Advance Access publication on August 18, 2010
doi:10.1093/comjnl/bxq061

Signposting in Documents

HAROLD THIMBLEBY*

Future Interaction Lab, Swansea University, Singleton Park, Swansea, SA2 8PP, UK
**Corresponding author: harold@thimbleby.net URI: harold.thimbleby.net*

Signposts make complex documents more flexible and easier to read; signposts exist in many forms, but are familiar as cross references and as hypertext links. Signposting systems need to be carefully designed, so that they are reliable, easy to use for readers yet convenient for authors to embed in their writing. There are fundamental problems in achieving good signposting, which this paper explores through cross-referencing in the popular medium of LaTeX, which is often the typesetting system of choice for heavily cross-referenced documents. This paper provides an implementation of cross references for LaTeX users. Problems with LaTeX and TeX arise, and are explored and mostly solved: some problems are due to the designs of these systems, but, crucially, some problems are unavoidable, an inevitable part of signposting in principle, whatever systems are used, even including the web.

Keywords: document processing; cross referencing; signposts; LaTeX; TeX

Received 10 October 2009; revised 8 July 2010
Handling editor: Peter Robinson

1. INTRODUCTION

Document processing raises many interesting and deep issues in computer science. Even without considering its contribution to user interfaces and interactivity (as in WYSIWYG word processing systems [1, 2]), document processing invariably throws up problems that illuminate wider issues such as complexity, stability, parsing, ambiguity and programming language design. Many seemingly trivial features of documents conceal serious computational and conceptual difficulties, particularly if they are to be handled well by the computer.

This paper introduces *signposts* as one such document feature: they seem simple, they are essential in complex documents, yet they are very rarely supported well. Moreover, we shall see that even in very flexible programmable document processing systems problems unavoidably remain in implementing them correctly. The problems are non-trivial.

Since the invention of printing, the position on the page where text falls has become consistent and easily reproduced. Eisenstein [3] makes a powerful argument for the cultural transformation in thought that occurred in every area—science, religion and politics—from the 15th century onwards that the 'simple' advance of printing permitted. Every book of the same edition, or every handout to a student, has or can have consistent layout and page numbers. With consistency came the opportunity to refer reliably to locations within documents; with printing and cheaper paper the cultural transformation spread to all literature.

The now ubiquitous numbering of pages in Arabic numerals encouraged the development of tables of contents and indices, and these ideas in turn transformed how books were used. Tables of contents and indexes, for instance, enabled readers to find chapters or sections. Teachers could ask all students in a class to turn to a particular page in a text. The writing style changed too, as text *within* a document itself became able to use page numbers or section numbers, sometimes even line numbers (especially for critical editions), to direct readers to the right place to pursue some aspect of the narrative.

Magazines and newspapers very often present multiple articles on a single page but, for lack of space, continue the articles elsewhere on another page; each column then finishes with a direction to the reader to turn to another page and column so they can continue reading the article.

We call such directions to the reader *signposts*. The destinations of signposts are *labels*. Thus, a number in the usual position of a page number on a page is a label, and a page number in a phrase such as 'please turn to page 5' is a signpost.[1]

Our goal in this paper is to provide automatic and effective support for signposts in documents—but even now, we can start to see tricky computer science problems: what *exactly* is a signpost or label? What if the signpost says 'turn over'

[1]Where helpful, this paper uses the convention that text written, edited or intended to be seen by an author looks like `this` (typewriter font) and final or typeset text intended to be seen by a reader of a document looks like this (sans serif font).

(from where and where to?) or what if the destination page has no number, or if the numbers are Roman and not Arabic numerals? What of purely implicit signposts, such as the end of line, which most readers will take to mean 'continue reading on to start of next line'? Considering all the possible answers to such questions, which approaches are feasible to compute, sufficiently general, yet simple enough to use reliably, so authors and readers can take advantage of them without getting lost in restrictions and exceptions?

Documents, from books to journal articles, even websites, are a mixture of *signposts* and *labels*, and content, or *narrative*. Narrative is the story or message that the author wishes to convey, and signposts are the signs that help the reader navigate through that narrative, pursuing their own goals. Often the reader's goals evolve and need not fit in a simple way onto the author's structure. *Hypertext* is a style of document where the author has tried to produce many signposts to enable the reader to easily pursue any or many choices. Generally hypertext is managed interactively (as on the web), but it can be used in conventional print books, for instance giving the reader choices to develop a story as they wish.

Interactive documents may dynamically change signposting depending on reader activity, though the reader's experience is a little different (they cannot easily tell that their signposting is personalized) other than the technology—button pressing, say, rather than page turning. Examples of dynamic signposting are history mechanisms, familiar in web browsers.

Interactive signposting under the control of readers is possible too: the user may introduce bookmarks [4] that stand as signposts to where the reader is at the time. In paper documents, a reader may use physical bookmarks or sticky notes that protrude from the book, or they may turn over page corners. It is clear that readers like signposts.

Signposts can be computed. Generally, an author writes *signpost expressions* (HTML tags, MSWord field codes, LaTeX macros, as appropriate) and a reader sees their values as some number or text. With signpost expressions, the label itself need be nowhere directly visible to a reader, and even need not be visible to the author: for example, the signpost 'continues on column 2' refers to a location that need not be explicitly labelled for the reader's benefit, and (in many typesetting systems) the author will be editing a galley with no direct concept of column anyway.

Authors and readers have different views of documents: the author's mark-up (including signpost expressions) versus the reader's view of it in the typeset document (their values). As we shall see, the computation of signpost values is non-trivial, and popular mark-up systems such as LaTeX have additional problems that need to be worked around.

There are many forms of signpost:

- Tables of contents, indexes in the document itself, where people or lecturers (standing outside the document) can use page numbers to signpost readers to the right place in the narrative.
- When similar narratives appear in many editions and page sizes—the Bible and Shakespeare's plays being familiar examples—page numbers are rarely used in signposts. Instead, chapter and verse numbers or line numbers (referring to some agreed definitive manuscript) are used. The Bible is often printed with copious internal cross-references, though the present scheme of labelling (versification) was invented in the 13th century, long after the original authors had stopped writing. Labels are largely invariant between editions, as publishers are careful to preserve them.
- Documents with footnotes and endnotes typically signpost the notes, either using Arabic numbers or 'numbers' chosen from a sequence of characteristic symbols (e.g. †, ‡, ⋆, …) so that the reader knows which notes are which.
- Documents with chapters, sections, figures or tables may number these features so that they can more easily be referred to by the author and located by the reader.
- Documents may have cross references so that the reader can easily find related topics. Cross references may refer to section names, section numbers or page numbers.
- Many documents also have signposts pointing outside the document itself. For example, this document has a list of references at the end that refer to other documents elsewhere.
- Hypertext is a modern form of signposting—now most familiar from the web—where typically the target destination of the sign itself (called a *link*) is not visible to the user, but the computer, that is, the browser, follows the sign without the user needing to know where they are being taken. HTML allows the signpost that the readers sees, and the corresponding action that the computer takes if the reader follows the signpost, to be independent (which unfortunately is often exploited when the author has malicious intentions).
- The web has search engines that serve a similar purpose to conventional document indexes. A conventional document is indexed by someone, usually the author or a professional indexer, familiar with the document's narrative, and therefore the index signposts are relevant and useful to the reader. The web has no such author, and it is then a computational problem to estimate the relative importance of pages so that index signposts can refer to the 'best' pages [5]. Surprisingly, such techniques have not yet been used on conventional documents—though we will give an example below, in Section 8.2.
- An author may have 'private' signposts that a reader never sees. Story-telling software often allows an author to create various narrative structures, which the author will interleave to create mystery or tension in the reader. If the author has made the private signposts explicit, the reader could too easily jump to revelations or skip distractions! Yet the author needs the private signposts so that they can manage the sub-plots independently.

- An author may have private labels. Most mark-up languages permit this, as, for example, in HTML where an author uses a private label in the `<a href='label'>` tag, which need be nowhere visible to a reader.
- An author may use circumscription as a signpost; that is, rather than providing an explicit signpost and label (as in, 'please turn to section 3.2 on page 5') the author describes the signpost in a mixture of natural language and symbols (e.g. '⇒ next column') or in other ways (e.g. 'see later'). Page breaks are an implicit form of a signpost, taken to mean 'please turn to the next page.'
- Finally, we note that an author may use signposting playfully; Laurence Sterne's humourous and scatological novel *The Life and Times of Tristram Shandy, Gentleman* [6] is one of the earliest and most famous examples of this.[2] Interestingly, in some versions of *Tristram Shandy* publishers have had to resort to footnotes to adapt Sterne's original signposting, because volume numbers and other details have changed between editions (e.g. the opening sentence of book *six*, chapter one incorrectly refers to the previous *seven* volumes [6, p. 327]). Much research has been published on Sterne's work, which therefore requires its own signposts, such as '(1.21.66)' [8], pointing *into* Sterne's complex book: these labels would have been unfamiliar to Sterne himself.

All these examples of signposts require various important features for their success:

(1) it is crucial that signposting (and labelling) is reliable;
(2) the overhead for the author should not be so high as to be a distraction from their main task of writing;
(3) particularly when there are many signposts, the author will benefit from automatic tools to help manage them;
(4) signposting should define dependable relations for the reader of a document, but it is possible that the author may make some accidental errors while authoring and specifying the relations: the signposting system should report errors to the author;
(5) once the decision to use a particular signpost is made by the author, its content is determined, but the information the reader sees may change as the author edits and restructures the document;
(6) the author needs generality; accommodating readers' requirements may stretch any purely programmed approach. For example `<signpost id ='cont1' > &rarr; story continues </signpost>` might allow the author to both be precise about the signposting (hidden from the reader), but also to choose an arbitrary description (seen by the reader);
(7) finally, using signposts should be *graceful*, particularly for authors learning their trade who are starting to write large documents before they are fully familiar with the author's writing and signposting tools.

The concept of 'graceful signposts' requires elaboration. Consider an author writing a short document; initially, they may wish to number sections the 'easiest way'—namely, explicitly, 1, 2, 3… but as the document grows in length or complexity, they then resort to automating the signposting. The author may need special help during the transition to automation: otherwise reordering some sections will renumber automatically whilst, confusingly, reordering of others will not.

It is generally important that page numbers are consecutive integers placed distinctively in a consistent position on every page; and, once the author has decided to have a run of page numbers in a particular place on the page, the actual choice of numbers can be automated. Obviously, if the author adds another page, the system should renumber affected pages. Once the design decision has been made by the author, there is no subsequent work. However, not all signposts are as simple as page numbers. To help, LaTeX has a simple facility where it warns the author that signposts have changed, and it asks the author to reprocess the document so the signposts can be corrected.[3]

Authors typically do not want the burden of fully specifying the values of signposts as seen by readers. A computer can number pages; an author need not. An author may write a document, but the publisher may decide on a different design: for example, numbering chapter parts with Roman numerals rather than Arabic numerals—such a decision should not affect the author's use of signposts. What is more, if the author rewrites their document, and page contents change or chapter order changes, the computer can still renumber the document—renumber the pages or chapters reliably using the pattern already set. Ideally, the author need do nothing, other than concentrate on the narrative; once specified the signposts should look after themselves. Without computer support, the author would need to remember where all signposts were and fix them individually; this would be an error-prone process, particularly for signposts such as cross-references that do not appear, as page numbers do, in typographically distinct forms in a document.

Signposting is a natural task for computers to support as part of the document preparation process. Microsoft Word, for example, is easily set up to paginate, and it is not difficult to get it to number sections and footnotes, or to get it to insert tables of contents and indices—though the author has to manually keep track of house keeping to ensure reliability. Cross-references are harder, however, one problem being the unavoidable clash between needing a specification for signposts

[2] *Tristram Shandy* is also the source of the Tristram Shandy Paradox—itself of direct interest to computer scientists, but beyond our present scope as there is no space to discuss self-reference in this paper (which is exactly the paradox…). By the way, Knuth uses TeX to define TeX [7], and like Sterne he exploits the typography of his book to sustain the narrative.

[3] LaTeX has a facility where cross-references can themselves be signposted, so a reader can see what they are in the author's view of the document as well as in the reader's view of the document.

and wanting a WYSIWYG display. In contrast, LaTeX is a markup-based document preparation system that makes cross-references easier for the author—markup languages are not WYSIWYG, and signposts do not require extra modes, special manual intervention or house keeping, and the system is better placed to provide useful diagnostics.

This paper discusses issues around programming signposts, specifically cross-references, and it raises some fundamental computational problems with signposting, for which there is no perfect solution. Occasionally, authors *have* to rephrase text in order to achieve correct signposting; though rewriting to achieve good typography is nothing new, and indeed is a proper part of the pleasure of good typography, from aligning ascenders and descenders on consecutive lines to avoiding widows and orphans.

The examples in this paper are presented in terms of the popular system LaTeX [9], and in addition to the fundamental problems, we also find some problems in TeX, the engine underlying it [7]. Readers of this paper will find complete LaTeX code to generate nice cross-references in Section 6, below. Using them, authors will achieve improved cross-references directly. Once the software design problems are overcome, the code is simple and can readily be modified to achieve different styles, for instance, for signposts in other languages, such as French.

The goal is to have better cross-references to help authors refer to, and to help readers to find parts of documents, whether sections, equations, tables or figures. Cross-references can refer to page numbers and we would rather change a rigid style 'p. 5' into a context-sensitive style like 'next page', 'facing page' etc where appropriate. This is not only more specific and helpful, but creates stylistic variety. It is also important that cross-references are reliable, so a key requirement is to make the approach robust so that the author need not worry about it. Achieving these seemingly simple goals turns out to be a non-trivial undertaking, for many reasons, but the end result should be an easy-to-use feature: easy for the author, and easy and reliable for the reader.

2. SIGNPOSTS OR LINKS?

In the preceding discussion, we did not make a distinction between *signposts* and *links*. Signposts are seen and used by readers; links are seen and understood by computers. Often the intention of a link is to behave like a signpost.

In an HTML document, the author may write `<a href="#label">text</a>` and the reader would see 'text' (though the exact style can be controlled by style sheets [10]); then when the reader clicks on the link (the underlined text in this case) the browser takes them to the part of the document they are browsing named '`label`' or possibly even to a different document if the label provides a URI. However, what the users sees is not necessarily a signpost, since the behaviour is entirely internal to the browser and is invisible to the user; the browser follows the link and does all the work for the user. It may be considered good authoring style that the text refers the reader as if it is a signpost, but there are no guarantees that the text visible to the user is actually a signpost of any sort—though the author can try to make it an informal signpost, as in writing something like `see <a href="#label">above</a>`. Unfortunately, this signpost could break as soon as the author edits the document and changes the order of the document. Different sorts of signposts will break in different sorts of ways, for example, if the author referred to the name of the section, then if they later edit the name of the section, the signpost breaks even though the document order is unchanged.

A common problem on the web is that there is no guarantee for the author that the use of the labels is consistent in a document. It is possible for an author to write, say, `name=label ... href=#lable` (i.e. making a spelling mistake for a label name) and the link will simply not work! Unless the author uses additional tools, such errors may remain undetected for a long time.

The difference between links and signposts becomes more obvious when a document is printed on paper. Now the links *cannot* work, for the printed document is a conventional paper document and has no browser to follow links automatically for the reader. The only signposts that remain are those introduced by hand by the author informally and without support from the (in the web case) HTML mark-up.

It is possible to use the document object model (DOM) to generate signposts in an HTML document to support the author with automatic (hence reliable) features, for instance, using JavaScript, but unfortunately the DOM is unaware of page breaks, page numbers and other navigational features of conventional printed documents, so the signposting that can be provided is very limited. For reliable and flexible signposting, we have to turn to proper typesetting languages, such as LaTeX, which we now discuss at length.

3. SIGNPOSTS IN LaTeX

LaTeX is a very popular and straightforward typesetting language [9], which is implemented on top of TeX, a powerful, very flexible and idiosyncratic assembly language for typesetting.

In a document, the author writes sections, and in LaTeX the sections (and subsections, etc) are numbered automatically unless the author chooses not to have unnumbered sections.

The basic idea of a cross-reference is to enable the reader to easily find a relevant topic. To do this, the author *labels* a target section, and *refers* to it from the section the reader is assumed to be reading. If the author referred to the section by number, which is what the reader sees (as in this paper: sections are automatically numbered, and this section happens to be number 3), then this number could change if the author modified the document, for instance, deleting or inserting a new section, or

if the author moved this section to some point earlier or later in the document.

Since the section numbers are being provided automatically, it is likely that the author does not know or even especially care about the actual section numbers in any case. Indeed, if I actually wrote '`this is section 3`' in my original document—as if I knew which section this paragraph was in *when I wrote it*—and then later edited my document and inserted a new section, then this section might be renumbered as Section 4, which would then make the '3' the reader sees above incorrect. The same problem would arise if I cut-and-pasted this paragraph and put it somewhere else in the document, so that the text I wrote, '`section 3`', ended up a section other than Section 3.

Rather than write section numbers like '3', then, which may go wrong, an author associates a named label with a section (or other unit of text) in a document, and refers to the label name instead of writing the section number explicitly; the document processing system then converts the reference to the value it means, which here would be the correct section number—'3'.

Crucially, the label name, once chosen, need not change as the document structure evolves. In LaTeX, label names are bound to the section number and the page that it labels. Anywhere else in the document, the author can use the name to refer to the section number, even if it gets renumbered or has its title changed. Full details are given in the LaTeX manual [9].

Thus, in this section of this paper, I actually referred to a label '`selfreference-paragraph`' everywhere you read '3' (and I wrote a trivial calculation, plus one, for where you read '4'—in a paper on signposting it would have been very embarrassing if I had said this section was *not* Section 4 when in fact it was). This simple use of signposts ensured that what I wrote always refers to the correct section when it is read (i.e. referring to Section 3) even if earlier drafts of this paper had the example placed in a different section. It has taken me 3 or 4 years to write this paper, and it has been restructured and has changed enormously over time: at times this section has certainly gone under other section numbers. Nevertheless, at any time in the lifetime of this document, the signposting would work correctly, and whenever you might have read it, it would have been using the correct section number.

For a simpler example consider how a routine document might use and refer to cross-references:

```
\section{Example introduction}
\label{example-introduction}
...
\subsection{Far, far away}
Please refer to the introduction,
Section \ref{example-introduction}, which
can be found on page
\pageref{example-introduction}.
```

This example would generate typeset text such as this:

1 Example introduction

…

4.5 Far, far away

Please refer to the introduction, Section 1, which can be found on page 1123.

During editing, an author may add and remove sections anywhere, which would cause them to be renumbered, and LaTeX will update and resolve the cross-referencing transparently, with no further work from the author. Note that the example above illustrates the author's need for generality: the author referred to the introduction section by using a mnemonic label referring to its title. This happens to be fine at this point in the development of the document, but the label may become confusing if the author later changes the name of the section to 'Chapter 1', say.

In general, a signposting system cannot *guarantee* perfection, because the author might want to do anything. There are many variations on signposting, not all provided by LaTeX (and not all of which we can discuss in this paper), such as the following:

- The cross-reference name system in `cweb` [11] encourages authors to use long and mnemonic names for references, by providing a name abbreviation mechanism. Provided a name is defined in full at least once in a document, it may be referred to by unambiguous abbreviations anywhere. For example, an author could refer to the label (as shown immediately below) by writing a prefix such as `\ref{this section...}`.
- The cross-referencing system `xr` [12] provided such good error messages that authors used unreferenced labels as personal writing reminders. In LaTeX's syntax, authors wrote things like `\label{this section needs finishing}`, and `xr` summarized these 'errors'.
- Footnotes are a form of signpost with a conventional typographical form, as well as the constraint that footnotes are rarely, if at all, referenced by the author more than once. In consequence, a system like LaTeX supports signposting to footnotes without requiring any names; the author writes the footnote text where they want it referred to in the document `\footnote{As here.}`[4] and the typesetting system moves the text to the foot of the page (or to the end of the chapter as an 'end note', or where ever), labelling the reference and the footnote uniquely (here, by using a unique number).
- Cross-references are usually signposts to *places*. In contrast, indexes are signposts to *concepts* or words—and typically the same concept will occur in many places, unlike a cross-reference label, which in most systems has to be unique. LaTeX has an indexing system, `makeindex` that automatically sorts the words and provides page numbers.

[4] As here.

It does not provide an index to section numbers, though this can be programmed.

- An author (more so than a reader) may want additional meta-signposting, to help manage signposts during the authoring process. Where did the author use signposts, and what are they called (which the reader need not know)? Some tools exist to make indexes of cross-references.
- A reference work might make extensive use of topical cross-references. Here, the author would name topics. Like an index, the same cross-reference name would be used any number of times and many places throughout a document, but unlike an index, no 'index entries' would be collected at the end of the document. Instead, signposts would occur typically at the end of sections where each topic was mentioned—and the cross-referencing system would automatically refer to every *other* occurrence (correctly sorted)—that is, except for the current section.
- For some reference documents, a reader may like a blend of cross-reference and index. In an ordinary document, if the reader wants to read more about the current topic, they must decide what the index term for the topic is, then look it up in the index, and then turn to any pages in the document other than the current one—quite a process! In literate programming documents (such as [13]), the footnotes on each page list the topics (e.g. variable names) and signpost other locations in the document where they are defined; there is no need for the reader to turn to a separate index.

In any approach, authors may make mistakes, such as misspelling a label name. Error reporting for signposts is thus very important. The main errors that can be detected are: multiple use of the same label, and referring to a label that has nowhere been defined. A different sort of error, that LaTeX could detect, but does not, is the case where an author writes material they plan to refer to elsewhere—so they label it, but they later fail to refer to it from anywhere. It is typically an oversight, if not a serious error, not to refer to a defined label.

An unfortunate error is to misspell the LaTeX names `ref` or `label`; for example, writing `\ ref{er}` (when `\ref{er}` was intended) which inserts the word refer in the document, refers to no reference and is an error only very diligent proof-reading (or specialist programs) will find [14]!

Indeed, authors can make many sorts of errors with signposts, many of which are hard to detect automatically, or, although detectable in principle may not be usefully reported for all sorts of documents. For example, an author may label a section, and refer to it from elsewhere in the document. Sometime later they edit the document, and move the referring text into the same section it refers to. This creates a section referring to itself. Saying 'See Section 27' inside Section 27 is almost certainly an error, but if the context of the signpost is something like, 'The definitions of these terms can be found at the beginning of this section, Section 27', then the author is clearly using the signpost to refer to a place in the document more specifically than the entire section, and the self-referential structure in itself indicates nothing about potential errors. Neither LaTeX nor any other automatic system can tell whether a self reference is an error or good style, yet it would clearly be useful if an author could be alerted to the potential problem.

4. THE HELPFUL SIGNPOST PROBLEM

Signposts cover a very wide range of uses, from the very structured table of contents, endnotes, footnotes, bibliographies and index (which often use their own specialist languages, like `makeindex` and BibTeX [9]), to the less structured use embedded in the author's narrative. This paper now defines 'helpful signposts,' concentrating on embedded signposts, and develops a solution for them in LaTeX documents.

Often an author of a large document will accidentally refer to a section as being earlier or later, trying to be helpful to the reader, but due to editing the section (or the referring section) has been moved out of order. For example, an author might write `see Section \ref{somewhere} earlier in this book`, but now the section comes later. This sort of error is not easily detected by LaTeX. A better solution would be for LaTeX to have a macro `\direction` so that the author could write

```
see Section \ref{somewhere}
\direction{somewhere} in this book
```

Here, LaTeX conditionally inserts either earlier or later as appropriate. Since LaTeX now generates the directional word itself, it will be correct—even if the document changes later. Hence, with appropriate automatic support, rather than detecting an error, the error can be avoided.

This 'helpful signpost' idea raises issues this paper will now address; to be done well, signposts depend on the deceptively simple notion of knowing where we are in the document when we write a signpost.

Because LaTeX is such a popular and widely used system, there are many packages available for signposting, including many for doing cross-references. See `http://tug.ctan.org`, the Comprehensive TeX Archive Network, for a definitive and up-to-date collection. In particular, Mittelbach has a substantial package, `varioref` [15], that covers similar ground to our solution, though additionally supporting numerous languages and styles such as Catalan, Croatian and Czech. `Varioref` is widely documented in LaTeX manuals such as [16]. However, none describes the rationale behind the concepts, which is part of the purpose of the present article.

An author can choose between `varioref` or the approach described here. `Varioref` is a large, very sophisticated package, which is well supported; its full documentation runs to many pages. In contrast, the approach described in this paper is a very small, simple package—a decision that we further justify in Section 7. A LaTeX author wishing to extend or provide

specialized signposting would probably prefer to start with this paper; certainly reading it will illuminate many otherwise obscure design decisions in `varioref`, should the author wish to modify its code. An author using a package other than LaTeX is helped directly by neither `varioref` nor the literal code here, but of course the abstract approach is still applicable.

5. DEVELOPING A SOLUTION TO THE HELPFUL SIGNPOST PROBLEM

My book, *Press On*, has 492 cross-references [17]—which are visualized in Fig. 1 (next page).[5] With its 510 pages, the book averages about one cross-reference per page.

Initially, when authoring *Press On* I used LaTeX's basic cross-referencing features, which allow the author to refer to sections easily. The author labels a section with a name of their choice, then uses the label to generate a reference to the actual section number. The author does not need to know the section number, and the cross-reference is generated automatically.

As we saw above, LaTeX makes it easy to generate text like

> See Section 12.3 on page 425

by writing

```
See Section~\ref{target}
on page~\pageref{target}
```

where `target` is the label of the target section; in this case it happens to be Section 12.3.

I realized readers would be helped more if I also provided the page number of the target section. Many cross-references get tedious for the reader: where are these sections the author is referring to? Providing page numbers solves that, but creates a new irritation when, as happens occasionally, the cross-reference is to something on the same page—for example, saying 'p. 425' could make the reader think they have to *turn* to p. 425, even when in fact they are already on it. Why not, then, program LaTeX so that it generates text such as

> See Section 12.3 on page 425
>
> See Section 12.3 on this page

as appropriate?

To choose between saying 'page 425' and 'this page' depending on the current page number seems to be a simple matter of writing a conditional, something like:

```
if currentPage = targetPage then
  say("this page");
else
    say("page " + targetPage);
```

[5] Figure 1 is from the book *Press On* [17]; since the figure is cross-referenced in the book, one vertex in the figure is the figure itself.

In TeX's macroprogramming notation this conditional would be written exactly, though perhaps not quite so clearly, as

```
\ifnum \thepage = \targetPage
    this page%
\else
    page \the\targetPage%
\fi
```

It may be tempting to write all further code in this paper in some seemingly lucid pseudo-programming language in an attempt to make this paper's arguments clearer and more accessible. However, it is not clear what real benefit this would bring, since translation of the original, working TeX examples on which this paper is based into some 'more intuitive' language could accidentally miss nuances, introduce errors or simply be misunderstood by the reader. Despite the (I think, unfortunate [18]) popularity of pseudo-programming languages for superficially clear presentations in papers, to use a psuedo-language, at least in the present context, would amount to cargo cult science [19]; that is, it would look like it works, but it would be misleading.

On the other hand some LaTeX- and TeX-specific details of algorithms here may be less applicable in other environments, but, if so, the details would need very careful explanation in this paper—and that would require a very careful definition of the pseudo-language. TeX, while acknowledging its limitations, is actually *very* well defined, and so we do not need to define its features here; TeX is also *very* widely used and understood, notwithstanding its opaqueness to readers unfamiliar with it. Nevertheless, readers of this paper must be assured that the complexity of the solutions developed *and that work exactly as shown* for TeX indicate the complexity of the problem; the complexities are not symptomatic of the baroque features of TeX!

To make the code above reusable, it can be defined as a macro, so that it can be used consistently in many places throughout a document. This may be done as follows:

```
\def \whereis#1{%
    \targetPage = 0\pageref{#1}\relax
    \ifnum \thepage = \targetPage
        this page%
    \else
        page \the\targetPage%
    \fi
}
```

The line `\targetPage = 0\pageref{#1}\relax` assigns to the variable `targetPage` the page number of where the LaTeX label `#1` was defined. If it happens that the label is undefined, `\pageref` will return question marks, which would cause an error message since TeX requires a number to be assigned to `targetPage`; this problem is avoided by the leading zero: whether `\pageref` returns '?' or a number,

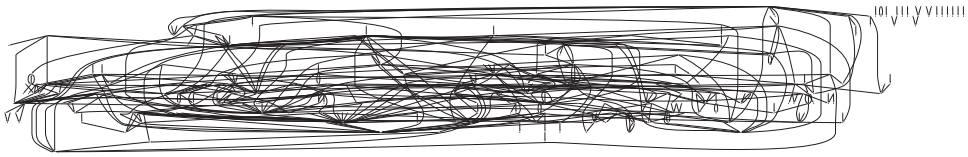

FIGURE 1. The signposts of a document form a directed graph. This figure shows a visualization of the signposts graph of the book *Press On* [17], where each vertex is taken to be a section containing at least one `\label` or `\ref`, and where arcs point from references to labels. For reasons of scale, we have not shown any cross-reference labels—which, in fact, would be better done interactively by mouse hover than by fixed text that would clutter the diagram. The graph was drawn by *Dot*.

a number (possibly zero) can be assigned to `\targetPage`. (The `\relax` ensures that anything following this assignment is not 'sucked' into and treated as part of the number.)

One might further define a new cross-referencing macro, along these lines:

```
\def \myref#1{\ref{#1} \whereis{#1}}
```

Every conventional use of LaTeX's original `\ref{`*name*`}` can now be replaced by our `\myref{`*name*`}`. (It is possible to redefine the original `\ref` instead, so that the author gets the advantages of the new `\myref` without having to change any text that uses it.)

Before we have a completely correct solution, however, there are problems to explore and overcome: with page numbers, Section 5.1; with macro expansion, Section 5.2; with local variables, Section 5.4; and with moving arguments, Section 5.3.

5.1. Problems with page numbers

Although we discuss TeX and LaTeX for concreteness, the issues raised relate to any authoring tools.

TeX typesets paragraphs and assembles them into pages *after* the paragraphs are typeset. Thus, code is evaluated before TeX knows which page it will be on. For example, if something occurs in a large paragraph, too large to fit on the current page, it may be carried forward on to the next page. Then the variable `\thepage` will be smaller when the code is evaluated than the page number of the page where the text 'this page' or whatever appears.

Since 'floating' objects such as top-of-the-page figures and tables, as well as footnotes, also contribute to page contents, it is not possible to know exactly how the variable `\thepage` will change between typesetting a paragraph and its actual appearance on a particular page.

The simplest solution to this problem is to be cautious: if `\thepage` is within, say, 5 pages of the target page we want to refer to, we could say nothing—the reader can be assumed to be fairly familiar with nearby pages. We need a reliable page number, but the only place in TeX that knows the page number is the so-called output routine, whose job it is to split text into pages and paginate. There is a built-in way to communicate between paragraphs being formatted and the output routine, and that is by using the `write` command. This command writes text to a file, and TeX arranges that the actual write is synchronized with flushing complete pages out of TeX—that is, when the page number has been defined. Very complex!

In fact, LaTeX's cross-referencing mechanism works in exactly this way. Of course, `\pageref` needs to know the correct page number for a label; thus, a reliable and as-it-were pre-debugged way of determining the current page number reliably within a paragraph is to write as follows:

```
This text occurs on
page \label{at}\pageref{at}.
```

This circumlocution will give us the correct page number for where it occurs, even if this sentence is in a paragraph that is taken forward to another page by the output routine.

Since every use of `\whereis` is potentially on a different page, we need to generate unique labels $\texttt{at}_n$ for each use. This is slightly harder than it seems, since the label does not get evaluated until the output routine has assembled the page, yet we need the label name (specifically, its unique identifier) to be evaluated on the right page, well before the output routine evaluates the label.

We use a counter `\uniquen` to generate unique names, `at\the\uniquen`: but we need `\the\uniquen` to be evaluated *here* but the page number `\thepage` to be evaluated inside the output routine when the page has been assembled and when we know what page it will be on. If we wrote simply `\label{at\the\uniquen}` the value of `\uniquen` would be evaluated in the output routine, possibly after further calls to `whereis` which would change the variable `uniquen`; it might have been moved to *later* pages!

Using TeX's `edef`, we can force evaluation where we want it. The `\edef`, as used below, ensures that `\pr` is defined to be `at` followed by the number `\the\uniquen` *at this point*, rather than evaluated later, when the number might have changed. Here is one way to do it:

```
\advance \uniquen by 1
\edef \pr{at\the\uniquen}%
\label{\pr}\pageref{\pr}
```

5.2. Problems with macro expansion

In a macro language like TEX, all variables have values that are text strings. We have already seen the 'trick' where a numeric value is obtained from a variable by prefixing it with `0` and following it by `\relax` (see §5). Unfortunately, the standard macro `\pageref` gives a textual value, not a numerical value; obviously TEX can cope with Arabic numerals like 23, but if the page in question uses roman numerals, then the code fails, as (unsurprisingly) TEX does not accept roman numerals as number literals. There are various solutions to this problem, but we will not cover them here.

5.3. Problems with 'moving arguments'

The caption of a figure may be copied to appear in the table of figures, which is typeset elsewhere in a document. Hence, LATEX calls captions (and some other features) *moving arguments*. If we need a figure caption that contains a cross-reference, the cross-reference will potentially be evaluated in (at least) two places: the page where the figure is typeset, and the page where the entry in the table of figures occurs. In fact, it is worse than this: the figure may be typeset on one page, but after typesetting it, it may turn out not to fit there, and so it will have to be carried forwards to a later page with sufficient space for it, *and* the caption will also have to be typeset a third time to fit into the table of figures, which is at the beginning of the document.

Similarly, paragraphs are typeset. Some may not fit on the current page (or column), so they are either split between pages or moved to the next page in their entirety. Again, this creates the problem that the text has to be typeset in order to determine its dimensions—a process which potentially involves defining labels and using signposts—then a layout decision has to be made, and then the paragraph may be reset on the same page, over two pages or on the next page, as the case may be. In the worst case, as with figure captions, the side effects of using signposts can change the values of the signposts.

There is no general solution that generates a correct signpost (e.g. a page number reference) automatically without some manual intervention from the author. This problem applies to moving arguments in all cases, simply because a caption *has* to be typeset in several different contexts: which one is the correct page that the author wishes to refer to?

Unfortunately, there is no space in this paper to discuss possible solutions to this problem and their trade-offs, other than to point out that the least that should be done is for a signposting system to report warnings when signposts continue to change despite the document otherwise being stable.

5.4. Problems with local variables

The macro `\whereis` uses a variable `\targetPage`. Best practice would have this be a local variable, so that its use in this macro does not conflict with its use anywhere else. Unfortunately, there is no way to declare local variables in TEX.

Further, discussion of this point is postponed to the Appendix, since it is complex and of only indirect relevance to this paper.

6. FINAL SOLUTION

The final solution generates cross-reference signposts in the following forms

> See Section 12.3 (p. 425)
>
> See Section 12.3 (this page)
>
> See Section 12.3 (next page)
>
> See Section 12.3 (previous page)
>
> See Section 12.3 (facing page)

Since most page references are more distant, the word 'page', which would otherwise be repetitious, is abbreviated 'p.' for the commonest case. Also, relevant for double-sided printing, the special cases when the target pages is the previous page on the left or the next page on the right, are both expressed as 'facing page'.

In the code below, the macro `\xrtrace` is used to insert the appropriate cross-reference form in the document and, simultaneously, to report on the TEX console what it is doing (using the diagnostic call `\typeout`)—using a distinctive `!!` prefix. This approach makes it easy for an author to review all cross-reference use.

```
\newcount \targetPage
\newcount \thisPage
\newcount \pageDelta
\newcount \uniquen

\uniquen = 0

\def \xrtrace#1#2#3{%
    \typeout{!! #2: #1 used on p.\the\thisPage,
             refers to p.\the\targetPage: #3}%
    #3%
}

\def \whereis#1{%
    \global\advance\uniquen by 1
    \edef\pr{at\the\uniquen}%
    \label{\pr}%
    \whereisrelative{#1}{\pr}%
}

\def \whereisrelative#1#2{%
    \targetPage = 0\pageref{#1}\relax
    \thisPage = 0\pageref{#2}\relax
```

```
    % we are on \thisPage,
    % and want to refer to \targetPage
    \pageDelta = \thisPage
    \advance \pageDelta by -\targetPage
    \ifnum \pageDelta = 0
        \xrtrace{#1}{#2}{(this~page)}%
    \else
        \ifnum \pageDelta = 1
            \ifodd \thisPage
                \xrtrace{#1}{#2}
                    {(facing~page)}%
            \else
                \xrtrace{#1}{#2}
                    {(previous~page)}%
            \fi
        \else
            \ifnum \pageDelta = -1
                \ifodd \thisPage
                    \xrtrace{#1}{#2}
                        {(next~page)}%
                \else
                    \xrtrace{#1}{#2}
                        {(facing~page)}%
                \fi
            \else
                \xrtrace{#1}{#2}
                    {(p.\,\the\targetPage)}%
            \fi
        \fi
    \fi
}
```

Note that all forms use unbreakable spaces (~), to avoid the confusion that '(facing' and 'page)' split across two pages might cause! With unbreakable spaces, LaTeX will format the phrase on a single page—and `\whereis` will ensure it refers to the right page correctly.

Note that occasionally the pagination of a document may depend sensitively on the exact result from `\whereis`. Consider the pathological case where it happens that `\whereis` saying '(next page)' is such long text that the referenced item moves further away, causing `\whereis` to say '(p. 345)' which being shorter brings the referenced text closer so that it returns to the next page again. Given the choices available to `\whereis`, there is no stable solution. This problem was explored in [12], where it was compared with the well-known NP-complete span-dependent instruction problem in assembly code. Thus, there is no good, general solution. Worse, there are ways to exacerbate the problem, for instance, by using Roman numerals—which vary far more in length than Arabic numerals! The only solution is to rewrite paragraphs—though, of course, review and rewriting is always required when writing well, and so this is not such a burden as it might at first appear [7]. Certainly, signposts being split across pages are tedious to read, and the problems they cause the typography system can be entirely avoided by rewriting to make the signposts easier for the human reader—which, after all, is the whole point.

The code above defines a macro `\whereis`, that is effectively a replacement for LaTeX's `\pageref`. It would be tempting to add the following definition to a LaTeX file:

```
\def \pageref#1{\whereis{#1}}
```

… except that LaTeX's `\pageref` only provides the page number of a cross-reference, not the word 'page' as well. To get the best results, the author must consider each use of `\ref` and `\pageref` individually. It would be ironic to do a global replace—either using an editor, or by using macro definitions (as above)—that made the English indifferent when the goal was to improve it!

6.1. Simple and necessary extensions

This paper is typeset in *The Computer Journal* with two column pages, and it probably helps the reader to refer not just to a page number but to the specific column on that page; thus, referring to the current Section 6.1 at this point in this document gets, unsurprisingly '(this page, this column)'. Yet this simple demonstration of signposts (in the preceding sentence) immediately suggests modifying `whereis` so that instead of saying something so pedantic it says 'above' or 'below' as appropriate when the label is on the same page and column as the signpost itself. The real issue, though, is why would an author want to say such things, and should we try to parametrize a general system for specialized uses? For example, the answer to the question depends on the following: if we are developing a document that has many figures, for instance, we might want to refer to their locations frequently and conveniently, and another, conflicting, answer is that we might best handle the issue by generating improved diagnostics from `whereis`—suggesting to the author that there appears to be an opportunity to finesse some automatically generated text. However, generalizing `whereis` to handle these ideas *as well as* what it already does goes beyond its remit. We will have more to say about adding features in §7 below.

To achieve references to columns as used in this paper, LaTeX's `\label` has to be redefined to save the current column (it already saves the section number and page number) and then `\ref` and `\pageref` have to be redefined to cope with the extension, and a new `\colref` introduced to recover the column from a label. Finally, our `whereis` macro needs to be extended so that it is aware whether a document is typeset in one or two columns—as there is no point being specific on columns if there is only ever one used throughout a document![6] The

[6] My own code provides a macro `pagewhereis`, which never refers to the column even when used in a two-column format document. An example of its use would be to refer to the two-column figure 1 (p. 9), as is also done twice elsewhere in this paper, in Section 5 and Section 8.2.

Appendix to this paper hints at the complexities of redefining built-in LaTeX commands, but the rest is routine programming.

Sometimes one might like to refer to the paragraph number, line number or some other more precise label; such extensions are handled in the same way.

Another issue is that `whereis` has a built-in view of left and right page numbering. Some documents do not follow these conventions, and some documents, once printed on one or two sides, or even 'four up' and so on, at the whim of the printer's paper handling, may lose the original document's concept of 'facing' pages: as currently conceived, then `whereis` would need revising to accommodate these possibilities.

6.2. Possible simplifications

LaTeX's original `\pageref` is very simple: it provides the user with a page number, like '5' and the author can easily decide whether to write this as 'p. 5', 'page 5', '5ff', simply by writing '`p.`' or whatever suits their needs in the context. In contrast, our `whereis` macro has more built-into it: it always brackets its result.

Rather than providing lots of formatting options for the author to overcome this inflexible style, it would perhaps be better to strip out the brackets. Unfortunately in English, the syntax of signposts varies too much: we write '*this* page' on the one hand and 'page *six*' on the other hand; putting the page reference within brackets solves that problem but forces the author to use the page reference in a context where a parenthesis is appropriate. It is not as flexible as `pageref`'s simpler approach, though its rigidity may usefully discourage an author from overuse of the fancy signpost.

7. MORE VARIATIONS—AND WHY NOT TO HAVE THEM

Given that we have provided a framework to solve the fundamental problems, to pursue more variations could now take us beyond the scope of this paper. The design balance is this: the more control the author is given over the construction of any specific cross-reference, inevitably the less reliable each particular cross-reference will be. Will every cross-reference take on the right form when some section or other moves? I prefer a more generic cross-reference form (as provided in full in the previous section), even at the cost of being more stylistically limited so that the *same* form can be used heavily in many contexts, and hence debugged to a high degree of confidence. It is important that signposts are clear and reliable; complex packages (as `varioref` certainly is) encourage authors to step beyond this voluntary boundary. As Tony Hoare has notably said [20],

> 'I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.'

Hoare also remarked,

> 'A feature which is omitted can always be added later, when its design and its implications are well understood. A feature which is included before it is fully understood can never be removed later.'

A third way, illustrated by our experience of signposting, is to make something so simple that everybody can fully understand it *and* that they can also fully understand its deficiencies. A misunderstood deficiency is as bad as a bug (and is far worse than a deficiency you understand), can anticipate and can work around ahead of time.

As if to vindicate Hoare, `varioref` documentation explicitly discusses its premature features that are, in hindsight, poorly designed, but which are now too late to correct [15, *op cit*]. `Varioref`'s highly parametrized sophistication certainly makes it harder to use reliably in some cases than the approach suggested here; authors will have to make a trade-off when choosing between the two approaches.

If generalizations were to be considered, the obvious approach would be to view the signpost as an object encapsulating the additional features. As conceived in the rest of this paper, the various uses of the reference have been programmed in a conventional way: there are different procedures for each use (`\ref`, `\whereis` and so on), and the procedures are separate and unrelated. Under the object orientation view, the object is the author's sign, and the various new features would be specializations of the same object: for example (and continuing in LaTeX syntax), one might write `\ref{name}{page}` for the page number of the signpost, or `\ref{name}{section}` for the section number of the signpost. The trouble with this re-factoring is that it encourages more generality (e.g. `\ref{name}{author}` or `\ref{name}{url}`) than can probably be implemented coherently. Fortunately, the author does not need a very general system, but one, that is, easy to use and is reliable—and understandable for the author—in all contexts. The more features there are, then, the less gracefulness and predictability can be sustained (unless the author is the programmer).

8. FURTHER USES OF SIGNPOSTS

8.1. Signpost scanners

Conventional documents are divided into a hierarchy of volumes, chapters, sections and subsections and 'quantized' into pages. These are familiar physical concepts, and indeed integral to the nature of paper printing and to books in particular. It is therefore natural to consider signposts as referring to these things, so that the user can find their way around them.

With computers, however, it is possible and perhaps desirable to extend the possibilities further. For example, a signpost might be part text (as usual, so it is familiar) and part barcode. If

the user prints the document, then holds up the barcode to a camera, the computer could locate the referenced part of some document. See ▌▌▌▌▌▌▌ (which is *of course* the DOI of this document, and hence takes the reader to the website where full bibliographic details can be obtained …).

8.2. Web searching ideas applied to single documents

There is an important sense in which the author's use of cross-referencing tries to signpost readers to key concepts within the narrative. Figure 1 (p. 9) shows the graph of explicit cross-references in the book *Press On* [17]. If we assume that a reader reads the book sequentially but, with some probability, follows signposts, and with some probability leaps to random places (e.g. if they look up something in the table of contents, they might then resume reading at any section of the book), then they can be modelled as a Markov process. We can therefore work out where readers are likely to be reading in the book, to the extent that the author's narrative and signposting is helpful to them. Under a wide range of such probabilities it turns out that a key concept in *Press On* is a section about cradle-to-cradle design. The author may disagree, but this should lead them to reconsider the signposts or signpost structures that they have so far defined in the document.

Such techniques also improve web searching [5], because they allow signposted pages to be ranked by importance, on the assumption that web linking (and other data that can be collected, such as user rankings) is related to relevance to the person searching for information. What is novel is to point out that the same techniques can be used to improve document searching *within* documents—the primary concern of this paper.

Automatic cross-referencing was discussed as long ago as 1989 [21], and perhaps even earlier. More recently, documents have been presented as small world networks [22]. Such techniques and perspectives could be used to either enrich the data available for organizing signposts in a document (or in a document reader tool), or to help an author check the structure of their own explicit signposting.

8.3. Structural uses for signposts

Signposts create a structure that helps both authors and readers navigate a document. They help the author communicate narrative structure to the reader and they also help the author manage the writing process, for instance, in tracking reminders, as was illustrated by the `xr` system [12] described in Section 3.

It is obvious that signposts can be supported by computer tools for the reader: this is the concept behind hypertext and web documents, as discussed in Section 2. However, tools can also support the author. For example, as the author introduces signposts, an authoring tool can create interactive outlines of the signposts, so that rearranging the signposts rearranges the document [23]—just as in a conventional outliner, rearranging the section headings rearranges the sections themselves. Such tools are very convenient and create huge benefits for the author: the author was going to use signposts anyway, and now finds that they support other organizational activities in structuring the document *with no further work from the author*.

For example, such a tool could provide an outline representing the structure of this document relative to all discussion of the cross-referencer `xr`, or any other signpost used in this document, as the author chooses. Some previous implementations and extensions of this idea are:

- To manage outlines and generate interactive tables of contents [24].
- To organize and structure the development of ideas and thoughts within a document. For example, [23] considers colouring signposts to simulate Edward de Bono's six thinking hats (which are different colours: blue, yellow, white, etc): certain colour sequences represent development of thought for particular rhetorical purposes, such as persuasion, exploration of alternatives and so on.

As it happens, both of the implementations cited above were in 1990s style; modern approaches would integrate them with web technologies (Ajax, RSS etc) and give the authors and readers further benefits that were unforeseeable in the 1990s.

9. CONCLUSIONS

A very satisfactory signposting solution for both reader and author can be programmed in LaTeX to provide helpful cross-references. The bad news, though, is that programming LaTeX to achieve this seemingly trivial but useful outcome is a major undertaking; this paper exhibits the solution to one particular signposting problem—other authors with different requirements will have to face and solve their own problems. TeX and LaTeX do *not* provide a good platform for doing this, largely because they are based on macroprocessing, though it is not obvious that any similar system can in principle do better: some problems (such as knowing the page number) are fundamental to the complexity of document preparation.

ACKNOWLEDGEMENTS

George Buchanan and Leslie Lamport gave many useful comments. Anonymous referees helped improve the presentation enormously.

FUNDING

The author was a Royal Society-Leverhulme Trust Senior Research Fellow, and gratefully acknowledges this support for the research reported here.

REFERENCES

[1] Bornat, R. and Thimbleby, H. (1989) The Life and Times of Ded, Display Editor. In Long, J.B. and Whitefield, A. (eds), *Cognitive Ergonomics and Human Computer Interaction*, pp. 225–255. Cambridge University Press, Cambridge, UK.

[2] Smith, D.C., Irby, C., Kimball, R. and Harslem, E. (1982) The Star User Interface: An Overview. *AFIPS'82, National Computer Conf.*, Las Vegas, NV, USA, pp. 515–528. ACM, New York, NY, USA.

[3] Eisenstein, E.L. (1983) *The Printing Revolution in Early Modern Europe*. Cambridge University Press, Cambridge, UK.

[4] Buchanan, G. and Pearson, J. (2008) Improving Placeholders in Digital Documents. *Proc. 12th European Conf. Research and Advanced Technology for Digital Libraries*, Aarhus, Denmark, pp. 1–12, Springer, Berlin.

[5] Langville, A.N. and Meyer, C.D. (2006) *Google's PageRank and Beyond*. Princeton University Press, Princeton.

[6] Sterne, L. (1759) *The Life and Opinions of Tristram Shandy, Gentleman*. Republished in Sterne, L. (1857) *The Works of Laurence Sterne*, Derby & Jackson, New York, NY, USA.

[7] Knuth, D.E. (1992) *The TEX Book*. Addison-Wesley, Boston, MA.

[8] Brady, F. (1970) Tristram Shandy: Sexuality, morality, and sensibility. *Eighteenth Century Stud.*, **4**, 41–56.

[9] Lamport, L. (1994) *LATEX: A Document Preparation System* (2nd edn). Addison-Wesley, Reading, MA.

[10] Lie, H.W. and Bos, B. (2005) *Cascading Style Sheets* (3rd edn). Addison-Wesley, Boston, MA.

[11] Thimbleby, H. (1986) Experiences with literate programming using CWEB (a variant of Knuth's WEB). *Comput. J.*, **29**, 201–211.

[12] Thimbleby, H. (1992) An Author's Cross–Referencer. In Holt, P.O. and Williams, N. (eds), *Computers and Writing—State of the Art*, pp. 90–108. Intellect Books.

[13] Knuth, D.E. (1986) *TEX the Program*. Addison-Wesley, Boston, MA.

[14] Thimbleby, H. (1991) Low Tech LATEX. In Sharples, M. (ed.), *Proc. Conf. Computers & Writing*, Brighton, England, pp. 124–130.

[15] Mittlebach, F. (2003) *The* `varioref` *package*. http://tug.ctan.org.

[16] Kopka, H. and Daly, P.W. (2004) *A Guide to LATEX*, (4the edn). Addison Wesley, Boston, MA.

[17] Thimbleby, H. (2007) *Press On*. MIT Press, Boston, MA.

[18] Thimbleby, H. (2003) Explaining code for publication. *Softw.—Pract. Exp.*, **33**, 975–1001.

[19] Feynman, R.P. (1985) 1974 Caltech Commencement Address. In Hutchings, E. (ed.), *Surely You're Joking, Mr. Feynman!* W W Norton. Reproduced in Feynman, R.P. and Leighton, R. (contributor).

[20] Hoare, C.A.R. (1981) The emperor's old clothes. *Commun. ACM*, **24**, 75–83.

[21] Salton, G. and Buckley, C. (1989) On the Automatic Generation of Content Links in Hypertext. Technical Report UMI Order Number: TR89-993.

[22] Matsuo, Y.O., Ohsawa, Y. and Ishizuka, M. (2001) A Document as a Small World. *JSAI Workshop*, Lecture Notes in AI, Matsue, Japan, pp. 444–448. Springer, Berlin.

[23] Thimbleby, H. (1994) Designing interfaces for problem solving, with application to hypertext and creative writing. *AI Soc.*, **8**, 29–44.

[24] Thimbleby, H. (1997) Gentler: a tool for systematic web authoring. *Int. J. Human Comput. Stud.*, **47**, 139–168.

[25] Knuth, D.E. (1999) *Digital Typography*. Center for the Study of Language and Information, Stanford, CA.

[26] Hoenig, A. (1998) *TEX Unbound*. Oxford University Press, Oxford, UK.

[27] Knuth, D.E. (1979) *TEX and METAFONT*. Digital Press, Bedford, MA, USA.

APPENDIX

A.1. On local names in TEX

TEX was designed for typography and not as a general-purpose programming language. When the inevitability of creating a *programmable* typesetting system was forced on Knuth, he retained TEX's basic features as a macroprocessing language [25, p. 648 ff] and expanded them. Further discussion of TEX can be found in [26], and in Knuth's own excellent discussions, [7, 27].

Macroprocessing languages are very flexible for text processing, particularly as they allow the processing commands to be embedded into the text itself, which makes macroprocessing particularly suitable for authors to control the typesetting of their documents. One disadvantage of macroprocessing is that layout and whitespace that may be used by a programmer to clarify program code may have a side-effect on the operation of the macroprocessing: this is the reason for the otherwise obscure `%` marks in the code in this paper—they comment-out whitespace that would otherwise be significant. A more serious problem is that, unfortunately, TEX provides no name scoping: name clashes are unavoidable, and when they do occur it is very difficult to solve the problems that arise. Unlike use or misuse of `%`, bad name scoping often creates very obscure problems, not least because the impact of the problems may be felt far away in the document from their causes. Often scoping problems are discovered by independent programmers using libraries that mysteriously break their code; the causes of their problems like deep inside code that they are unfamiliar with (and sometimes it may be proprietary and they will not have direct access to the source code).

As the bulk of this paper shows, document processing in general requires sophisticated programming, and name scoping is certainly one aspect of that. On the other hand, the popularity of TEX is testament to the wisdom of Knuth's design decisions; ironically, now that we know better what features a successful typesetting system should have, the very popularity of TEX itself argues against developing a new system from the revised requirements—it would not be possible to be both backwards compatible and to solve the problems for the existing user base! The awkward problems described in this Appendix, which were uncovered while programming signposts for non-trivial

documents, however, are a fundamental problem for TEX (and for LATEX, which is implemented in TEX) but they could be solved in a backwards compatible way if TEX was to be revised.

Blocks in programming languages are used to control the scope of names (i.e. the association of a name and the value it denotes within a block). In languages like C, Java and, as it happens, TEX too, the symbols { and } are used to enclose blocks. Generally, a name defined within a block is not available or known about in any other block, except in blocks entirely enclosed textually within the defining block.

In a block-structured language like Java, the following code defines two different and independent variables that happen to use the same name, x:

```
{ int x; ... } { int x; ... }
```

Equal freedom to use the same name also applies when blocks are nested, as in the following block structure:

```
{ int x;
  ...
  { int x;
    ...
  }
  ...
}
```

So, as a simple example, the following code would print 5, because the final use of x refers to the first declaration of x and its immediately following assignment; the x in the inner block that gets assigned 10 is a different variable.

```
{ int x;
  x = 5
  { int x;
    x = 10;
  }
  print x;
}
```

Although this code looks contrived, the important point is that each block can use any names it likes—including x—and its behaviour is independent of other blocks, its use of x neither depends on them nor influences them. Technically, we can replace the second x consistently with any other name throughout the block, and the final result will be exactly the same (this property is called alpha or α equivalence in lambda calculus). It is important to emphasize that this is a standard and familiar concept in programming, precisely because it is so useful.

Without blocks, every name would have to be different and programmers would quickly run out of sensible names to use. Programmers would also have to check that the names they do want to use do not clash with their or other programmers' choices of names. In an environment that allows libraries of code to be used, code written by independent programmers, such checking is impossible—one might have a working program, decide to take advantage of a library and then see the program break because of a name clash. Thus, blocks (or modules, and other similar programming language features) ensure that names intended to be private remain local concerns.

The name-scoping concept is very general, useful and widely used. Thus, in the real world, I am called Harold, but a town does not break down if I go there and find another Harold! Our use of names remains local to each person called Harold. Of course, if I wish to imitate somebody else that may be possible by getting appropriate permissions (or by being devious), but the standard behaviour is that private use of names is a matter for each separate person.

In a language for processing documents, as with any other language, name scoping has the same benefits: authors should be able to introduce local names in parts of a document without worrying about clashes with uses of names elsewhere, or values of variables being changed in remote code—or parts of the document—that ought to be unrelated.

A very simple example of the use of a local name in a document might be a counter (say, \itemCount) to help generate a numbered list of items, such as (1) red, (2) green and (3) blue, and so on, that are numbered automatically. If we need another item later, the list carries on like (4) this, and the counter \itemCount is incremented ready for the next item.

Here is a simple pull-quote.
(1) mauve,
(2) pink.

A section in a document might include a figure, table or pull-quote (indeed, as shown nearby), written by a different author (or by the same author, but at a different time—perhaps long ago and forgotten).[7] If the insertion itself has another list of items, we would surely want those items to be numbered correctly, most likely starting from (1) within the inclusion. However, after the inclusion, the document should return back to the original sequence of item numbers, for instance in our pull-quote example it should continue from (5) but not from (3), which is where the pull-quote left its *own* counter.

It is important to note that whereas we used item counters in this simple example, an author might want to do practically anything, not just count items. They may want to do things like numbering sections, or they may wish to do complex things requiring sophisticated programming such as sorting index entries, recognizing when runs of numbers may be elided (as in typesetting '2–5' instead of '2, 3, 4, 5') and more. They may even want to program signposts.

In general, we want the authors of an insertion, or any nested material in a document, to be able to program and specify

[7] *Tristram Shandy*, cited earlier in Section 1, is an example of a book that embeds all manner of material with its own rather discursive narrative structure. A multi-authored encyclopædia would be a more conventional example of a document requiring merging of independently written material.

almost any document feature, and to be able to do so safely without affecting the larger document of which it only forms a part. Of course, we should expect almost anything, not just simple sequence numbering. Certainly, we want authors to be able to introduce and use names for their own purposes (in the example, we used an item counter) and not accidentally clash them with any names already in use for other purposes. In a programming language, such safe contexts are *blocks* (or more generally *modules*). Unfortunately, despite the seductive appearances of the examples above, the naming system of TEX does *not* respect block scoping rules in general.

In our very simple example of a pull-quote, above, the font and point size were changed for the quote itself, and they were correctly restored automatically by TEX for the rest of the document: the *values* of the font variables were scoped correctly. However, TEX is far more interesting. If we define our own name for the item counter, a new definition of the name in the insertion will corrupt the name outside. In fact, if we continue with the same list of items, their numbers would now continue from (3), not from (5) as we might have wanted. (All item numbers used in these examples were actually generated by TEX itself, by using a single item counter name.)

A very important advantage of local definition blocks is that different parts of large programs, or different libraries, can use the same names without risk of their meanings clashing. TEX has libraries: for example, an author may use the TEX code introduced in this paper for signposts, but they would like to benefit from the signposting without other parts of their document breaking down because our signposting uses some private name that clashes with some other name already used for a different purpose. It is important, then, that names are not aliased with each other and thus given different meanings by different people (or even the same person on different occasions)—especially when convenient names like `t` might be used repeatedly, or (perhaps worse) diligent programmers are driven to use obscure names in the hope that nobody else would be silly enough to use them too. This defensive strategy stops programmers using a consistent naming scheme, and generally makes programs unreadable and hard to maintain.

TEX beguilingly has what looks like a sensible approach to block structure, but it does not work as expected, as we shall see.

We could illustrate TEX's problems with dimensions, tokens, registers and other nameable TEX objects. However, it is simplest to use simple numeric counters. First, then, consider the following TEX code that prints 9 0 9, which is what we would expect from block structure behaving correctly.

```
\count0 = 9 % assign 9 to variable count0
\the\count0 % print its value, namely 9
{  % enter a new block
   \count0 = 0 % assign 0 to variable
               % also named count0
   \the\count0 % print its value, 0
}
% after block
\the\count0 % print count0's value
            % -- it will print 9
```

Here, we see that the variable name `count0` refers to different things, depending on which block it is used in.[8] Evidently, what `count0` refers to respects the block structure.

Now, naming variables by numbers (e.g. `count0`, `count1`, `count2`, etc) is not as helpful as using names, which will be more convenient and mnemonic. Hence, TEX provides a command to create named identifiers. The command `\newcount \fred` defines `\fred` to be a counter. Rather than writing `count124`, `count99` or whatever, you write `sectionNumber` or `itemCount`, which then refer to the appropriate underlying counter register.

The TEX declaration `newcount` works by binding names to specific counters. If we write

```
\newcount \itemCount
```

this would bind `itemCount` with an as-yet unused counter, `count124` perhaps. Specifically, *The TEXbook* (p. 121) [*op. cit.*] says that `newcount` 'dedicates a `\count` register to a special purpose'.

Now `newcount` is an ordinary macro itself. If it defines `itemCount` to be `count124` in the obvious way, TEX's block structure would ensure that the definition of `itemCount` would be local to the block that is inside the definition of `newcount`, but that is not the block where we want to use the name `itemCount`!

TEX provides a mechanism for making definitions global, so they are known everywhere, and not just within the block of the definition. In order to fix the `newcount` problem, the solution chosen by TEX is that `newcount` creates a global definition of `itemCount`, so `itemCount` will mean `count124` (or whatever) *everywhere*, not just within the block that is the code of the `newcount` definition itself.

Global definitions allow features such as `newcount` to work correctly in simple cases: `newcount` defines a name that is globally bound to a particular count variable, such as `count124`. Unfortunately, this approach means that names defined by `newcount` *do not* respect block structure because the names it defines are globally defined.

Consider the following TEX, which is structurally exactly the same as the last example, but uses TEX's `newcount` to define names instead of using raw counters. Thus, instead of using an explicit register name (above, we used `count0`)

[8] Technically, `\count` is a control sequence `count` followed by a parameter `0`. Thus `\count 0` (with a space) is the same as `\count0`; we can also write `\count \n` to use a variable as an index. We could consider `count` to be an array; we could consider `\count0` to be a name; or `0` to be an index, *etc*. Such low level detail makes no difference to the high-level issue of name scoping being discussed in this Appendix.

we will use a name `itemCount` that TEX allocates to some register.

Now, surprisingly, the code prints the values 9 0 0, not 9 0 9 as before. The inner, second, definition of `itemCount` is global and 'escapes', so the second use of `itemCount`, outside the block, refers to the inner definition not to the first, as happened before. In other words, the TEX code inside the block has accidentally destroyed a variable outside of the block: the original meaning of `itemCount` is no longer available.

```
\newcount \itemCount
    % declare a new variable, itemCount
\itemCount = 9
    % assign 9 to variable itemCount
\the\itemCount
    % print its value, namely 9

{  % enter a new block,
   % hopefully defining a local itemCount
   \newcount \itemCount
   % assign 0 to variable itemCount
   \itemCount = 0
   % and print its value, namely 0
   \the\itemCount
}

\the\itemCount  % print itemCount's value
                % it will be 0,
                % not 9 as before
```

No error or diagnostic is created by the unfortunate clashing of names; a TEX user just gets unexpected results they may not notice. TEX allows names to refer to different things. Here, the `itemCount` in the inner block usurps the `itemCount` in the outer block. In general, *this behaviour means that TEX programs cannot define "local" names without risking breaking other code.*

The next example, below, defines `t` to be a macro, then defines the same name in an *inner* block to be a counter. This causes an error, detected only after the inner block in the example `t` is no longer a macro, because `newcount` redefined it globally inside the body of `anotherMacro`.

```
\def \t{Stuff}
The value of t is '\t' (i.e., Stuff)
\def \anotherMacro{\newcount\t ....}
\anotherMacro
This might be expected to show
the value of the macro: '\t'
```

TEX implements `newcount` by binding names to successive counters in turn. This approach does not respect the block structure. It means for instance that in the last example that every time the macro `anotherMacro` is used the 'local' variable `t` is bound to a new counter and old counters never get recycled! Eventually, TEX will run out of counters.

To summarize: the definition of TEX *says* it respects block structure, a design feature that if it worked would have had many advantages for the user, particularly for writing complex documents and for reusing other people's code or libraries (such as libraries for signposting). In practice, however, it is not possible to define local names safely. You cannot define, say, `t` or any other name in an inner block or a library without risking it messing up or being messed up by something else.

Despite the problems, TEX will presumably have to stay as it is, as this preserves the meaning of all existing TEX material—but this leaves the name difficulties remaining. Ironically, since the problem is to do with names, one cannot even introduce a corrected `newcount` (and all the related functions), without risking breaking any TEX that uses whatever name is chosen for the corrected version, and the more one makes the new name obscure and unlikely to clash with anything, the less likely it is to be brief or mnemonic!

A.2. Implications for using TEX and LATEX

Our code for signposting in LATEX or TEX, as given above in Section 6, relies on local names (`targetPage`, `thisPage`, `pageDelta`, `uniquen` and `xrtrace`) and it relies on other parts of the document being processed not changing their values unexpectedly. This Appendix has established that there is no way, within TEX, to *ensure* that name clashes and unexpected behaviour do not occur. In contrast, conventional programming languages would provide modules, blocks, packages or other such features to control names and ensure that either name clashes were innocuous (i.e. various uses of the same local names are quite independent of each other) or at least clashes were treated as errors and reported by the language processing system (typically the compiler, but here it would have to be TEX itself).

The only feasible solution is to rename all 'private' variables within the new signposting package with some funny names that ordinary users of TEX will never use, and hence name clashes will not happen. LATEX's own solution to this problem is to rename local variables by introducing an `@` sign into their name. In normal use, an `@` sign cannot be part of a name, as `@` is defined not to be a letter, so LATEX's approach uses names that nobody else can use, and thus ensures there are no name clashes.

For example, LATEX actually redefines `\newcount` to use its own private memory allocation counter system, based on a macro called `alloc@`. LATEX's `\newcount` works like TEX's original `\newcount`; it does not fix the name scoping problems this Appendix discusses. The point of the redefinition is that it ensures a user thinks `\newcount` works correctly, but avoids any clashes with LATEX's internal use of the original `\newcount` scheme for its own purposes.

An ordinary user of LaTeX cannot even use the name `alloc@`, since syntactically the `@` is not part of a name an ordinary user can create[9]—if they tried using `\alloc@`, this would instead create a call to the macro `alloc` with a parameter `@`. LaTeX would simply complain that `\alloc` was undefined and then print the @ on its own!

In order to use a name including an `@` sign, TeX has to be told to treat `@` as a letter—this essentially changes the name space, as if entering a 'LaTeX block' where names are local to LaTeX. Given the complexity of TeX naming, this is a reasonable solution, as at least is separates hardcore LaTeX programming (where we *assume* programmers know what they are doing, and somehow have checked for name clashes) from 'ordinary' programming that the rest of us do. At least the rest of us do not need to check for possible name clashes with LaTeX's private names, as no ordinary programming—as used in this paper—can have name clashes with the core of LaTeX. But it is not a very good solution, as it creates only two namespaces: LaTeX and the document. As we have argued, documents themselves need block structure. TeX cannot deliver it (except by losing almost everything we use and value TeX for).

A.3. Local names in signposting

Local names have advantages because separate parts of programs can be written independently, and they do not interfere with each other. Similar advantages would apply to local signpost names. Thus, if a book is made from chapters, the chapters may have local signposts that make sense, but when the chapters are concatenated name clashes may arise. An obvious example of a local signpost name would be `conclusions`, useful, for instance, if some or all chapters in a book had a conclusions section the author wishes to refer to.

LaTeX, `varioref` and the approach defined in this paper provide no support for local names; however, `xr` [12] does (and reference [12] discusses some of the advantages of doing so). Local names are a major advantage in multi-author documents.

[9]The macro `\makeatletter` turns `@` into a letter. Once this is called, any user can create names that clash with LaTeX's, at least until `\makeatother` is called when `@` returns to its normal, safe, role. Essentially, then, these macros (if used properly!) create a block where LaTeX's namespace can be used deliberately, but also at the cost of risking name clashes from within LaTeX and other packages using the same technique. Plain TeX does the same sort of thing; see [7, Appendix B].