Writing Documentation in Lean with Verso

David Thrane Christiansen

Contents

Introduction

Verso is a tool for writing about Lean. Or, rather, it is a framework for constructing such tools, together with concrete tools that use this framework. Technical writing can take many forms, including but not limited to:

- Reference manuals
- Tutorials
- Web pages
- Academic papers

All of these genres have common concerns, such as displaying Lean code, including tests to prevent bit-rot of the text, and linking to other resources. However, they are also very different. Some have a very linear structure, while others combine date-based content with an unordered set of pages. Some should generate highly interactive output, while others should generate PDFs that can be turned into published papers books. Verso consists of the following components:

Markup language Verso's markup language is a simplified variant of Markdown. It is also an alternative concrete syntax for Lean itself, so Verso documents are just Lean files. Just as TeX, Sphinx, and Scribble allow their languages to be extended using their own programming languages, Verso's markup language is extensible. Define a Lean function at the top of a file, and use it in the text of that very same file.

Extensible document structure All Verso documents can contain a set of common elements , such as paragraphs, emphasized text, or images. They also share a hierarchical structure of sections and subsections. These types are extensible by individual genres.

Elaboration and rendering framework Verso provides a shared paradigm for converting text written by an author into readable output. Different genres will produce different output formats, but they don't need to reinvent the wheel in order to resolve cross-references, and they can benefit from shared libraries for producing output in various formats.

Cross-reference management Verso includes a common paradigm for representing the documented items, and a format for sharing cross-reference databases between genres that emit HTML, which enables links and cross-references to be automatically inserted and maintained.

Lean rendering Verso includes facilities for elaborating and displaying Lean code in documents. In HTML output, this code is rendered with toggleable proof states, hovers, and hyperlinks. It's also highlighted accurately, which is impossible with regexp-based highlighting due to Lean's syntactic extensibility.The `SubVerso` helper library allows Verso documents to process Lean code written in any version of Lean, starting with `4.0.0`. This makes it possible to write a document that compares and contrasts versions, or to decouple upgrades to the Lean version used in a project from the Lean version used in the document that describes it.

Utility libraries Verso includes utility libraries that can be used by genres to provide features such as full-text search of HTML content. These libraries have no additional build-time dependencies, avoiding the complications of staying up to date with multiple library ecosystems at once.

1 Genres

Documentation comes in many forms, and no one system is suitable for representing all of them. The needs of software documentation writers are not the same as the needs of textbook authors, researchers writing papers, bloggers, or poets. Thus, Verso supports multiple *genres* , each of which consists of:

- A global view of a document's structure, whether it be a document with subsections, a collection of interrelated documents such as a web site, or a single file of text
- A representation of cross-cutting state such as cross-references to figures, index entries, and named theorems
- Additions to the structure of the document - for instance, the blog genre supports the inclusion of raw HTML, and the manual genre supports grouping multiple top-level blocks into a single logical paragraph
- Procedures for resolving cross references and rendering the document to one or more output formats

All genres use the same markup syntax , and they can share extensions to the markup language that don't rely on incompatible document structure additions. Mixing incompatible features results in an ordinary Lean type error.

2 Verso Markup

Lean's documentation markup language is a close relative of Markdown, but it's not identical to it.

2.1 Design Principles

Syntax errors Fail fast rather than producing unexpected output or having complicated rules.

Reduce lookahead Parsing should succeed or fail as locally as possible.

Extensibility There should be dedicated features for compositionally adding new kinds of content, rather than relying on a collection of ad-hoc textual subformats.

Assume Unicode Lean users are used to entering Unicode directly and have good tools for it, so there's no need to support alternative textual syntaxes for characters not on keyboards such as em dashes or typographical quotes.

Markdown compatibility Users benefit from existing muscle memory and familiarity when it doesn't lead to violations of the other principles.

2.2 Syntax

Like Markdown, Lean's markup has three primary syntactic categories:

Document structure Headers, footnote definitions, and named links give greater structure to a document. They may not be nested inside of blocks.

Block elements The main organization of written text, including paragraphs, lists, and quotations. Some blocks may be nested: for example, lists may contain other lists.

Inline elements The ordinary content of written text, such as text itself, bold or emphasized text, and hyperlinks.

Document Structure

Documents are organized into *parts* . A part is a logical division of the content of the document, such as a chapter, section, subsection, or volume. Parts may have associated metadata, such as authors, publication dates, internal identifiers for cross-referencing, or desired URLs; the specific metadata associated with a part is determined by the document's genre .

A part contains a sequence of blocks followed by a sequence of sub-parts, either of which may be empty.

A part is introduced with a *header* . Headers consist of one or more hash marks (#) at the beginning of a line followed by a sequence of inlines. The number of hash marks indicates the nesting of a header, and headers with more hash marks indicate parts at a lower level. A lower-level header introduces a sub-part of the preceding header, while a header at a level at least as high as the preceding header terminates that part. In other words, header levels induce a tree structure on the document.

Headers must be well-nested: the first header in a document must have exactly one hash mark, and all subsequent headers may have at most one more hash mark than the preceding header. This is a *syntactic* requirement: regardless of whether a Verso file contains the text of a whole book, a chapter, or just a single section, its outermost header must be introduced with a single hash mark. Headers indicate the logical nesting structure of the document, rather than the headers to be chosen when rendering the document to output formats such as HTML.

Metadata may be associated with a header by following it with a *metadata block* . Metadata blocks begin and end with %%%, and they contain any syntax that would be acceptable in a Lean structure initializer.

Syntax

```
# Top-Level Header
```

Result

```
<h1>Top-Level Header</h1>
```

Markup Example 1: Header

Syntax

```
a b c
```

Result

```
<p> a b c </p>
```

Markup Example 2: Blah

Block Syntax

Paragraphs are undecorated: any sequence of inlines that is not another block is a paragraph. Paragraphs continue until a *blank line* (that is, a line containing only whitespace) or another block begins:

Syntax

```
This is one paragraph.
Even though this sentence is on a
new line, the paragraph continues.

This is a new paragraph.
* This list stopped the paragraph.
* As in Markdown and SGML, lists
  are not part of paragraphs.
```

Result

```
<p>
  This is one paragraph. Even
  though this sentence is on a new
  line, the paragraph continues.
</p>

<p> This is a new paragraph. </p>

<ul>
  <li>
    <p>
      This list stopped the
      paragraph.
    </p>
  </li>
  <li>
    <p>
      As in Markdown and SGML,
      lists   are not part of
      paragraphs.
    </p>
  </li>
</ul>
```

Markup Example 3: Paragraphs

Lists

There are three kinds of lists:

Unordered lists Unordered lists indicate that the order of the items in the list is not of primary importance. They correspond to `<ul>` in HTML or `\begin{itemize}` in LaTeX.

Ordered lists Unordered lists indicate that the order of the items in the list is important. They correspond to `<ol>` in HTML or `\begin{enumerate}` in LaTeX.

Description lists Description lists associate a term with more information. This very list is a description list. They correspond to `<dl>` in HTML or `\begin{description}` in LaTeX.

A line that begins with zero or more spaces followed by a `*`, `-`, or `+` starts an item of an unordered list. This character is called the *list item indicator* . Subsequent items are part of the same list if they use the same indicator character and have the same indentation. Any subsequent blocks whose first character is indented further than the indicator are part of the item itself; items may contain multiple blocks, or even other lists.

A line that starts with zero or more spaces, followed by one or more digits and then either a period (e.g. `1.`) or a closing parenthesis (`1)`), begins an

Syntax

```
* A list with two items
* They both start in column 0
```

Result

```
<ul>
  <li>
    <p> A list with two items </p>
  </li>
  <li>
    <p>
      They both start in column 0
    </p>
  </li>
</ul>
```

Markup Example 4: Unordered Lists

Syntax

```
 * A list with two items
 * They both start in column 1
```

Result

```
<ul>
  <li>
    <p> A list with two items </p>
  </li>
  <li>
    <p>
      They both start in column 1
    </p>
  </li>
</ul>
```

Markup Example 5: Indented Unordered Lists

Syntax

```
 * Two lists
 + They have different indicators.
```

Result

```
<ul>
  <li> <p> Two lists </p> </li>
</ul>

<ul>
  <li>
    <p>
      They have different
      indicators.
    </p>
  </li>
</ul>
```

Markup Example 6: List Indicators

Syntax

```
 * A list with one element
* Another list, due to different
  indentation
```

Result

```
<ul>
  <li>
    <p>
      A list with one element
    </p>
  </li>
</ul>

<ul>
  <li>
    <p>
      Another list, due to
      different   indentation
    </p>
  </li>
</ul>
```

Markup Example 7: List Indentation Sensitivity

Syntax

```
* A list with one item.
  It contains this paragraph

  * And this other sub-list

  And this paragraph
```

Result

```
<ul>
  <li>
    <p>
      A list with one item.   It
      contains this paragraph
    </p><ul>
      <li>
        <p>
          And this other sub-list
        </p>
      </li>
    </ul><p>
      And this paragraph
    </p>
  </li>
</ul>
```

Markup Example 8: Sub-Lists

item of an ordered list. Like unordered lists, ordered lists are sensitive to both indentation and indicator characters.

Syntax

```
1. First, write a number.
2. Then, an indicator (`.` or `)`)
```

Result

```
<ol>
  <li>
    <p> First, write a number. </p>
  </li>
  <li>
    <p>
      Then, an indicator
      (<code>"."</code> or
      <code>")"</code>)
    </p>
  </li>
</ol>
```

Markup Example 9: Ordered Lists

Syntax

```
1. Two lists
2) They have different indicators.
```

Result

```
<ol>
  <li> <p> Two lists </p> </li>
</ol>

<ol>
  <li>
    <p>
      They have different
      indicators.
    </p>
  </li>
</ol>
```

Markup Example 10: Ordered Lists and Indicators

Description lists consist of a sequence of description items that have the same indentation. A description item is a line that starts with zero or more spaces, followed by a colon, followed by a sequence of inlines and a blank line. After the blank line, there must be one or more blocks with indentation greater than the colon.

Quotes

Quotation blocks show that a sequence of blocks were spoken by someone other than the author. They consist of a line that starts with zero or more spaces, followed by a greater-than sign (>), followed by a sequence of blocks that are indented more than the greater-than sign.

Syntax

```
: Item 1

  Description of item 1

: Item 2

  Description of item 2
```

Result

```
<dl>
    <dt>  Item 1 </dt>
  <dd>
    <p> Description of item 1 </p>
  </dd>

    <dt>  Item 2 </dt>
  <dd>
    <p> Description of item 2 </p>
  </dd>
   </dl>
```

Markup Example 11: Description Lists

Syntax

```
It is said that:
> Quotations are excellent.

  This paragraph is part of the
  quotation.

 So is this one.

But not this one.
```

Result

```
<p> It is said that: </p>

<blockquote>
  <p>
    Quotations are excellent.
  </p>
  <p>
    This paragraph is part of the
    quotation.
  </p>
  <p> So is this one. </p>
</blockquote>

<p> But not this one. </p>
```

Markup Example 12: Quotations

Code Blocks

Code blocks begin with three or more back-ticks at the start of a line (they may be preceded by spaces). This is referred to as a *fence* . They may optionally have a name and a sequence of arguments after the back-ticks. The code block continues until a line that contains only the same number of back-ticks at the same indentation. Every line in the code block must be at least as indented as the fence, and it denotes the string that results from removing the indentation from each line. Code blocks may not contain a sequence of back-ticks that is at least as long as the fence; to add more back-ticks to the code block, the fence must be made larger.

When a code block has a name, then the name is resolved in the current Lean namespace and used to select an implementation. The chapter on Verso markup extensions has more details on this process.

Syntax

````
```
This is a code block
with two lines
```
````

Result

```
<codeblock>
  1|This is a code block⏎
  2|with two lines⏎
</codeblock>
```

Markup Example 13: Code Blocks

Syntax

````
```lean
-- This code block is named
-- `lean`. It was called with
-- no arguments.
def x := 5
```
````

Result

```
<lean >
  1|-- This code block is named⏎
  2|-- `lean`. It was called with⏎
  3|-- no arguments.⏎
  4|def x := 5⏎
</lean>
```

Markup Example 14: Code Blocks as Extensions

Syntax

````
  ```lean
  -- This indented code block
  -- denotes the de-indented
  -- string
  def x := 5
  ```
````

Result

```
<lean >
  1|-- This indented code block⏎
  2|-- denotes the de-indented⏎
  3|-- string⏎
  4|def x := 5⏎
</lean>
```

Markup Example 15: Indented Code Blocks

Syntax

````
```lean (error := true)
-- This code block is named
-- `lean`, called with the
-- named argument `error`
-- set to `true`.
def x : String := 5
```
````

Result

```
<lean  error="true">
  1|-- This code block is named⏎
  2|-- `lean`, called with the⏎
  3|-- named argument `error`⏎
  4|-- set to `true`.⏎
  5|def x : String := 5⏎
</lean>
```

Markup Example 16: Code Blocks and Arguments

Directives

A *directive* is a kind of block with no meaning other than that assigned by extensions, akin to a custom environment in LaTeX or a `<div>` tag in HTML. Directives begin with three or more colons and a name with zero or more arguments, and end with the same number of colons at the same indentation on a line by themselves. They may contain any number of blocks, which must be indented at least as much as the colons. Nested directives must begin and end with strictly fewer colons than the surrounding directives.

The chapter on Verso markup extensions describes the processing of directives in more detail.

This is an empty directive:

Syntax

```
:::nothing
:::
```

Result

```
<nothing >  </nothing>
```

Markup Example 17: Directives

This directive contains a directive and a paragraph:

Syntax

```
::::outer
:::inner
Hello
:::
This is a paragraph
::::
```

Result

```
<outer >
  <inner > <p> Hello </p> </inner>
  <p> This is a paragraph </p>
</outer>
```

Markup Example 18: Nested Directives

Commands

A line that consists of only a set of curly braces that contain a name and zero or more arguments is a *command* . The name is used to select an implementation for the command, which is then invoked during elaboration. The chapter on Verso markup extensions has more details on this process.

Syntax

```
{include 0 MyChapter}
```

Result

```
<include  0 MyChapter/>
```

Markup Example 19: Commands

Inline Syntax

Emphasis is written with underscores:

Syntax

```
Here's some _emphasized_ text
```

Result

```
<p>
  Here's some
  <emph> emphasized </emph> text
</p>
```

Markup Example 20: Emphasis

Emphasis can be nested by using more underscores for the outer emphasis:

Syntax

```
Here's some __emphasized text
with _extra_ emphasis__ inside
```

Result

```
<p>
  Here's some
  <emph>
    emphasized text with
    <emph> extra </emph> emphasis
  </emph>
  inside
</p>
```

Markup Example 21: Nested Emphasis

Strong emphasis (bold) is written with asterisks:

Syntax

```
Here's some *bold* text
```

Result

```
<p>
  Here's some <bold>bold</bold>
  text
</p>
```

Markup Example 22: Strong Emphasis (Bold)

Hyperlinks consist of the link text in square brackets followed by the target in parentheses:

Syntax

```
[Lean](https://lean-lang.org)
```

Result

```
<p>
  <a href="https://lean-lang.org">
    Lean
  </a>
</p>
```

Markup Example 23: Links

Link targets may also be named:

Syntax

```
The [Lean website][lean]

[lean]: https://lean-lang.org
```

Result

```
<p>
  The
  <a href="(value of «lean»)">Lean
  website</a>
</p>

where
  «lean» := https://lean-lang.org
```

Markup Example 24: Named Link Targets

Literal code elements are written in back-ticks. The same number of back-ticks that opens a code element must be used to close it, and the code element may not contain a sequence of back-ticks that's at least as long as its opener and closer.

Syntax

```
The definition of `main`
```

Result

```
<p>
  The definition of
  <code>"main"</code>
</p>
```

Markup Example 25: Literal Code

As a special case, code inlines that being and end with a space denote the string that results from removing one leading and trailing space. This makes it possible to represent values that begin or end with back-ticks:

Syntax

```
`` `quotedName ``
```

Result

```
<p> <code>"`quotedName"</code> </p>
```

Markup Example 26: Spaces in Code

or with spaces:

Syntax

```
``  one space  ``
```

Result

```
<p> <code>" one space "</code> </p>
```

Markup Example 27: Multiple Spaces in Code

Images require both alternative text and an address for the image:

TeX math can be included using a single or double dollar sign followed by code. Two dollar signs results in display-mode math, so $`\sum_{i=0}^{10} i`

Syntax

```
![Alt text](image.png)
```

Result

```
<p>
  <img
    src="image.png"
    alt="Alt text"/>
</p>
```

Markup Example 28: Images

Syntax

```
![Alternative text][image]

[image]: image.png
```

Result

```
<p>
  <img
    src="value of «image»"
    alt="Alternative text"/>
</p>

where «image» := image.png
```

Markup Example 29: Named Image Addresses

results in $\sum_{i=0}^{10} i$ while `$$`\sum_{i=0}^{10} i`` results in:

$$\sum_{i=0}^{10} i$$

A *role* indicates that subsequent inlines have a special meaning. Roles can be used to trigger special handling of code (e.g. elaboration), register definitions and uses of technical terms, add marginal notes, and much more. They correspond to custom commands in LaTeX. A role consists of curly braces that contain a name and arguments, all on the same line, followed immediately either by a self-delimiting inline or by square brackets that contain zero or more inlines.

An inline is *self-delimiting* if it has distinct start and end tokens that make its ending position clear. Emphasis, strong emphasis, code, links, images, math, and roles are self-delimiting.

This role contains multiple inlines using square brackets:

Syntax

```
{ref "chapter-tag"}[the chapter on
$`\frac{1}{2}`-powered syntax]
```

Result

```
<p>
  <ref "chapter-tag">
    the chapter on
    <math contents="\\frac{1}{2}"/>
    -powered syntax
  </ref>
</p>
```

Markup Example 30: Roles With Explicit Scope

This one takes a single inline code element without needing square brackets:

Syntax

```
{lean}`2 + f 4`
```

Result

```
<p>
  <lean >
    <code>"2 + f 4"</code>
  </lean>
</p>
```

Markup Example 31: Single-Argument Roles

2.3 Differences from Markdown

This is a quick "cheat sheet" for those who are used to Markdown, documenting the differences.

Syntax Errors

While Markdown includes a set of precedence rules to govern the meaning of mismatched delimiters (such as in `what _is *bold_ or emph*?`), these are syntax errors in Lean's markup. Similarly, Markdown specifies that unmatched delimiters (such as * or _) should be included as characters, while Lean's markup requires explicit escaping of delimiters.

This is based on the principle that, for long-form technical writing, it's better to catch typos while writing than while reviewing the text later.

Reduced Lookahead

In Markdown, whether `[this][here]` is a link depends on whether `here` is defined as a link reference target somewhere in the document. In Lean's markup, it is always a link, and it is an error if `here` is not defined as a link target.

Header Nesting

In Lean's markup, every document already has a title, so there's no need to use the highest level header (#) to specify one. Additionally, all documents are required to use # for their top-level header, ## for the next level, and so forth, because a single file may represent a section, a chapter, or even a whole book. Authors should not need to maintain a global mapping from header levels to document structures, so Lean's markup automatically assigns these based on the structure of the document.

Genre-Specific Extensions

Markdown has no standard way for specific tools or styles of writing to express domain- or genre -specific concepts. Lean's markup provides standard syntaxes to use for this purpose, enabling compositional extensions.

Fewer Unused Features

Markdown has a number of features that are rarely used in practice. They have been removed from Verso to reduce surprises while using it and make documents more predictable. This includes:

- the distinction between tight and loose lists,
- four-space indentation to create code blocks,
- Setext-style headers, indicated with underlines instead of leading hash marks (#),
- hard line break syntax,
- and HTML entities and character references

Other Markdown features don't make sense for non-HTML output, and can be implemented by a genre using code blocks or directives. They have also been removed from Verso. In particular, this includes HTML blocks, raw HTML and thematic breaks.

Finally, some Markdown features are used by a minority of authors, and make sense in all backends, but were not deemed worth the complexity budget. In particular, this includes auto-links.

3 Building Documents

Verso is a general framework for implementing documentation tools, and this flexibility means that the details of the process described in this section may differ for specific tools.

A Verso document passes through the following steps on its way from its author to its readers:

1. The author writes the document's text in Verso's markup language , which is parsed to Lean's own `Syntax` type
2. The document is elaborated to a representation as a Lean data structure
3. The resulting Lean code is compiled to an executable
4. When run, the executable resolves cross-references and computes other globally-scoped metadata in a step referred to as the traversal pass
5. Next, the executable generates the output

3.1 Elaboration

During the elaboration process, Verso's markup language is converted into its internal representation as a Lean inductive type. When Verso's elaborator encounters an extension point , it consults an internal table to select an implementation of the extension, and delegates to it. Other syntax is translated into the appropriate constructors of Verso's data.

All Verso documents are parameterized by their genre :

structure

`Verso.Doc.Genre : Type 1`

A genre is a kind of document that can be written with Verso.A genre is primarily defined by its extensions to the Verso framework, provided in this type. Additionally, each genre should provide a `main` function that is responsible for the traversal pass and for generating output.

Constructor

`Verso.Doc.Genre.mk`

Fields

`PartMetadata` : `Type`
The metadata that may be associated with each `Part` (e.g. author, publication date, cross-referencing identifier).

`Block` : `Type`
Additional block-level values for documents written in the genre.

`Inline` : `Type`
Additional inline-level values for documents written in the genre.

`TraverseContext` : `Type`
The reader-style data used in the genre's traversal pass. Instances of `TraversePart` and `TraverseBlock` for a genre specify how this is updated while traversing parts and blocks, respectively.

`TraverseState` : `Type`
The mutable state used in the genre's traversal pass.

Each document consists of a `Part`. The part's title is the title of the entire document.

def

```
Verso.Doc.Part (genre : Doc.Genre) : Type
```

A logical division of a document.

`Part`s contain `Block`s:

def

```
Verso.Doc.Block (genre : Doc.Genre) : Type
```

Block-level content in a document.

`Block`s contain `Inline`s:

def

```
Verso.Doc.Inline (genre : Doc.Genre) : Type
```

Inline content that is part of the text flow.

The `metadata` field of `Part` typically gets its value from a metadata block written by the author, though it may be assigned more information during traversal. The `Block.other` and `Inline.other` constructors typically result from elaborating extension points .

3.2 Compilation

After elaboration, the document is compiled into an executable program. Each genre provides a `main` function that will carry out the remainder of the steps. Usually, this `main` function can be applied to the part that represents the whole document; however, genres that don't have a strict linear order (such as the website genre) will provide their own means of configuring the document's layout. The `main` function typically also takes configuration parameters both in the code and on the command line, such as which output formats to generate or customizations to the generated output.

3.3 Traversal

Because they are Lean values, Verso documents adhere to the structure of Lean programs in general. In particular, Lean doesn't support cyclic import graphs. It's common, however, for technical writing to include cyclic references; two sections that describe different aspects of something will frequently refer to one another. Similarly, a bibliography that's generated from a database needs a global view of a document to include only those works which are, in fact, cited.

The *traversal* phase occurs at runtime, before generating output. During the traversal phase, the document is repeatedly traversed from beginning to end, and metadata is accumulated into a table. The document may also be modified during traversal; this allows the title of a section to be inserted into a cross-reference. This traversal is repeated until the resulting document and metadata tables are not modified; it fails if a set number of passes are executed that result in modifications each time.

Verso provides a general-purpose traversal mechanism for `Part`, `Block`, and `Inline` that genres may use. `Genre.TraverseState` contains the genre-specific information that's accumulated during traversal, while `Genre.TraverseContext` provides a means of tracking the surrounding document context. To use this framework, genres should define instances of `Traverse`, which specifies the traversal of a genre's custom elements. Additionally, instances of `GenrePart` and `GenreBlock` specify how traversal keeps track of the current position in a document.

type class

```
Verso.Doc.Traverse (g : Doc.Genre) (m : outParam (Type → Type)) : Type 1
```

Genre-specific traversal.The traversal pass is where cross-references are resolved. The traversal pass repeatedly applies a genre-specific stateful computation until a fixed point is reached, both with respect to the state and the document. Traversal may update the state or rewrite parts of the document.The methods `part`, `block`, and `inline` pro-

vide effects to be carried out before traversing the given level of the AST, and `part` allows the part's metadata to be updated.`genrePart` is carried out after `part`. It allows genre-specific rewriting of the entire part based on genre-specific metadata. This is typically used to construct a table of contents or permalinks, but it can in principle arbitrarily rewrite the part. `inPart` is used to locally transform the genre's traversal context along the lines of `withReader`, and can be used to keep track of e.g. the current position in the table of contents.`genreBlock` and `genreInline` are invoked when traversal encounters `Block.other` and `Inline.other`. It may rewrite them, or have state effects.

Instance Constructor

`Verso.Doc.Traverse.mk`

Methods

`part : [MonadReader g.TraverseContext m] → [MonadState g.TraverseState m] → Doc.Part g → m (Option g.PartMetadata)`

The effects carried out before traversing a `Part`.

`block : [MonadReader g.TraverseContext m] → [MonadState g.TraverseState m] → Doc.Block g → m Unit`

The effects carried out before traversing a `Block`.

`inline : [MonadReader g.TraverseContext m] → [MonadState g.TraverseState m] → Doc.Inline g → m Unit`

The effects carried out before traversing an `Inline`.

`genrePart : [MonadReader g.TraverseContext m] → [MonadState g.TraverseState m] → g.PartMetadata → Doc.Part g → m (Option (Doc.Part g))`

Operations carried out after `part`, when a part has metadata. It allows genre-specific rewriting of the entire part based on genre-specific metadata. This is typically used to construct a table of contents or permalinks, but it can in principle arbitrarily rewrite the part. If it returns `none`, then no rewrite is performed.

`genreBlock : [MonadReader g.TraverseContext m] → [MonadState g.TraverseState m] → g.Block → Array (Doc.Block g) → m (Option (Doc.Block g))`

The traversal of genre-specific block values. If it returns `none`, then no rewrite is performed.

`genreInline : [MonadReader g.TraverseContext m] → [MonadState g.TraverseState m] → g.Inline → Array (Doc.Inline g) → m (Option (Doc.Inline g))`

The traversal of genre-specific inline values. If it returns `none`, then no rewrite is performed.

type class

`Verso.Doc.TraversePart (g : Doc.Genre) : Type`

Specifies how to modify the context while traversing the contents of a given part.

Instance Constructor

`Verso.Doc.TraversePart.mk`

Methods

`inPart : Doc.Part g → g.TraverseContext → g.TraverseContext`

How to modify the context while traversing the contents of a given part. This is applied after `part` and `genrePart` have rewritten the text, if applicable.It is also used during HTML generation.

type class

`Verso.Doc.TraverseBlock (g : Doc.Genre) : Type`

Specifies how to modify the context while traversing the contents of a given block.

Instance Constructor

`Verso.Doc.TraverseBlock.mk`

Methods

`inBlock : Doc.Block g → g.TraverseContext → g.TraverseContext`

How to modify the context while traversing a given block.It is also used during HTML generation.

3.4 Output Generation

Following traversal, the readable version of the document is generated. This may be in any format; each genre defines its supported formats.

Additionally, genres that emit HTML may generate a serialized version of their cross-reference database. This can be used to automatically maintain the links that implement cross-references between Verso documents: if content moves, rebuilding the linking document is sufficient to fix the link.

4 Extensions

Verso's markup language features four extension points:

- Roles
- Directives
- Code blocks
- Commands

These can be used to extend Verso to support new documentation features.

4.1 Syntax

All four extension points share a common syntax. They are invoked by name, with a sequence of arguments. These arguments may be positional or by name, and their values may be identifiers, string literals, or numbers. Boolean flags may be passed by preceding their name with `-` or `+` for `false` or `true`, respectively.

In this example, the directive `syntax` is invoked with the positional argument `term` and the named argument `title` set to `"Example"`. The flag `check` is set to `false`. It contains a descriptive paragraph and the code block `grammar`, which is invoked with no arguments:

````
:::syntax term (title := example) -check
This is an example grammar:
```grammar
term ::= term "<+-+>" term
```
:::
````

More formally, an invocation of an extension should match this grammar:

```
CALL := IDENT ARG*
ARG := VAL | "(" IDENT ":=" VAL ")" | "+" IDENT | "-" IDENT
VAL := IDENT | STRING | NUM
```

A `CALL` may occur after an opening fence on a code block. It is mandatory after the opening colons of a directive, in the opening curly braces of a role, or in a command.

4.2 Elaborating Extensions

Each kind of extension has a table that maps names to expanders. An *expander* converts Verso's syntax to Lean terms. When the elaborator encounters a code block, role, directive, or command invocation, it resolves the name and looks up an expander in the table. Expanders are attempted until one of them either throws an error or succeeds. Expanders use the monad `DocElabM`, which is an extension of Lean's term elaboration monad `TermElabM` with document-specific features. Expanders first parse their arguments into a suitable configuration type, typically via a `FromArgs` instance, after which they return Lean syntax.

There are two ways to associate an expander with a name: the `@[code_block]`, `@[role]`, `@[directive]`, and `@[block_command]` attributes (preferred) or the `@[code_block_expander]`, `@[role_expander]`, and `@[directive_expander]` attributes. Using the former attributes results in an expander that invokes the argument parser automatically, and they enable Verso to automatically compute usage information from a `FromArgs` instance. The latter are lower-level, and require manual parsing of arguments.

Parsing Arguments

This grammar is fairly restrictive, so each extension is responsible for parsing their arguments in order to afford sufficient flexibility. Arguments are parsed via instances of `FromArgs`:

type class

```
Verso.ArgParse.FromArgs (α : Type) (m : Type → Type) :
Type 1
```

A canonical way to convert a sequence of Verso arguments into a given type.

Instance Constructor

`Verso.ArgParse.FromArgs.mk`

Methods

`fromArgs` : `ArgParse m α`

Converts a sequence of arguments into a value.

Implementations of `FromArgs.fromArgs` specify parsers written using `ArgParse`:

inductive type

`Verso.ArgParse (m : Type → Type) : Type → Type 1`

A parser for arguments in some underlying monad.

Constructors

| `fail {m : Type → Type} {α : Type} (stx? : Option Lean.Syntax) (message? : Option ArgParse.SigDoc) : ArgParse m α`
Fails with the provided error message.

| `pure {m : Type → Type} {α : Type} (val : α) : ArgParse m α`
Returns a value without parsing any arguments.

| `lift {m : Type → Type} {α : Type} (desc : String) (act : m α) : ArgParse m α`
Provides an argument value by lifting an action from the underlying monad.

| `positional {m : Type → Type} {α : Type} (nameHint : Lean.Name) (val : ArgParse.ValDesc m α) (doc? : Option ArgParse.SigDoc := none) : ArgParse m α`
Matches a positional argument.

| `named {m : Type → Type} {α : Type} (name : Lean.Name) (val : ArgParse.ValDesc m α) (optional : Bool) (doc? : Option ArgParse.SigDoc := none) : ArgParse m (if optional = true then Option α else α)`
Matches an argument with the provided name.

| `anyNamed {m : Type → Type} {α : Type} (name : Lean.Name) (val : ArgParse.ValDesc m α) (doc? : Option ArgParse.SigDoc := none) : ArgParse m (Lean.Ident × α)`
Matches any named argument.

| `flag {m : Type → Type} (name : Lean.Name) (default : Bool) (doc? : Option ArgParse.SigDoc := none) : ArgParse m Bool`
Matches a flag with the provided name.

| `flagM {m : Type → Type} (name : Lean.Name) (default : m Bool) (doc? : Option ArgParse.SigDoc := none) : ArgParse m Bool`
Matches a flag with the provided name, deriving a default value from the monad

| `done {m : Type → Type} : ArgParse m Unit`
No further arguments are allowed.

| `orElse {m : Type → Type} {α : Type} (p1 : ArgParse m α) (p2 : Unit → ArgParse m α) : ArgParse m α`
Error recovery.

| `seq {m : Type → Type} {α β : Type} (p1 : ArgParse m (α → β)) (p2 : Unit → ArgParse m α) : ArgParse m β`

The sequencing operation of an applicative functor.
| `many {m : Type → Type} {α : Type} : ArgParse m α → ArgParse m (List α)`
Zero or more repetitions.
| `remaining {m : Type → Type} : ArgParse m (Array Doc.Arg)`
Returns all remaining arguments. This is useful for consuming some, then forwarding the rest.

Individual argument values are matched using `ValDesc`:

structure

`Verso.ArgParse.ValDesc (m : Type → Type) (α : Type) : Type`

A means of transforming a Verso argument value into a given type.
Constructor
`Verso.ArgParse.ValDesc.mk`
Fields
`description : ArgParse.SigDoc`
How should this argument be documented in automatically-generated signatures?
`signature : ArgParse.CanMatch`
Which of the three kinds of values can match this argument?
`get : Doc.ArgVal → m α`
How to transform the value into the given type.

A canonical value description for a Lean type can be registered via an instance of `FromArgVal`:

type class

`Verso.ArgParse.FromArgVal (α : Type) (m : Type → Type) : Type`

A canonical way to convert a Verso argument into a given type.
Instance Constructor
`Verso.ArgParse.FromArgVal.mk`
Methods
`fromArgVal : ArgParse.ValDesc m α`
A canonical way to convert a Verso argument into a given type.

In addition to the constructors of `ArgParse`, the `Applicative` and `Functor` instances are important, as well as the following helpers:

def

```
Verso.ArgParse.namedD {α : Type} {m : Type → Type} (name
: Lean.Name) (val : ArgParse.ValDesc m α) (default : α) :
ArgParse m α
```

Matches an argument with the provided name. If the argument is not present, `default` is returned.

def

```
Verso.ArgParse.positional' {α : Type} {m : Type → Type}
[ArgParse.FromArgVal α m] (nameHint : Lean.Name) (doc? :
Option ArgParse.SigDoc := none) : ArgParse m α
```

Matches a positional argument using the type's `FromArgVal` instance.

def

```
Verso.ArgParse.named' {α : Type} {m : Type → Type}
[ArgParse.FromArgVal α m] (name : Lean.Name) (optional :
Bool) (doc? : Option ArgParse.SigDoc := none) : ArgParse
m (if optional = true then Option α else α)
```

Matches a named argument using the type's `FromArgVal` instance.

def

```
Verso.ArgParse.namedD' {α : Type} {m : Type → Type}
[ArgParse.FromArgVal α m] (name : Lean.Name) (default :
α) : ArgParse m α
```

Matches an argument with the provided name using the type's `FromArgVal` instance. If the argument is not present, `default` is returned.

5 Output Formats

Verso provides genre authors with tools for generating HTML and TeX code via embedded languages that reduce the syntactic overhead of constructing ASTs. These libraries may also be used by authors of extensions to the `Manual` genre, who need to define how each new element should be rendered to each supported backend.

5.1 HTML

Verso's `Html` type represents HTML documents. They are typically produced using an embedded DSL that is available when the namespace `Verso.Output.Html` is opened.

inductive type

`Verso.Output.Html : Type`

A representation of HTML, used to render Verso to the web.

Constructors

| `text (escape : Bool) (string : String) : Html`
  Textual content. If `escape` is `true`, then characters such as `'&'` are escaped to entities such as `"&"` during rendering.
| `tag (name : String) (attrs : Array (String × String)) (contents : Html) : Html`
  A tag with the given name and attributes.
| `seq (contents : Array Html) : Html`
  A sequence of HTML values.

def

`Verso.Output.Html.empty : Html`

The empty HTML document.

def

```
Verso.Output.Html.fromArray (htmls : Array Html) : Html
```

Converts an array of HTML elements into a single element by appending them.This is equivalent to using `Html.seq`, but may result a more compact representation.

def

```
Verso.Output.Html.fromList (htmls : List Html) : Html
```

Converts a list of HTML elements into a single element by appending them.This is equivalent to using `Html.seq` on the corresponding array, but may result in a more compact representation.

def

```
Verso.Output.Html.append : Html → Html → Html
```

Appends two HTML documents.

opaque

```
Verso.Output.Html.visitM.{u_1} {m : Type → Type u_1}
[Monad m] (text : Bool → String → m (Option Html) := fun x
x_1 => pure none) (tag : String → Array (String × String) →
Html → m (Option Html) := fun x x_1 x_2 => pure none) (seq :
Array Html → m (Option Html) := fun x => pure none) (html :
Html) : m Html
```

Visit the entire tree, applying rewrites in some monad. Return `none` to signal that no rewrites are to be performed.

opaque

```
Verso.Output.Html.format : Html → Std.Format
```

Converts HTML into a pretty-printer document. This is useful for debugging, but it does not preserve whitespace around preformatted content and scripts.

opaque

```
Verso.Output.Html.asString (html : Html) (indent : Nat
:= 0) (breakLines : Bool := true) : String
```

Converts HTML into a string that's suitable for sending to browsers, but is also readable.

HTML documents are written in double curly braces, in a syntax very much like HTML itself. The differences are:

- Double curly braces escape back to Lean. This can be done for HTML elements, attribute values, or whole sets of attributes.
- Text content is written as Lean string literals to facilitate precise control over whitespace.
- Interpolated Lean strings (with `s!`) may be used in any context that expects a string.

For example, this definition creates a `<ul>` list:

```
open Verso.Output.Html

def mkList (xs : List Html) : Html :=
  {{ <ul> {{ xs.map ({{<li>{{·}}</li>}}) }} </ul>}}

#eval mkList ["A", {{<emph>"B"</emph>}}, "C"]
  |>.asString
  |> IO.println


<ul>
  <li>
    A</li>
  <li>
    <emph>B</emph></li>
  <li>
    C</li>
  </ul>
```

5.2 TeX

Verso's `TeX` type represents LaTeX documents. They are typically produced using an embedded DSL that is available when the namespace `Verso.Output.TeX` is opened.

inductive type

`Verso.Output.TeX : Type`

TeX output

Constructors

`| text (string : String) : TeX`
Text to be shown in the document, escaped as needed.

| `raw (string : String) : TeX`
  Raw TeX code to be included without escaping.
| `command (name : String) (optArgs args : Array TeX) : TeX`
  A LaTeX command, with the provided optional and mandatory arguments (in square and curly brackets, respectively)
| `environment (name : String) (optArgs args content : Array TeX) : TeX`
  A LaTeX environment, with the provided optional and mandatory arguments (in square and curly brackets, respectively)
| `paragraphBreak : TeX`
  A paragraph break, rendered to TeX as a blank line
| `seq (contents : Array TeX) : TeX`
  Concatenation of TeX

def

`Verso.Output.TeX.empty : TeX`

The empty TeX document

opaque

`Verso.Output.TeX.asString (doc : TeX) : String`

Converts a TeX document to a string to be processed by LaTeX

TeX documents are written in `\TeX{...}`, in a syntax very much like LaTeX itself. The differences are:

- `\Lean{...}` escapes back to Lean, expecting a value of type `TeX`.
- Text content is written as Lean string literals to facilitate precise control over whitespace.
- Interpolated Lean strings (with `s!`) may be used in any context that expects a string.

For example, this definition creates a bulleted list list:

```
open Verso.Output.TeX

def mkList (xs : List TeX) : TeX :=
  \TeX{
    \begin{itemize}
      \Lean{xs.map (\TeX{\item " " \Lean{·} "\n"})}
    \end{itemize}
  }
```

```
#eval mkList ["A", \TeX{\emph{"B"}}, "C"]
  |>.asString
  |> IO.println


\begin{itemize}
\item A
\item \emph{B}
\item C

\end{itemize}
```

6 Websites

Verso's website genre is a static site generator. It contains two Verso `Genre`s: `Page` and `Post`, which are identical except for their metadata:

def

```
Verso.Genre.Blog.Page : Genre
```

An ordinary web page that is not a blog post.

def

```
Verso.Genre.Blog.Post : Genre
```

A blog post.

Both feature the same set of extensions:

inductive type

```
Verso.Genre.Blog.BlockExt : Type
```

The additional blocks available in pages and posts.

Constructors

| `highlightedCode (opts : Blog.CodeOpts) (highlighted : SubVerso.Highlighting.Highlighted) : Blog.BlockExt`
  Highlighted Lean code.
| `message (summarize : Bool) (msg : SubVerso.Highlighting.Highlighted.Message) (expandTraces : List Lean.Name) : Blog.BlockExt`
  Highlighted Lean messages.
| `lexedText (content : Blog.LexedText) : Blog.BlockExt`
  Lexed text, to be displayed with highlighted syntax.
| `htmlDiv (classes : String) : Blog.BlockExt`
  When rendered, the content is wrapped in a `<div>` with the given classes.
| `htmlWrapper (tag : String) (attributes : Array (String × String)) : Blog.BlockExt`

When rendered, the content is wrapped in the specified HTML tag.

| `htmlDetails (classes : String) (summary : Output.Html) : Blog.BlockExt`
When rendered, the content is wrapped in a `<details>` tag with the given summary.

| `blob (html : Output.Html) : Blog.BlockExt`
A blob of HTML. The contents are discarded.

| `component (name : Lean.Name) (data : Lean.Json) : Blog.BlockExt`
A reference to a component.

| `docstring (indent : Nat) (declName? : Option Lean.Name) : Blog.BlockExt`
A Lean docstring that was indented `indent` spaces in the original source.

| `docstringSection (level : Nat) : Blog.BlockExt`
A section in a Lean docstring.Lean docstrings may contain nested headers, but they are not sections of the document as a whole. The contents of a docstring section are expected to be a paragraph that contains the section's title, followed by the actual content.

inductive type

`Verso.Genre.Blog.InlineExt : Type`

The additional inline elements available in pages and posts.

Constructors

| `highlightedCode (opts : Blog.CodeOpts) (highlighted : SubVerso.Highlighting.Highlighted) : Blog.InlineExt`
Highlighted Lean code.

| `message (msg : SubVerso.Highlighting.Highlighted.Message) (expandTraces : List Lean.Name) : Blog.InlineExt`
Highlighted Lean messages.

| `lexedText (content : Blog.LexedText) : Blog.InlineExt`
Lexed text, to be displayed with highlighted syntax.

| `customHighlight (highlighted : SubVerso.Highlighting.Highlighted) : Blog.InlineExt`
Highlighted code that is not necessarily from Lean.

| `label (name : Lean.Name) : Blog.InlineExt`
A label to serve as a cross-reference target.

| `ref (name : Lean.Name) : Blog.InlineExt`
A reference to a label.

| `pageref (name : Lean.Name) (id? : Option String) : Blog.InlineExt`
A reference to a page or post's internal name as a Lean value, to be shown as a link. If `id?` is `some X`, then the link is suffixed with `#X`.

| `htmlSpan (classes : String) : Blog.InlineExt`
  An HTML span element with the given classes.
| `blob (html : Output.Html) : Blog.InlineExt`
  A blob of HTML. The contents are discarded.
| `component (name : Lean.Name) (data : Lean.Json) : Blog.InlineExt`
  A reference to a component.

However, their metadata are different:

structure

`Verso.Genre.Blog.Page.Meta : Type`

The metadata used for non-blog-post pages
Constructor
`Verso.Genre.Blog.Page.Meta.mk`
Fields
`showInNav : Bool`
Whether to hide this page/part from navigation entries
`htmlId : Option String`
The HTML ID to assign to the header

structure

`Verso.Genre.Blog.Post.Meta : Type`

The metadata used for blog posts
Constructor
`Verso.Genre.Blog.Post.Meta.mk`
Fields
`date : Blog.Date`
The post's date. By default, this is used in the URL as well as included in the content.
`authors : List String`
The authors of the post
`categories : List Post.Category`
The categories in which to include the post
`draft : Bool`
If `true`, the post is not rendered by default
`htmlId : Option String`
The HTML ID to assign to the header

6.1 Generating a Site

Blogs should have an executable that invokes `blogMain` on the appropriate site and theme , forwarding on command-line arguments. It is responsible for traversing the site and generating the HTML.

def

```
Verso.Genre.Blog.blogMain (theme : Blog.Theme)
(site : Blog.Site) (linkTargets : Code.LinkTargets
Blog.TraverseContext := { }) (options : List
String) (components : Blog.Components := by exact
%registered_components) (header : String :=
Output.Html.doctype) : IO UInt32
```

Generates the HTML for `site`.Parameters:

- `theme` is the theme used to render content.
- `site` is the site to be generated.
- `options` are the command-line options provided by a user.

Optional parameters:

- `linkTargets` specifies how to create hyperlinks from Lean code to further documentation. By default, no links are generated.
- `components` contains the implementation of the components. This is automatically filled out from a table.
- `header` is emitted prior to each HTML document. By default, it produces a `<doctype>`, but it can be overridden to integrate with other static site generators.

6.2 Configuring a Site

The URL layout of a site is specified via a `Site`:

inductive type

```
Verso.Genre.Blog.Site : Type
```

A specification of the layout of an entire site
Constructors
| `page (id : Lean.Name) (text : Part Page) (contents : Array Blog.Dir) : Blog.Site`
The root of the site is a page
| `blog (id : Lean.Name) (text : Part Page) (contents :`

`Array Blog.BlogPost) : Blog.Site`
The root of the site is a blog with its associated posts

inductive type

`Verso.Genre.Blog.Dir : Type`

A directory within the layout of a site.
Constructors
| `page (name : String) (id : Lean.Name) (text : Part Page) (contents : Array Blog.Dir) : Blog.Dir`
The directory's root is the provided page
| `blog (name : String) (id : Lean.Name) (text : Part Page) (contents : Array Blog.BlogPost) : Blog.Dir`
The directory's root is a blog
| `static (name : String) (files : System.FilePath) : Blog.Dir`
The directory's root contains static files, copied from `files` when the site is generated

These are usually constructed using a small embedded configuration language.

A blog is rendered using a theme, which is a collection of templates. Templates are monadic functions that construct `Html` from a set of dynamically-typed parameters.

structure

`Verso.Genre.Blog.Theme : Type`

A specification of how to render a site.
Constructor
`Verso.Genre.Blog.Theme.mk`
Fields
`primaryTemplate : Blog.Template`
The template used to render every page. It should construct an HTML value for the entire page, including the `<html>` element.In the `<head>` element, it should invoke `builtinHeader` to ensure that the required dependencies are present and that Verso-specific initialization is performed.In the body, it should check whether the parameter `"posts"` of type `Html` is present. If so, the page being rendered is a blog index, so it should place the parameter's value as a list of posts. If `"categories"` of type `Post.Categories` is present, it should render the contents as a category list.The parameter `"content"` of type `Html` contains the content of the page. It should be placed accordingly in the result.
`pageTemplate : Blog.Template`

Pages are rendered using `pageTemplate`, with the result being passed in the `"content"` parameter to `primaryTemplate`. This template should use the `"title"` and `"content"` parameters to construct the contents of the page.

`postTemplate : Blog.Template`
Blog posts are rendered using `pageTemplate`, with the result being passed in the `"content"` parameter to `primaryTemplate`. This template should use the `"title"` and `"content"` parameters to construct the contents of the post. Additionally, the `"metadata"` template of type `"Post.PartMetadata"` may be present; if so, it can be used to render the author, date, and categories of the post.

`archiveEntryTemplate : Blog.Template`
The template used to summarize a blog post in a post archive. It receives the parameters `"post"`, which contains the post, and `"summary"`, which contains the HTML summary to display.

`categoryTemplate : Blog.Template`
The template used to display a category at the top of a category's post list. It receives one parameter, `"category"`, which contains a `Post.Category`.

`adHocTemplates : Array String → Option Blog.Template.Override`
Customize the rendering of a given path by replacing the template and providing additional parameters

def

```
Verso.Genre.Blog.Template : Type
```

A procedure for producing HTML from parameters. An abbreviation for `TemplateM Html`

def

```
Verso.Genre.Blog.TemplateM (α : Type) : Type
```

A monad that provides template instantiation via dynamically-typed parameters.

def

```
Verso.Genre.Blog.Template.param {α : Type} [TypeName α]
(key : String) : Blog.TemplateM α
```

Returns the value of the given template, if it exists. If it does not exist or if it exists but has the wrong type, an exception is thrown.

def

```
Verso.Genre.Blog.Template.param? {α : Type} [TypeName α] (key : String) : Blog.TemplateM (Option α)
```

Returns the value of the given template, if it exists. If it exists but has the wrong type, an exception is thrown.

def

```
Verso.Genre.Blog.Template.builtinHeader : Blog.TemplateM Output.Html
```

Contains the contents of `<head>` that are needed for proper functioning of the site.

7 Manuals and Books

Verso's `Manual` genre can be used to write reference manuals, textbooks, or other book-like documents. It supports generating both HTML and PDFs via LaTeX, but the PDF support is relatively immature and untested compared to the HTML support.

def

```
Verso.Genre.Manual : Genre
```

A genre for writing reference manuals and other book-like documents.

structure

```
Verso.Genre.Manual.PartMetadata : Type
```

Metadata for the manual

Constructor

`Verso.Genre.Manual.PartMetadata.mk`

Fields

`shortTitle` : `Option String`
A shorter title to be shown in titlebars and tables of contents.

`shortContextTitle` : `Option String`
A shorter title to be shown in breadcrumbs for search results. Should typically be at least as short as `shortTitle`.

`authors` : `List String`
The book's authors

`authorshipNote` : `Option String`
An extra note to show after the author list

`date` : `Option String`
The publication date

`tag` : `Option Tag`
The main tag for the part, used for cross-references.

`file` : `Option String`
If this part ends up as the root of a file, use this name for it

`id` : `Option InternalId`

The internal unique ID, which is automatically assigned during traversal.

`number` : `Bool`
Should this section be numbered? If `false`, then it's like `\section*` in LaTeX

`draft` : `Bool`
If `true`, the part is only rendered in draft mode.

`assignedNumber` : `Option Numbering`
Which number has been assigned? This field is set during traversal.

`htmlToc` : `Bool`
If `true`, this part will display a list of subparts that are separate HTML pages.

`htmlSplit` : `HtmlSplitMode`
How should this document be split when rendering multi-page HTML output?

inductive type

`Verso.Genre.Manual.HtmlSplitMode : Type`

When rendering multi-page HTML, should splitting pages follow the depth setting?

Constructors

| `default : HtmlSplitMode`
  Follow the main setting
| `never : HtmlSplitMode`
  Do not split here nor in child parts

The `Manual` genre's block and inline element types are extensible. In the document, they consist of instances of `Manual.Block` and `Manual.Inline`, respectively:

structure

`Verso.Genre.Manual.Block : Type`

A custom block. The `name` field should correspond to an entry in the block descriptions table.

Constructor

`Verso.Genre.Manual.Block.mk`

Fields

`name` : `Lean.Name`
A unique name that identifies the block.

`id` : `Option InternalId`
A unique ID, assigned during traversal.

`data` : `Lean.Json`

Data saved by elaboration, potentially updated during traversal, and used to render output. This is the primary means of communicating information about a block between phases.

`properties` : `NameMap String`

A registry for properties that can be used to create ad-hoc protocols for coordination between block elements in extensions.

structure

`Verso.Genre.Manual.Inline` : `Type`

A custom inline. The `name` field should correspond to an entry in the block descriptions table.

Constructor

`Verso.Genre.Manual.Inline.mk`

Fields

`name` : `Lean.Name`

A unique name that identifies the inline.

`id` : `Option InternalId`

The internal unique ID, which is automatically assigned during traversal.

`data` : `Lean.Json`

Data saved by elaboration, potentially updated during traversal, and used to render output. This is the primary means of communicating information about a block between phases.

The fields `Block.name` and `Inline.name` are used to look up concrete implementations of traversal and output generation in run-time tables that contain descriptions:

structure

`Verso.Genre.Manual.BlockDescr` : `Type`

Implementations of all the operations needed to use a block.

Constructor

`Verso.Genre.Manual.BlockDescr.mk`

Fields

`init` : `TraverseState → TraverseState`

All registered initializers are called in the state prior to the first traversal.

`traverse` : `BlockTraversal Manual`

How the traversal phase should process this block

`toHtml : Option (BlockToHtml Manual (ReaderT ExtensionImpls IO))`

How to generate HTML. If `none`, generating HTML from a document

that contains this block will fail.

`extraJs : List String`

Extra JavaScript to add to a `<script>` tag in the generated HTML's `<head>`

`extraJsFiles : List JsFile`

Extra JavaScript to save to the static files directory and load in the generated HTMl's `<head>`

`extraCss : List String`

Extra CSS to add to a `<style>` tag in the generated HTML's `<head>`

`extraCssFiles : List (String × String)`

Extra CSS to save to the static files directory and load in the generated HTMl's `<head>`

`licenseInfo : List LicenseInfo`

Open-source licenses used by the block, to be collected for display in the final document.

`localContentItem : InternalId → Lean.Json → Array (Doc.Block Manual) → Except String (Array (String × Output.Html))`

Should this block be an entry in the page-local ToC? If so, how should it be represented?Entries should be returned as a preference-ordered array of representations. Each representation consists of a string and some HTML; the string should represent the HTML's content if all formatting were stripped. Items are compared for string equality, with later suggestions used in case of overlap, but the HTML is what's displayed. Exceptions are logged as errors during HTML generation.The empty array means that the block should not be included.

`toTeX : Option (BlockToTeX Manual (ReaderT ExtensionImpls IO))`

How to generate TeX. If `none`, generating TeX from a document that contains this block will fail.

`usePackages : List String`

Required TeX `\usepackage` lines

`preamble : List String`

Required items in the TeX preamble

structure

`Verso.Genre.Manual.InlineDescr : Type`

Implementations of all the operations needed to use an inline element.

Constructor

`Verso.Genre.Manual.InlineDescr.mk`

Fields

`init : TraverseState → TraverseState`

All registered initializers are called in the state prior to the first traver-

sal.

`traverse : InlineTraversal Manual`

Given the contents of the `data` field of the corresponding `Manual.Inline` and the contained inline elements, carry out the traversal pass.In addition to updating the cross-reference state through the available monadic effects, a traversal may additionally replace the element with another one. This can be used to e.g. emit a cross-reference once the target becomes available in the state. To replace the element, return `some`. To leave it as is, return `none`.

`toHtml : Option (InlineToHtml Manual (ReaderT ExtensionImpls IO))`

How to generate HTML. If `none`, generating HTML from a document that contains this inline will fail.

`extraJs : List String`

Extra JavaScript to add to a `<script>` tag in the generated HTML's `<head>`

`extraJsFiles : List JsFile`

Extra JavaScript to save to the static files directory and load in the generated HTMl's `<head>`

`extraCss : List String`

Extra CSS to add to a `<style>` tag in the generated HTML's `<head>`

`extraCssFiles : List (String × String)`

Extra CSS to save to the static files directory and load in the generated HTMl's `<head>`

`licenseInfo : List LicenseInfo`

Open-source licenses used by the inline, to be collected for display in the final document.

`localContentItem : InternalId → Lean.Json → Array (Doc.Inline Manual) → Except String (Array (String × Output.Html))`

Should this inline be an entry in the page-local ToC? If so, how should it be represented?Entries should be returned as a preference-ordered array of representations. Each representation consists of a string and some HTML; the string should represent the HTML's content if all formatting were stripped. Items are compared for string equality, with later suggestions used in case of overlap, but the HTML is what's displayed.The empty array means that the inline should not be included.

`toTeX : Option (InlineToTeX Manual (ReaderT ExtensionImpls IO))`

How to generate TeX. If `none`, generating TeX from a document that contains this inline will fail.

`usePackages : List String`

Required TeX `\usepackage` lines

`preamble : List String`

Required items in the TeX preamble

Typically, the `inline_extension` and `block_extension` commands are used to simultaneously define an element and its descriptor, registering them for use by `manualMain`.

7.1 Tags and References

The `Manual` genre maintains a table of link targets for various namespaces, such as documented constants, documented syntax, technical terminology, and sections. In this table, domain-specific names are mapped to their documentation location. For items such as document sections that don't have a clear, unambiguous, globally-unique name, Verso requires such a name to be manually specified before it is in the table. Extensions and parts for which names should be manually specified take a `tag` parameter.

Specifying a tag has the following benefits:

- The item is included in the quick-jump box and the index.
- The tag can be used to construct permalinks that will continue to work even if the document is reorganized, so long as the tag is maintained.
- The item can be linked to automatically from other documents.

Tags should be specified for all sections that the author considers to have a stable identity.

7.2 Paragraphs

The `paragraph` directive indicates that a sequence of blocks form a logical paragraph. Verso's markup language shares one key limitation with Markdown and HTML: bulleted lists and code blocks cannot be contained within paragraphs. However, there's no *a priori* reason to reject this, and many real documents include lists in paragraphs. When using the `paragraph` directive, HTML output wraps the contents in a suitable element that causes their internal margins to be a bit smaller, and TeX output omits the blank line that would signal a paragraph break to TeX.

7.3 Docstrings

Docstrings can be included using the `docstring` directive. For instance,

```
{docstring List.forM}
```

results in

def

```
List.forM.{u, v, w} {m : Type u → Type v} [Monad m] {α :
Type w} («as» : List α) (f : α → m PUnit) : m PUnit
```

Applies the monadic action `f` to every element in the list, in order.`List.mapM` is a variant that collects results. `List.forA` is a variant that works on any `Applicative`.

The `docstring` command takes a positional parameter which is the documented name. It also accepts the following optional named parameters:

`allowMissing : Bool` If `true`, missing docstrings are a warning rather than an error.

`hideFields : Bool` If `true`, fields or methods of structures or classes are not shown.

`hideStructureConstructor : Bool` If `true`, constructors of structures or classes are not shown.

`label : String` A label to show instead of the default.

The `tactic` directive and the `optionDocs` command can be used to show documentation for tactics and compiler options, respectively.

```
:::tactic "induction"
:::
```

results in Assuming `x` is a variable in the local context with an inductive type, `induction x` applies induction on `x` to the main goal, producing one goal for each constructor of the inductive type, in which the target is replaced by a general instance of that constructor and an inductive hypothesis is added for each recursive argument to the constructor. If the type of an element in the local context depends on `x`, that element is reverted and reintroduced afterward, so that the inductive hypothesis incorporates that hypothesis as well.For example, given `n : Nat` and a goal with a hypothesis `h : P n` and target `Q n`, `induction n` produces one goal with hypothesis `h : P 0` and target `Q 0`, and one goal with hypotheses `h : P (Nat.succ a)` and `ih₁ : P a → Q a` and target `Q (Nat.succ a)`. Here the names `a` and `ih₁` are chosen automatically and are not accessible. You can use `with` to provide the variables names for each constructor.

- `induction e`, where `e` is an expression instead of a variable, generalizes `e` in the goal, and then performs induction on the resulting variable.
- `induction e using r` allows the user to specify the principle of induction that should be used. Here `r` should be a term whose result type

must be of the form `C t`, where `C` is a bound variable and `t` is a (possibly empty) sequence of bound variables

- `induction e generalizing z₁ ... zₙ`, where `z₁ ... zₙ` are variables in the local context, generalizes over `z₁ ... zₙ` before applying the induction but then introduces them in each goal. In other words, the net effect is that each inductive hypothesis is generalized.

- Given `x : Nat`, `induction x with` zero => tac_1 | succ x' ih => tac_2| uses tactic `tac₁` for the `zero` case, and `tac₂` for the `succ` case.

and

```
{optionDocs pp.deepTerms.threshold}
```

results in (pretty printer) when `pp.deepTerms` is false, the depth at which terms start being replaced with ⋯

7.4 Technical Terminology

The `deftech` role can be used to annotate the definition of a technical term . Elsewhere in the document, `tech` can be used to annotate a use site of a technical term. A *technical term* is a term with a specific meaning that's used precisely, like this one. References to technical terms are valid both before and after their definition sites.

def

```
Verso.Genre.Manual.deftech : Elab.RoleExpanderOf
TechArgs
```

Defines a technical term.Internally, these definitions are saved according to a key that is derived by stripping formatting information from the arguments in `args`, and then normalizing the resulting string by:

1. lowercasing it
2. replacing trailing `"ies"` with `"y"`
3. replacing consecutive runs of whitespace and/or hyphens with a single space

Call with `(normalize := false)` to disable normalization, and `(key := some k)` to use `k` instead of the automatically-derived key.Uses of `tech` use the same process to derive a key, and the key is matched against the `deftech` table.

def

`Verso.Genre.Manual.tech : Elab.RoleExpanderOf TechArgs`

Emits a reference to a technical term defined with `deftech`. Internally, these terms are found according to a key that is derived by stripping formatting information from the arguments in `args`, and then normalizing the resulting string by:

1. lowercasing it
2. replacing trailing `"ies"` with `"y"`
3. replacing consecutive runs of whitespace and/or hyphens with a single space

Call with `(normalize := false)` to disable normalization, and `(key := some k)` to use `k` instead of the automatically-derived key.

7.5 Open-Source Licenses

To facilitate providing appropriate credit to the authors of open-source JavaScript, CSS, and HTML libraries used to render a Verso document, inline and block elements can specify the licenses of components that they include in their rendered output. This is done using the `BlockDescr.licenseInfo` and `InlineDescr.licenseInfo` fields. These contain a `LicenseInfo`:

structure

`Verso.Genre.Manual.LicenseInfo : Type`

A description of an open-source license used by a frontend component that's included in generated HTML. This is used to create an attribution page.

Constructor

`Verso.Genre.Manual.LicenseInfo.mk`

Fields

`identifier : String`
SPDX license identifier

`dependency : String`
Dependency name. This is used in a header, and for alphabetical sorting.

`howUsed : Option String`
How is the dependency used? This can give better credit.Examples:

- "The display of mathematical formulae is powered by KaTeX."

- "The graphs are rendered using Chart.js."

`link` : `Option String`
A link target used to credit the dependency's author
`text` : `Array (Option String × String)`
The license or notice text in plain text, divided into sections with optional headers.

The `licenseInfo` command displays the licenses for all components that were included in the generated document:

def

```
Verso.Genre.Manual.licenseInfo : Elab.BlockCommandOf
Unit
```

Displays the open-source licenses of components used to build the document.

8 Index

9 Dependencies

This document contains the following open-source libraries, or code derived from them: