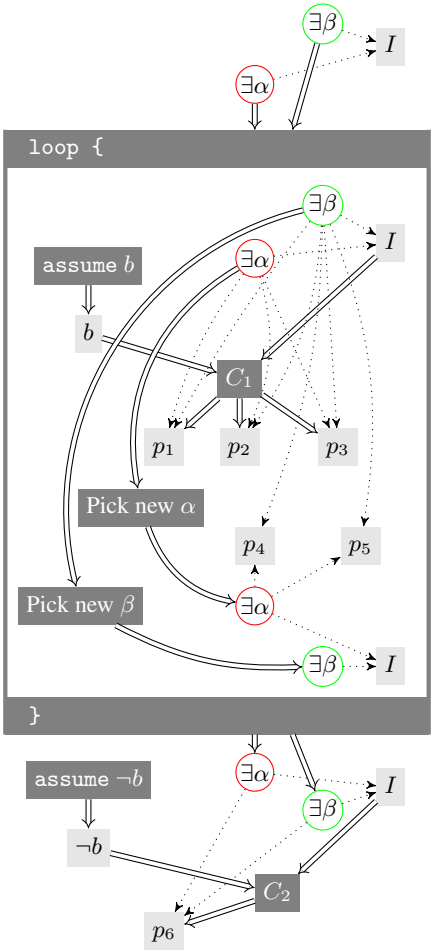
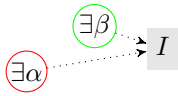


Fig. 4: Second refinement: relaxing the ‘tree’ property on dotted arrows

3.3 Flattening loop nodes

The loop invariant



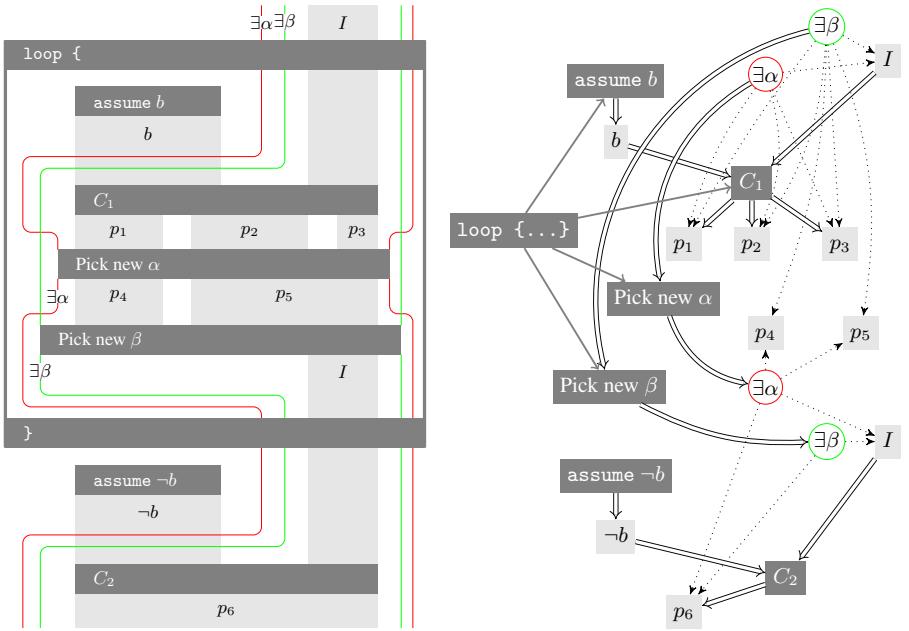
appears in four places in Fig. 4: above the loop, at the top of the loop body, at the bottom of the loop body, and below the loop. In Fig. 1, this loop invariant appears only twice: once above the loop, and once at the bottom of the loop body; the other positions can be inferred. This repetition in Fig. 4 results from the requirement that each loop block contains a complete, separate graph.

We have previously 'flattened' existential boxes; let us now perform this same trick on loop blocks. Figure 5b is the same as Fig. 4, but the loop block has been replaced by a single node labelled '`loop {...}`' that is linked to each command in the loop body by a solid grey arrow.

Note that it is not necessary to link the '`loop {...}`' node to the *assertions* that appear inside the loop body, only the *commands*. (This is something of a reversal of Main Idea 1.) This means that we no longer need one copy of the loop invariant just before the loop and one at the top of the loop body – the same nodes can now be reused in both cases.

> **Main Idea 3.** It is not necessary for assertion nodes to be located within the scope of loops.

Note also that the loop node does not have pre- or postconditions. That is, no ⇒-arrows enter or leave it. In fact, the only arrows that are permitted to connect to loop nodes are the solid grey ones. The loop invariant can still be deduced by inspecting the graph, however.



(a) A ribbon proof (repeated from Fig. 1) (b) Final refinement: flattening loop nodes

Fig. 5

4 Discussion

The refined graphs presented in the previous section seem to shed light on the 'dynamic scoping of existential boxes' seen in ribbon proofs [10], and as such, the primary objective of this paper has been achieved. We must question, however, the practical utility of the observations reported in this paper.

First, the refined graphs are much harder to formalise, and hence to prove sound. What makes graphs such as the one in Fig. 2b easier to work with is the fact that each loop node and each existential box contains a complete graph that is isolated from the rest of the diagram. In contrast, our refined graphs assemble all the nodes together as siblings, and allow arrows to pass between them quite freely, subject to some fairly subtle semantic constraints. This difficulty is surpassable, but initial investigations, using the Isabelle proof assistant, suggest that a considerable effort is required.

Second, we must ask whether our refined graphs are needed at all, when there exists an alternative semantics for ribbon proofs that does not involve graphs at all. Wickerson et al. [10] describe two formalisations of ribbon proofs: a 'stratified' version, which is sensitive to the particular layout of proof steps, and a 'graphical' version, that considers only the connectivity between the proof steps, ignoring their layout. The stratified semantics is the simpler of the two, does not require graphs, and does not require the 'variables-as-resource' scheme. The advantage of the graphical semantics is that proofs can be identified up to graph isomorphism, and this can be helpful when, for instance, analysing dependencies within the proof. If ribbon proofs prove helpful in this arena, then the observations reported here should be quite helpful.

Third, we question the utility of existential boxes in ribbon proofs in general. It is fairly clear that the ribbons themselves provide a useful intuition about the flow of resources through a proof. But it is not clear that the shapes of the existential boxes are so instructive. The existential boxes in the ribbon proofs shown in this paper are highly localised; that is, the rounded rectangles depict exactly those ribbons in which the existential binds. Perhaps this is giving too much information, and cluttering our diagrams. Figure 6 shows an alternative to Fig. 1, where each quantifier's horizontal extent is not shown, only its vertical extent. Although this diagram is simpler, it is less clear how to parse such a diagram as a graph.

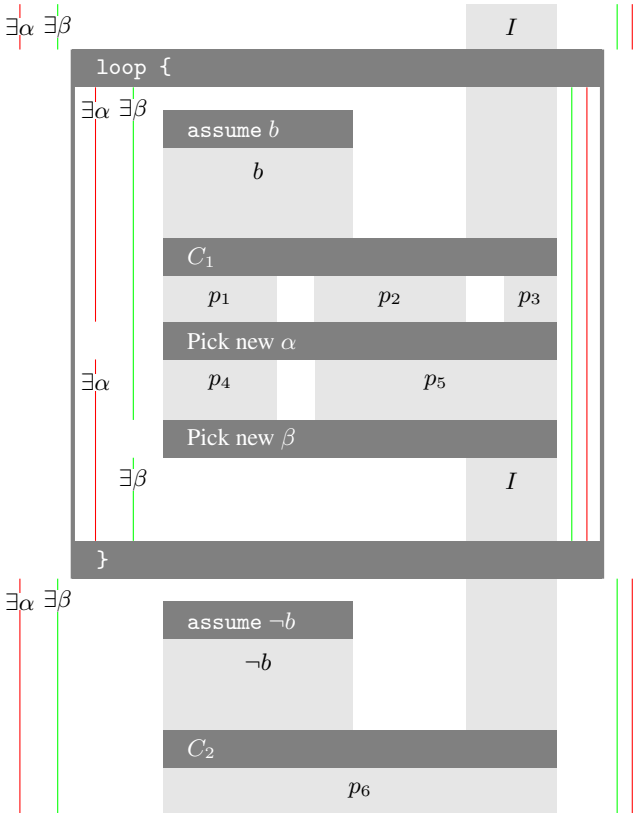


Fig. 6: A ribbon proof

5 Dynamically-scoped quantifiers

This section introduces the idea of dynamically-scoped quantifiers.

Consider the following logical formula:

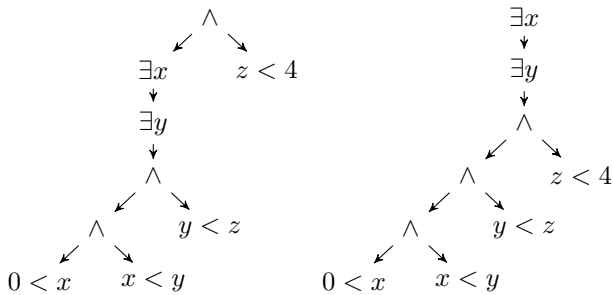
$$(\exists x.\, \exists y.\, 0 < x \wedge x < y \wedge y < z) \wedge z < 4. \tag{1}$$

Suppose we are interested in normalising such formulas. We could do so by pulling the quantifiers as far 'outside' as possible. In our example, this would involve extending the scopes of x and y to encompass the '$z < 4$' conjunct, like so:

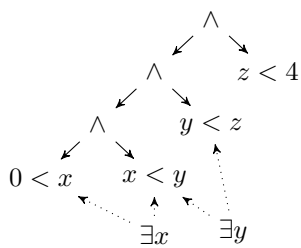
$$\exists x.\, \exists y.\, 0 < x \wedge x < y \wedge y < z \wedge z < 4. \tag{2}$$

Alternatively, we might prefer to normalise by pushing the quantifiers as far 'inside' as possible, such that the scope of each quantified variable includes only those conjuncts which mention that variable. In general, this is not possible. In our example, we should like x to scope over $0 < x \wedge x < y$, and y to scope over $x < y \wedge y < z$. Since these fragments of the formula overlap, we cannot attain this with 'statically-scoped' quantifiers.

Formulae (1) and (2) can be represented by the following syntax trees.



In this paper, we expand our attention from syntax *trees* to syntax *graphs*. We can hence represent a version of this formula whose quantifiers are pushed 'inside' as far as possible.



(3)

We have decorated the syntax tree with extra 'dotted' edges that connect each variable-binder with just those leaves which refer to that variable.

5.1 Dynamically-scoped quantifiers and first-order logic

We describe in this section how dynamically-scoped quantifiers may form a useful component of a graphical proof system for first-order logic.

Consider the following trivial example. Suppose we are given, for all x, y and z,

$$P(x) \implies \exists w.\, R(w, x) \tag{4}$$
$$Q(x) \implies \exists w.\, R(x, w) \tag{5}$$
$$R(x, y) \wedge R(y, z) \implies S(x, z) \tag{6}$$

and we are to prove:

$$\exists y.\, P(y) \wedge Q(y) \implies \exists x.\, \exists z.\, S(x, z)$$

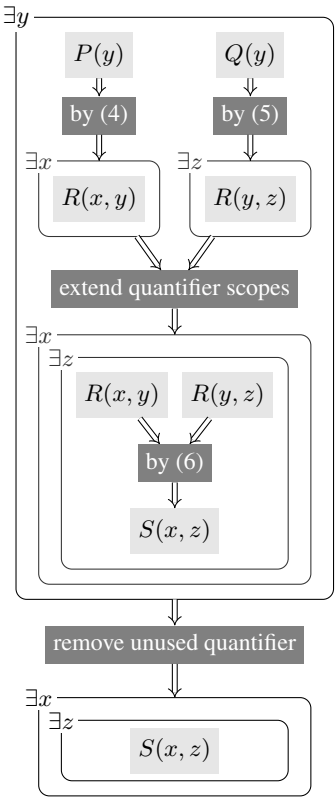
The proof can be written out 'line by line', like so:

$$\begin{aligned}
& \exists y.\, P(y) \wedge Q(y) \\
\implies & \text{ by (4)} \\
& \exists y.\, (\exists x.\, R(x, y)) \wedge Q(y) \\
\implies & \text{ by (5)} \\
& \exists y.\, (\exists x.\, R(x, y)) \wedge (\exists z.\, R(y, z)) \\
\implies & \text{ extend quantifier scopes} \\
& \exists y.\, \exists x.\, \exists z.\, R(x, y) \wedge R(y, z) \\
\implies & \text{ by (6)} \\
& \exists y.\, \exists x.\, \exists z.\, S(x, z) \\
\implies & \text{ remove unused quantifier} \\
& \exists x.\, \exists z.\, S(x, z)
\end{aligned}$$

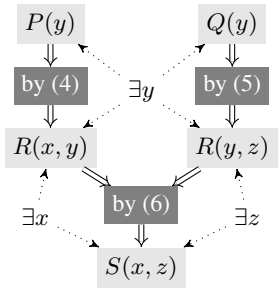
The proof can alternatively be depicted as a graph whose edges correspond to implications. In the picture shown in Fig. 7a, nodes side-by-side are to be read as conjoined, and the scope of $\exists$-quantifiers is depicted using a nesting hierarchy.

Note that those steps that manipulate quantifiers cause much repetition and additional complexity. If we move to using dynamically-scoped quantifiers, we can represent this proof much more concisely, as shown in Fig. 7b.

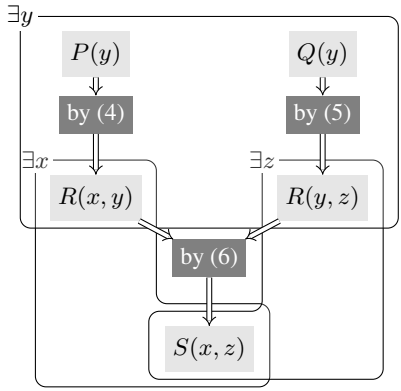
Rather than dotted arrows, it may be preferable to continue using boxes to delimit the scope of existential quantifiers. Of course, these boxes must now overlap. See Fig. 7c.



(a) Statically-scoped quantifiers



(b) Dynamically-scoped quantifiers, with scoping via dotted arrows



(c) Dynamically-scoped quantifiers, with scoping via (overlapping) boxes

Fig. 7: A simple proof in first-order logic

6 Semantics of dynamically-scoped quantifiers

In our formalisation, we shall avoid giving names to existentially-quantified variables. This is sensible because the main purpose of names for variables is to link variables to the relevant quantifier, but we now have dotted arrows to do this. The removal of names reduces the complexity of the formalisation.

Let Var be an infinite set of variable names, and $\mathcal{A}$ be a set of assertions that may mention variables in Var. The set Val contains the values that these variables range over. Let $\mathcal{N}$ be a set of assertion-nodes.

6.1 Dynamic formulas

Definition 1 (Syntax of dynamic formulas). *A dynamic formula is a tuple (A, V, Λ) where*

- *A is a finite set of assertion-nodes, drawn from $\mathcal{N}$;*
- *V is a finite set of variables, drawn from Var; and*
- *$\Lambda : V \to \mathcal{A}$ labels each assertion-node with an assertion.*

We write DF for the set of dynamic formulas. The components of a dynamic formula P can be projected like so:

$$P = (A_P, V_P, \Lambda_P)$$

Definition 2 (Scope of a bound variable). *For each dynamic formula P, the $scope_P$ function assigns to each variable in V_P the set of assertion-nodes in A_P that are in its scope:*

$$scope_P(v) = \{a \in A_P \mid v \in fv(\Lambda_P(a))\}.$$

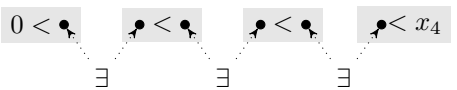
Example 1 (A dynamic formula). Suppose a_1, a_2, a_3 and a_4 are distinct elements of $\mathcal{N}$, and x_1, x_2, x_3 and x_4 are distinct elements of Var. The dynamic formula

$$P = (A, V, \Lambda),$$

where

$$\begin{aligned} A &= \{a_1, a_2, a_3, a_4\} \\ V &= \{x_1, x_2, x_3\} \\ \Lambda &= \{a_1 \mapsto \text{“}0 < x_1\text{”}, a_2 \mapsto \text{“}x_1 < x_2\text{”}, \\ &\qquad a_3 \mapsto \text{“}x_2 < x_3\text{”}, a_4 \mapsto \text{“}x_3 < x_4\text{”}\} \end{aligned}$$

can be visualised like so.



The particular choice of nodes is inconsequential; all that matters is the shape of the graph. Accordingly, we quotient the set DF of dynamic formulas by the following equivalence.

Definition 3 (Support equivalence). *Let us say that dynamic formulas P and Q are* support equivalent *if there exists a pair $(\pi_{\mathrm{ass}}, \pi_{\mathrm{var}})$ of bijections $\pi_{\mathrm{ass}} : A_P \to A_Q$ and $\pi_{\mathrm{var}} : V_P \to V_Q$ such that the following diagram commutes.*

$$\begin{array}{ccc} V_P & \xrightarrow{\pi_{\mathrm{ass}}} & V_Q \\ {\scriptstyle \Lambda_P}\downarrow & & \downarrow{\scriptstyle \Lambda_Q} \\ \mathcal{A} & \xrightarrow{\pi_{\mathrm{var}}} & \mathcal{A} \end{array}$$

Here we tacitly lift π_{var} to act on assertions, where it substitutes free variables in the usual way.

6.2 Meaning of dynamic formulas: by translation

One way to assign meaning to a dynamic formula is to translate it into a static formula.

Definition 4 (Translating dynamic formulas to static formulas). *It is straightforward to translate a dynamic formula to a static one, using the following Tr function:*

$$Tr(A, \{x_1, \ldots, x_n\}, \Lambda) \stackrel{\mathrm{def}}{=} \exists x_1. \cdots \exists x_n. \, (*_{a \in A}(\Lambda\, a)).$$

Example 2. The application of the Tr function to the dynamic formula P introduced in Example 1 yields the following static formula:

$$Tr(P) = \exists x_1, x_2, x_3. \, (0 < x_1) * (x_1 < x_2) * (x_2 < x_3) * (x_3 < x_4).$$

6.3 Meaning of dynamic formulas: direct semantics

Another way to assign meanings to dynamic formulas is to adapt the Tarksi-style semantics of ordinary first-order logic. We define

$$[\![(A, V, \Lambda)]\!]_\rho = \bigcup_{\rho' : V \to \mathsf{Val}} [\![*_{a \in A}(\Lambda\, a)]\!]_{\rho''}$$

where

$$\rho'' = \lambda x. \begin{cases} \rho'\, x & \text{if } x \in V \\ \rho\, x & \text{otherwise.} \end{cases}$$

Let us clarify the definition above with an example.

Example 3. Consider the dynamic formula introduced in Example 1, interpreted as a formula of first-order logic, and let Val be the natural numbers. To see that this formula is satisfiable, choose an interpretation ρ such that $\rho\, x_4 = 12$. Then define

$$\rho' = \{x_1 \mapsto 3, x_2 \mapsto 6, x_3 \mapsto 9\}.$$

Consequently,

$$\rho'' \supseteq \{x_1 \mapsto 3, x_2 \mapsto 6, x_3 \mapsto 9, x_4 \mapsto 12\}.$$

It is then easy to see that $[\![0 < x_1]\!]_{\rho''}$ and $[\![x_1 < x_2]\!]_{\rho''}$ and $[\![x_2 < x_3]\!]_{\rho''}$ and $[\![x_3 < x_4]\!]_{\rho''}$ all hold.

6.4 Dynamic contexts

We shall find ourselves wanting to compose dynamic formulas into contexts. For instance, we would like to say that if $P \Longrightarrow Q$ is a valid entailment, then so is $\mathcal{C}[P] \Longrightarrow \mathcal{C}[Q]$, for any context $\mathcal{C}$. What then, is a context?

Definition 5 (Dynamic context). *A dynamic context is simply a dynamic formula that is quotiented by the a stronger version of support equivalence that allows renaming of assertion-nodes but not variables.*

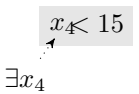
Let us say that dynamic contexts P and Q are strongly support equivalent *if*

- $V_P = V_Q$ *and*
- *there exists a bijection $\pi_{\mathrm{ass}} : A_P \to A_Q$ such that $\Lambda_Q \circ \pi_{\mathrm{ass}} = \Lambda_P$.*

Example 4 (A dynamic context). Suppose $a_5 \in \mathcal{N}$ and $x_4 \in \mathsf{Var}$. The dynamic context

$$\mathcal{C} = (\{a_5\}, \{x_4\}, \{a_5 \mapsto \text{“}x_4 < 15\text{”}\})$$

can be visualised like so.



Essentially, dynamic contexts are the same as dynamic formulas, but the variable names are made explicit. (As such, the dotted arrows become technically redundant.)

Let us now turn to the matter of inserting a dynamic formula into a dynamic context.

Definition 6 (Inserting a dynamic formula into a dynamic context). *Given a dynamic context $\mathcal{C} = (A_\mathcal{C}, V_\mathcal{C}, \Lambda_\mathcal{C})$ and a dynamic formula $P = (A_P, V_P, \Lambda_P)$, where $A_\mathcal{C} \cap A_P = \emptyset$ and $V_\mathcal{C} \cap V_P = \emptyset$, then*

$$\mathcal{C}[P] = (A_\mathcal{C} \cup A_P, V_\mathcal{C} \cup V_P, \Lambda_\mathcal{C} \uplus \Lambda_P)$$

Let us clarify the definition above with an example.

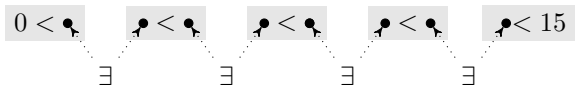
Example 5 (Inserting a dynamic formula into a dynamic context). The insertion of the dynamic formula P (from Example 1) into the dynamic context $\mathcal{C}$ (from Example 4),

$$\mathcal{C}[P] = (A, V, \Lambda),$$

where

$$\begin{aligned} A &= \{a_1, a_2, a_3, a_4, a_5\} \\ V &= \{x_1, x_2, x_3, x_4\} \\ \Lambda &= \{a_1 \mapsto \text{“}0 < x_1\text{”}, a_2 \mapsto \text{“}x_1 < x_2\text{”}, a_3 \mapsto \text{“}x_2 < x_3\text{”}, \\ &\quad\ a_4 \mapsto \text{“}x_3 < x_4\text{”}, a_5 \mapsto \text{“}x_4 < 15\text{”}\}, \end{aligned}$$

can be visualised like so.



Note that the x_4 variable has become bound, and hence no longer appears in the diagram.

7 Related Work

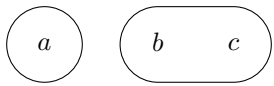
The idea of 'dynamically-scoped quantifiers' has appeared in various guises in various fields of study.

7.1 Peirce's existential graphs

Peirce's system of existential graphs [3, 8] involves dynamic scoping of existential quantifiers. In his system, formulas are written on a two-dimensional page, juxtaposition corresponds to conjunction, and drawing a ring around a fragment of the page negates the formulas therein. For instance, the formula

$$\neg a \wedge \neg(b \wedge c)$$

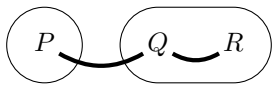
is depicted as the existential graph



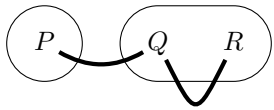
Existential quantifiers are depicted as heavy lines that link occurrences of the same variable. The variables do not have names. For instance, if P and R are unary predicates and Q is a binary predicate, then the formula

$$\exists x.\, \neg P\, x \wedge \neg(\exists y.\, Q\, x\, y \wedge R\, y)$$

is depicted as



Note that the heavy line for the y-quantifier lies entirely within the right-hand ring. If the line is pulled outside of that ring, like so



then we are several ways to parse the existential graph, all equivalent:

$$\begin{aligned} \exists x.\, \neg P\, x \wedge (\exists y.\, \neg(Q\, x\, y \wedge R\, y)) \\ \exists x.\, \exists y.\, \neg P\, x \wedge \neg(Q\, x\, y \wedge R\, y) \\ \exists y.\, \exists x.\, \neg P\, x \wedge \neg(Q\, x\, y \wedge R\, y). \end{aligned}$$

Peirce's existential graphs do not have a notion of free variables; all variables are bound.

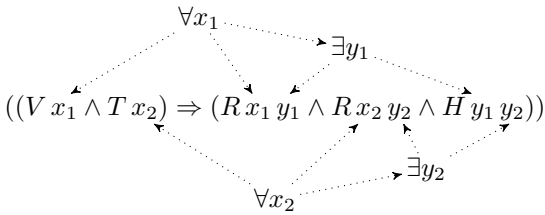
7.2 Branching quantifiers

Henkin's branching quantifiers [5] can be used to express certain statements that cannot be expressed within first-order logic. For instance, Hintikka [6] proposes the sentence "some relative of each villager and some relative of each townsman hate each other" as one that cannot be adequately captured using first-order logic, and proposes instead to write

$$\begin{pmatrix} \forall x_1 & \exists y_1 \\ \forall x_2 & \exists y_2 \end{pmatrix} ((V\,x_1 \wedge T\,x_2) \Rightarrow (R\,x_1\,y_1 \wedge R\,x_2\,y_2 \wedge H\,y_1\,y_2))$$

The matrix of quantifiers at the beginning of this formula indicates that the witness for y_1 depends on x_1 but not on x_2, and that the witness for y_2 depends on x_2 but not on x_1. Such complex dependencies are inexpressible using the statically-scoped quantifiers of first-order logic.

It may be possible to express Hintikka's sentence using dynamically-scoped quantifiers, as suggested by the following picture.



7.3 Dynamic logic

Groenendijk and Stokhof [2] describe an extension of dynamic logic [4], called dynamic predicate logic, in which quantifiers are thought of as state-modifiers. Their idea has been applied to natural language processing, to provide a compositional parsing of multi-sentence phrases. Formulas no longer denote predicates on variable interpretations, but are now *relations between* variable interpretations. In other words, each formula is a program that modifies the variable interpretation as it is 'executed' from left to right. For instance, '$\exists x$' is an action that havocs the value of x.

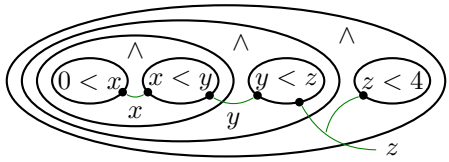
Vermeulen [9] proposes an extension of dynamic predicate logic in which each variable denotes a stack of values. The action $\exists x$ does not simply overwrite x with an arbitrary value, but rather pushes an arbitrary value onto x's stack. The new action $x\sqsupseteq$ pops from x's stack. Writing 'ϕ_0 ; ϕ_1' to denote relational composition, his system can express our formula (1) in a form in which the quantifiers are pushed 'inside' in a normalised way, like so:

$$\exists x\ ;\ (0 < x)\ ;\ \exists y\ ;\ (x < y)\ ;\ x\sqsupseteq\ ;\ (y < z)\ ;\ y\sqsupseteq\ ;\ (z < 4)$$

Vermeulen's system is able to express formulas using fewer variable names. It is very different from our proposed system, which does not involve a dynamic semantics.

7.4 Milner's bigraphs

Our dynamically-scoped quantifiers are somewhat reminiscent of Milner's bigraphs [7]. Here is formula (3) again, drawn this time as a bigraph.



Note that each *atomic control* has one *port* per free variable. The *link* corresponding to the z variable is an *outer name*, so that it can later be bound when this bigraph is substituted into a broader context.

Acknowledgements The author thanks Thomas Göthel for many helpful suggestions and stimulating discussions, and acknowledges the support of a postdoctoral scholarship from the German Academic Exchange Service (DAAD).

References

1. J. Bean. *Ribbon Proofs - A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary University of London, 2006.
2. J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 1990.
3. E. M. Hammer. Semantics for existential graphs. *Journal of Philosophical Logic*, 27(5):489–503, 1998.
4. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
5. L. Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods: Proceedings of the Symposium on Foundations of Mathematics*, pages 167–183. Państwowe Wydawnictwo Naukowe and Pergamon Press, Warsaw, 1961.
6. J. Hintikka. Quantifiers vs. quantification theory. *Linguistic Inquiry*, 5(2):153–177, 1974.
7. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
8. C. S. Peirce. Existential graphs. In *A Syllabus of Certain Topics of Logic*, pages 15–23. Alfred Mudge & Son, 1903.
9. C. F. M. Vermeulen. Variables as stacks: A case study in dynamic model theory. *Journal of Logic, Language and Information*, 9:143–167, 2000.
10. J. Wickerson, M. Dodds, and M. J. Parkinson. Ribbon proofs for separation logic. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, 2013.