The proofgraph package

Pierre Senellart
pierre@senellart.com

2026/07/31 v1.1.0

Abstract

The proofgraph package automatically produces a graph of the dependencies between the results (theorems, lemmas, propositions) of a mathematical article. It infers an edge from one result to another whenever the proof of the former refers to the latter through an ordinary cross-reference, so that most dependencies need no manual annotation; commands are provided for those a visible reference does not capture. The graph is written as a Graphviz `.dot` file, which can be rendered externally or, optionally, embedded into the document automatically.

1 Introduction

When writing or reading a mathematical article, it is often useful to visualise how the results depend on one another: which lemma is used in the proof of which theorem, and so on. proofgraph builds such a dependency graph automatically. Each theorem-like environment becomes a node; each cross-reference (`\ref`, `\cref`…) appearing *inside a proof* becomes an edge from the proved result to the referred-to result. Usually no manual annotation is required, and commands are provided for the dependencies a visible reference does not capture.

The package is best explained by a small example of its use. This manual is itself a proofgraph document, so the results stated below are typeset with the package enabled, and the graph shown at the end of this section is the one they produce.

Load proofgraph in the preamble, before the `\newtheorem` commands that declare your theorem-like environments (so it sees them as they are created). Here, as in this manual, we pass three options: `autorun` renders the graph with Graphviz and embeds it automatically (it needs shell-escape); `cite` captures the `\cite`s made inside proofs as dependencies on external work; and `citelabel=tag` labels those citation nodes by their printed tag:

```
\usepackage[autorun,cite,citelabel=tag]{proofgraph}
\newtheorem{theorem}{Theorem}
\newtheorem{lemma}{Lemma}
\newtheorem{corollary}{Corollary}
```

Then state and prove, as usual, a few results that build on one another:

```
\begin{lemma}\label{lem:even}
  The sum of two even integers is even.
\end{lemma}
```

```
\begin{lemma}\label{lem:odd}
  The sum of two odd integers is even.
\end{lemma}
\begin{theorem}\label{thm:parity}
  The sum of two integers of the same parity is even.
\end{theorem}
\begin{proof}
  By Lemmas~\ref{lem:even} and~\ref{lem:odd}; see~\cite{euclid}.
\end{proof}
\begin{corollary}\label{cor:double}
  For every integer $n$, the number $2n$ is even.
\end{corollary}
\begin{proof}
  Apply Theorem~\ref{thm:parity} to $n$ and $n$.
\end{proof}
```

We obtain the following results, exactly as without the package:

Lemma 1. *The sum of two even integers is even.*

Lemma 2. *The sum of two odd integers is even.*

Theorem 1. *The sum of two integers of the same parity is even.*

Proof. By Lemmas 1 and 2; see [1]. □

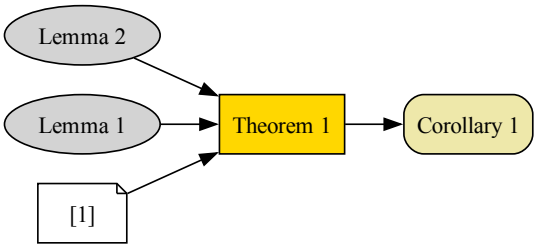
Corollary 1. *For every integer n, the number $2n$ is even.*

Proof. Apply Theorem 1 to n and n. □

In addition, proofgraph writes out the dependency graph. With the `autorun` option it is rendered with Graphviz and included wherever you call `\proofgraph`:

```
\proofgraph[width=0.7\textwidth]
```

which here produces:



Each cross-reference made inside a proof becomes an arrow *into* the result that proof establishes: the two lemmas point to Theorem 1, which in turn points to Corollary 1. The `\cite` in the proof of Theorem 1 is captured in the same way, thanks to the `cite` option: the external work appears as a distinct node (drawn note-shaped, and labelled here by its printed citation tag, "`[1]`", because of `citelabel=tag`) with an arrow into the theorem it helps prove. Moreover, since

this manual loads hyperref and TikZ, the graph's nodes are *clickable*: try clicking one to jump to the result it represents (see `\proofgraph` and the `hyperlinks` option).

A handful of commands tune such a graph: `\proofgraphstyle` sets the appearance of each kind of node (the colours here come from it); `\uses` and `\proofgraphedge` record a dependency that no visible `\ref` expresses; `\proofof` attaches a proof to its result; `\proofgraphuntrack` keeps an environment out of the graph; and `\proofgraphignore` and `\proofgraphexclude` drop an unwanted edge or node. The next section puts several of these to work on a real article; Section 3 then documents them, and the package options, in full.

2 A real-world example

The introductory example is deliberately tiny. To show what proofgraph produces on a genuine article, the graph in Figure 1 is the one obtained from *Connecting Knowledge Compilation Classes and Width Parameters* [2]. It has 62 numbered results, drawn from five environments (`result`, `theorem`, `proposition`, `corollary` and `lemma`), depends on 24 external works, and contains 114 edges in all.

The graph was produced from the paper essentially as published. Beyond loading the package, the only additions were the following.

- The `cite` option was switched on, and citation nodes labelled by their printed tag, by loading the package as `\usepackage[cite=true,citelabel=tag]{proofgraph}`. The `\cite`s made inside proofs then appear as the note-shaped nodes ("`[17]`", "`[46, Theorem 4.10]`"…) along the left of the graph; a work cited for two different results, or for two of its own theorems, gives one node per `\cite` note, so those 23 works appear as 39 nodes.

- One environment was kept out of the graph with a single `\proofgraphuntrack{observation}` (a lone preliminary observation that nothing else uses).

- The five result kinds were styled, with decreasing prominence, by five `\proofgraphstyle` declarations:

```
\proofgraphstyle{result}{shape=box,style="filled,bold",%
                         fillcolor=orange,peripheries=2}
\proofgraphstyle{theorem}{shape=box,style=filled,%
                          fillcolor=gold,penwidth=2}
\proofgraphstyle{corollary}{shape=box,style="rounded,filled",%
                            fillcolor=palegoldenrod}
\proofgraphstyle{proposition}{shape=box,style="rounded,filled",%
                              fillcolor=lightblue}
\proofgraphstyle{lemma}{shape=ellipse,style=filled,%
                        fillcolor=lightgray}
```

- Eleven `\proofgraphedge` commands, recording 27 edges in total, were added for the dependencies that no `\ref` inside a `proof` expresses: eight "obvious" corollaries stated without a `proof` environment (`\proofgraphedge{cor:upper_bound}{thm:upper_bound,...}`), and

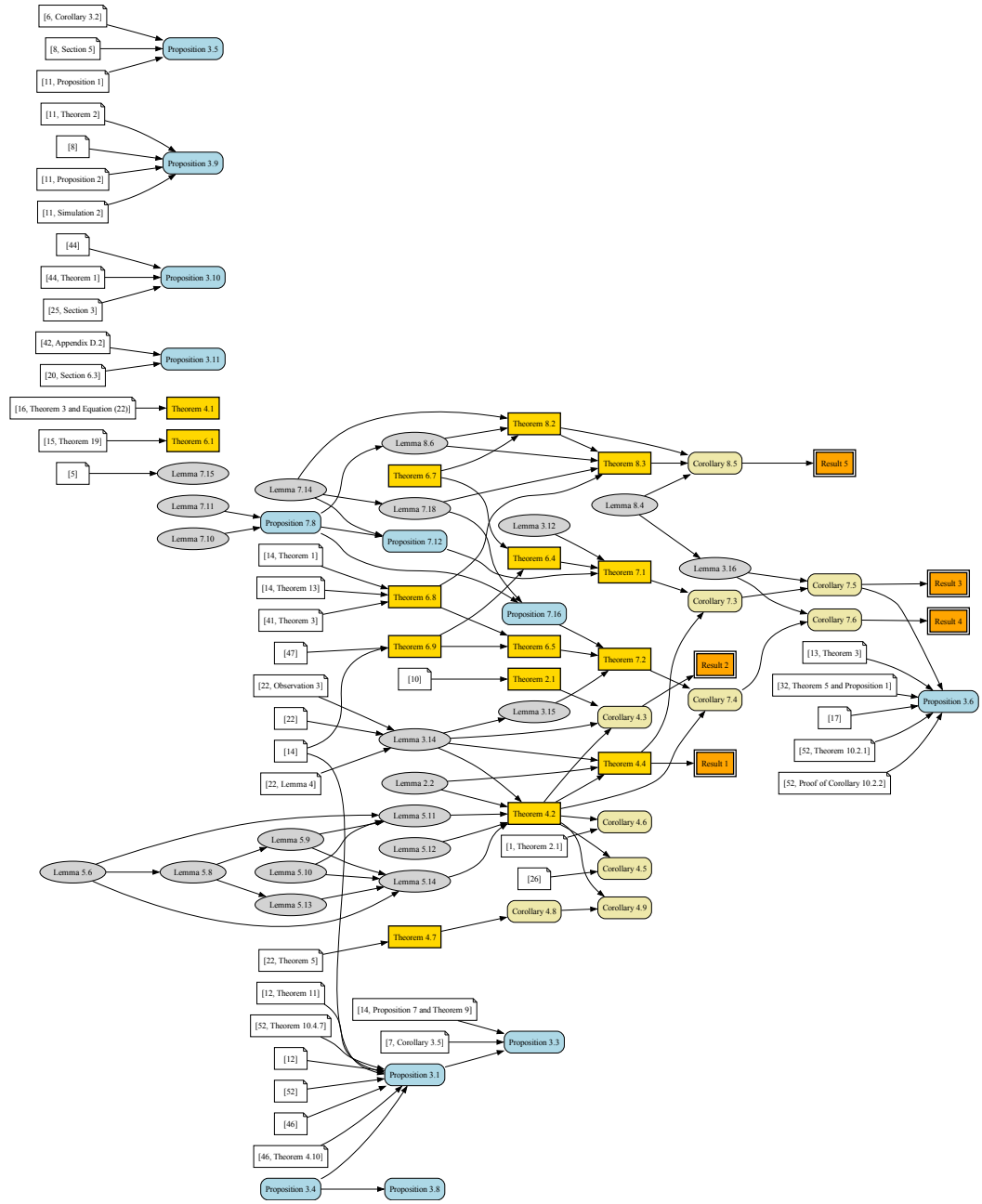


Figure 1: The graph proofgraph produces for [2]. The colours and shapes make the structure legible, with the orange double-bordered introduction `results` on the right, fed by the gold theorems, the blue and pale-gold propositions and corollaries, and the grey lemmas.

three theorems whose proof is developed across a whole section rather than inside a single `proof`.

- One `\proofgraphciteedge`, for a lemma stated without a proof, its proof being in the conference version of the article.

Nothing else was annotated: every other edge, and the numbering of all 62 nodes, was inferred automatically from the ordinary cross-references the proofs already contained. In particular, no `\uses`, `\proofof`, `\proofgraphignore` or `\proofgraphexclude` was needed. The five `result` environments that restate the headline results in the introduction (`\begin{result}[(Corollary~\ref{cor:obdd_omega})]`…) are linked to the body results they preview through the `\ref` in their optional title, captured automatically (see the note on reference-carrying titles in Section 3). Two compilation runs (plus BibTeX, needed once for the `tag` labels) suffice; the `.dot` file is then rendered with Graphviz, `dot -Tpdf main.dot -o main.pdf`, as with any other document.

3 Usage

3.1 Default behaviour

Load proofgraph in the preamble. It only needs to come *after* whatever provides your theorem environments, that is, after `\usepackage{amsthm}` if you load it yourself (some classes, such as lipics or acmart, provide theorems for you, in which case there is nothing to load first), and *before* the `\newtheorem` commands that declare your theorem-like environments, so that proofgraph sees them as they are created:

```
\usepackage{amsthm}      % or nothing, if your class provides theorems
\usepackage{proofgraph}
\newtheorem{theorem}{Theorem}
\newtheorem{lemma}{Lemma}
```

Then write your document as usual. After two compilation runs, a file `\jobname.dot` is produced. Render it with Graphviz:

```
dot -Tpdf myfile.dot -o mygraph.pdf
```

Every *numbered* result of a theorem-like environment you declare becomes a node, and every cross-reference (`\ref`, `\cref`…) inside a proof becomes an edge into the result that proof establishes, with no annotation. The commands captured are `\ref`, `\cref` and `\Cref` (cleveref), `\autoref` (hyperref), `\eqref` (amsmath), `\vref` (varioref) and `\zcref` (zref-clever, in every form: starred, with options, and with a comma-separated list of labels), each of them whenever it is defined. Purely page-related references (`\pageref`, `\cpageref`, `\zcpageref`) are not captured, as they point at a place rather than at a result; `\uses` records a dependency they would have expressed. Nodes are labelled by the result's formatted name and number ("Theorem 1"), not the environment name. Unnumbered (starred) results are skipped, as they are usually expository or repeated statements; `\proofgraphtrack` draws a chosen one anyway. A cross-reference in the *optional title* of a result is captured the same way: `\begin{theorem}[(generalises~\ref{thm:x})]` in a result labelled $\langle A \rangle$ adds

an edge from ⟨*A*⟩ to `thm:x`, handy for restatements (e.g., an introduction that previews results with `\begin{result}[(Corollary~\ref{cor:y})]`). The *body* of a result is read the same way: a statement that refers to an earlier result, "Under the hypotheses of Theorem 2…", depends on it just as a proof would. Set `statements=false` to build the graph from proofs alone.

A result *stated inside a proof*, a claim raised and settled in the course of an argument, is drawn as a node of its own and given an edge to the result that proof establishes: the claim is a step of that proof, and the graph says so without any annotation. Its own proof, whether it follows the claim as a nested `proof` environment or as the rest of the enclosing one, hangs below the claim, so the argument appears in the graph with the shape it has on the page. An unnumbered claim, like any unnumbered result, is left out, and its dependencies with it; number it, or name it in `\proofgraphtrack`, to have it drawn.

3.2 Recording dependencies by hand

`\uses{`⟨***label***⟩`}` Inside a proof, records a dependency on the result labelled ⟨*label*⟩ even when no visible reference is made. A comma-separated list is accepted, as in `\uses{lem:a,lem:b}`.

`\proofgraphedge{`⟨***A***⟩`}{`⟨***labels***⟩`}` Adds edges recording that result ⟨*A*⟩ depends on the result(s) ⟨*labels*⟩ (a comma-separated list), like a `\uses` but stated outside any proof. Handy for an "obvious" corollary that relies on earlier results without a `proof` environment, as in `\proofgraphedge{cor:x}{thm:y,lem:z}`. It may appear anywhere; all labels are resolved when the graph is written.

`\usescite[`⟨***note***⟩`]{`⟨***keys***⟩`}` Inside a proof, the `\uses` of external work: records a dependency on the bibliography entries ⟨*keys*⟩ (a comma-separated list) and draws them as citation nodes, without citing them visibly. The optional ⟨*note*⟩ is the one `\cite` would have taken, as in `\usescite[Thm~3]{knuth}`; it labels the node and, as with a captured `\cite`, tells one result of a work from another.

`\proofgraphciteedge[`⟨***note***⟩`]{`⟨***A***⟩`}{`⟨***keys***⟩`}` The `\proofgraphedge` of citation nodes: records that result ⟨*A*⟩ depends on the cited work(s) ⟨*keys*⟩, stated outside any proof, as in `\proofgraphciteedge{thm:x}{knuth}`. Useful for a result stated without a proof, which has no place to hold a citation. Like `\proofgraphedge` it may appear anywhere, all labels being resolved when the graph is written.

Both commands create the citation node when the work has none yet, and both work whether or not the `cite` option is on: that option only says whether a plain `\cite` is captured on its own. A work reached both ways, once by `\cite` capture and once by hand, still yields a single node.

`\proofof{`⟨***label***⟩`}` By default a proof is attached to the most recently started result. A detached proof whose optional title names its result is attached automatically: `\begin{proof}` with `[of Theorem~\ref{thm:x}]` attaches the proof to `thm:x` (for the `amsthm` `proof` environment, for `proof` defined as a theorem in the Springer classes, and for deferred proofs as with `apxproof`). When none of this applies, `\proofof` inside the proof pins it to the result

labelled ⟨*label*⟩. It pins the proof *as a whole*, wherever in it it stands, so the references made before it count as much as those made after; ⟨*label*⟩ may be the label of a result stated later in the document. This is what a proof written in an appendix, or anywhere else away from its result, needs and all that it needs: its dependencies are then captured as those of a proof placed right after its result would be, and recording them by hand with `\uses` or `\proofgraphedge` is not called for.

The pin covers the proofs *nested* in the pinned one as well, the steps of a structured proof (as pf2 writes them) being proofs in their own right; a `\proofof` in a nested proof pins that proof alone, overriding for it the pin of the proof around it. A nested proof that comes after a *result* opened inside the enclosing proof – the proof of a claim stated in the course of an argument – attaches to that claim instead, as an ordinary proof would.

3.3 Choosing what appears in the graph

Result environments are detected automatically; these commands adjust which results, and which edges, are kept.

`\proofgraphtrack{`⟨***names***⟩`}` Forces the listed environment types to be tracked, overriding the default exclusions, which are `definition`, `example`, `remark` and `note`. Use it in the preamble.

`\proofgraphuntrack{`⟨***names***⟩`}` Excludes further environment types, as in `\proofgraphuntrack{observation}`. Use it in the preamble.

`\proofgraphproofenv{`⟨***names***⟩`}` Treats the listed environments as proofs, besides `proof`, so that the references they make become dependencies of the result they establish. Use it in the preamble, as in `\proofgraphproofenv{pf}`. It serves a document whose proofs live in an environment of another name, for instance because a proof package had to be renamed to avoid the clash with amsthm's `proof` (as with pf2), and equally a `\newtheorem{claimproof}` used for the proofs of claims. Being proofs, the environments named are also removed from the results drawn, even when they are numbered; a `\proofgraphtrack` on the same name overrides that, for an environment to be drawn as a result as well.

An environment taking an optional title is wrapped as `proof` is, so a detached `\begin{pf}[of Theorem~\ref{thm:x}]` attaches to `thm:x`; one declared with `\newtheorem` attaches to the result it follows, or to the one `\proofof` names.

`\proofgraphignore{`⟨***A***⟩`}{`⟨***B***⟩`}` Drops the dependency of the result ⟨*A*⟩ on the result ⟨*B*⟩, the edge the proof of ⟨*A*⟩ produced by referring to ⟨*B*⟩: an unwanted forward reference, say. It may appear anywhere, the preamble included.

The two labels are those of `\proofgraphedge`, in that order: the result first, what its proof uses second, the way the dependency is *recorded*. They are *not* the ends of the arrow *drawn*, which with the default `direction=usedby` runs the other way, from ⟨*B*⟩ to ⟨*A*⟩. So the rule undoing a `\proofgraphedge{cor:x}{thm:y}` is

`\proofgraphignore{cor:x}{thm:y}`, with the same two labels in the same order, and the rule dropping the arrow the picture draws from `lem:a` to `thm:m` is `\proofgraphignore{thm:m}{lem:a}`. The `direction` option does not enter into it: like every filter the rule works on the dependency recorded, `direction` settling only how it is drawn, so a graph turned round keeps its rules. A rule that drops nothing is reported at the end of the run, and says so when it is the two labels that have been swapped.

`\proofgraphignorecite[`⟨***note***⟩`]{`⟨***A***⟩`}{`⟨***keys***⟩`}` The same for a citation node, which `\proofgraphignore` cannot name, both of its arguments being labels: drops the dependency of the result ⟨*A*⟩ on the cited works ⟨*keys*⟩ (a comma-separated list of BibTeX keys). Its arguments are those of `\proofgraphciteedge`, which it undoes. The optional ⟨*note*⟩ picks, of a work cited with a note and so drawn as more than one node, the node of that note, so `\proofgraphignorecite[Thm~3]{thm:x}{knuth}` leaves the node of a plain `\cite{knuth}` alone. A citation node no edge reaches is not drawn, so dropping the last dependency on a work removes its node with it.

`\proofgraphexclude{`⟨***label***⟩`}` Removes the result labelled ⟨*label*⟩, together with all its incident edges, from the graph.

`\proofgraphcitecommand{`⟨***names***⟩`}` Captures further citation commands, given as a comma-separated list of names without their backslash, as in `\proofgraphcitecommand{citeauthor,citeyear}`. Use it in the preamble. Any command taking the usual ⟨*cs*⟩`*[`⟨*pre*⟩`][`⟨*post*⟩`]{`⟨*keys*⟩`}` arguments works, including one you define yourself. Naming a command the document does not define is harmless.

It cannot help with the biblatex *multicite* commands (`\cites`, `\parencites`, `\textcites`), whose `(`⟨*pre*⟩`)(`⟨*post*⟩`)` syntax and repeated key groups proofgraph does not parse; cite such works individually, or record them with `\uses`.

3.4 Styling the graph

The attributes are passed verbatim to Graphviz; see its attribute reference, https://graphviz.org/doc/info/attrs.html, for the node attributes available (and https://graphviz.org/doc/info/shapes.html for the shapes).

`\proofgraphstyle{`⟨***env***⟩`}{`⟨***attrs***⟩`}` Sets Graphviz node attributes for all results of the environment ⟨*env*⟩, e.g., `shape=ellipse` or `style=filled,fillcolor=gold`.

`\proofgraphstylecite{`⟨***attrs***⟩`}` Sets the Graphviz attributes of the external citation nodes added by the `cite` option (default `shape=note`), as in `\proofgraphstylecite{shape=box,style=dashed,color=gray}`.

3.5 Embedding the rendered graph

`\proofgraph[`⟨***options***⟩`]` Includes the rendered graph image at this point, passing the optional ⟨*options*⟩ to the underlying `\includegraphics` (e.g., `\proofgraph[width=\linewidth]`). This is the only feature that needs

graphicx, which the package therefore does not load on its own: a document that uses `\proofgraph` must load graphicx itself (loading tikz for the `hyperlinks` overlay also suffices). With the `autorun` option, Graphviz is run at the end of a compilation in which the graph changed (shell-escape required) to produce that image, so `\proofgraph` embeds the graph from the previous run and is up to date after two runs. Without `autorun`, no Graphviz is run: `\proofgraph` embeds a pre-existing `\jobname-proofgraph.`⟨*format*⟩ file if you have rendered one under that name, and otherwise warns. When hyperref and TikZ are both loaded (with `hyperlinks` on and the `-Tplain` file from `autorun` present), every node becomes an internal hyperlink to the result it represents: Graphviz writes the node positions to that `-Tplain` file and `\proofgraph` overlays a transparent `\hyperref` area on each node (needed because the link annotations Graphviz puts in the `.pdf` are discarded by `\includegraphics`).

3.6 Options

`selfloops` Either `remove` (default), dropping edges from a result to itself, or `keep`, retaining them.

`autorun` Boolean (default false). Runs Graphviz automatically at the end of the run; requires compilation with shell-escape enabled. Graphviz is run only when the graph, the engine or the format has changed since the last rendering, so that repeated compilations converge and a build tool such as latexmk settles instead of rerunning indefinitely.

`engine` Graphviz layout engine (default `dot`); any Graphviz engine works, such as `neato`, `fdp`, `sfdp`, `circo` or `twopi` (see https://graphviz.org/docs/layouts/).

`format` Output format for `autorun` and `\proofgraph` (default `pdf`); any Graphviz output format works, such as `png`, `svg` or `ps` (see https://graphviz.org/docs/outputs/).

`hyperlinks` Boolean (default true). Makes the nodes of an embedded graph clickable when hyperref and TikZ are available (see `\proofgraph`); set it false to embed the graph as a plain image.

`direction` Either `usedby` (default), orienting an edge from the result that is used to the result that uses it (so an arrow `a->b` reads "`a` is used by `b`"), or `uses`, the reverse.

`rankdir` Graphviz graph direction (default `LR`).

`cite` Boolean (default false). Also captures `\cite` inside proofs as dependencies on external work, drawn as distinct nodes (Section 2), and, unless `statements` is off, in the body of a result too. A `\cite` note is kept: `\cite[Thm~3]{key}` labels the node "Thm 3, ...". In the two-optional-argument form of natbib and biblatex, `\cite[see][Thm~3]{key}`, the post-note likewise labels the node and the pre-note is ignored. Every citation command of LaTeX, natbib and biblatex that cites a work is captured the same way, whenever the document defines it, starred variants included:

`\citet`, `\citep`, `\citealt`, `\citealp`, `\parencite`, `\textcite`, `\autocite`, `\footcite`, `\footcitetext`, `\smartcite`, `\supercite`, `\fullcite` and `\footfullcite`, together with the commands printing only a fragment of a citation (`\citeauthor`, `\citefullauthor`, `\citetitle`, `\citeyear`, `\citeyearpar`, `\citedate`, `\citeurl`, `\citenum`) and all their capitalised forms. A work invoked twice in one proof, by `\citeauthor` and by `\cite` say, still yields a single edge. `\proofgraphcitecommand` adds any further command; `\usescite` and `\proofgraphciteedge` record a citation node by hand, with or without this option.

`statements` Boolean (default true). Captures cross-references, and citations when `cite` is on, in the *body* of a result as well, not only in its proof and in its optional title: a statement reading "Under the hypotheses of Theorem 2…" or "… the bound of [**?**, Thm 3]" records that dependency on the result being stated, as a proof making the same reference would. The capture stops at the `\end` of the result, so references in the surrounding text record nothing. Set it false for a graph built from proofs alone, which is the right reading when statements refer to earlier results to contrast with them rather than to rely on them.

`citelabel` Either `key` (default), labelling citation nodes by their BibTeX key, or `tag`, using the printed citation tag instead ("[1]", "[Knu74]", "[Abiteboul et al. 1995]"…), which is only known after a first run (so it needs two runs). It supports the standard, hyperref-wrapped and natbib (numeric and author–year) bibliographies. biblatex is supported in part: the tag is recovered for its numeric styles (`[1]`) and its alphabetic ones (`[Knu68]`), but not for its author–year styles, whose tag is a formatted name list that biblatex exposes as no single field. Whenever no clean tag can be recovered the node keeps the key, which is always a correct, if terser, label.

The note of a `\cite` is set where a citation prints it, inside the tag and after it: `\cite[pp.~23--27]{knuth}` gives a node labelled "[Knu68, pp. 23–27]", carrying the dashes of the document, and "knuth, pp. 23–27" when the label is the key.

`file` Base name of the output file (default `\jobname`).

3.7 Use with apxproof

apxproof moves proofs to the appendix and typesets them there, so the references a proof makes are only executed at that later point. To attribute them to the right result, proofgraph relies on the hook `\apxproofhook`, which apxproof fires inside each deferred proof. This hook is available in apxproof **v1.4.1** and later; load the two packages in either order and no further setup is needed.

With an *earlier* apxproof the document still compiles cleanly and every result still becomes a node, but *no proof dependencies are captured for the proofs deferred to the appendix*, because proofgraph cannot tell which appendix proof belongs to which result (dependencies from any proofs left in the main text are still captured as usual). In that case, add the deferred ones manually with `\uses` (and `\proofof` to pin a proof to its result), or upgrade apxproof.

4 Limitations

- Environments declared through `\newtheorem` (or `\spnewtheorem`) *after* the package is loaded are tracked automatically. Environments predefined by the document class *before* that (as in lipics, acmart or the Springer classes) are also detected, provided their name is one of the common result names proofgraph probes at `\begin{document}` (`theorem`, `lemma`, `proposition`, `corollary`…); an environment with an unusual name predefined before the package is loaded must be added with `\proofgraphtrack`.
- Results must be labelled for references to them to be resolved.
- Two compilation runs are required, as for any cross-reference.

5 See also

result-graph [3] does for a Lean 4 formalisation what proofgraph does for an article: it draws the dependency graph of the results of a development, deliberately with the same vocabulary, node styles and edge convention, so that the graph of a formalisation and the graph of the paper it formalises look like one family. There the dependencies are not inferred from cross-references but read off the elaborated proof terms, and are therefore exact; they also distinguish, for free, a result used in the *statement* of another from one used only in its *proof*, which proofgraph, under the `statements` option, records but draws alike. And since a whole development is far too large for one figure, it is sliced into a small neighbourhood of each result, rather than drawn as a single graph as here.

References

[1] Euclid. *Elements*, Book IX. c. 300 BCE.

[2] A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *Theory of Computing Systems*, **64**(5):861–914, 2020. doi:10.1007/s00224-019-09930-2.

[3] P. Senellart. *result-graph: dependency graphs of the results of a Lean 4 project.* https://github.com/PierreSenellart/result-graph.

6 Implementation

The format required is that of October 2020, the first to provide `\NewDocumentCommand` in the kernel; expl3, which the package also uses, has been preloaded since the February 2020 release.

```
⟨*package⟩
\NeedsTeXFormat{LaTeX2e}[2020/10/01]
\ProvidesPackage{proofgraph}
  [2026/07/31 v1.1.0 Dependency graph of results]
```

6.1 Dependencies

amsthm (loaded for its heading hooks) defines the `proof` environment with `\newenvironment`, which aborts with "Command `\proof` already defined" under a class that already provides one (e.g. the Springer `svjour3` `\proof`). When we are about to load amsthm ourselves (it is not already loaded), we save the class's `proof`/`\endproof` and free them so amsthm loads cleanly, then restore the class's versions afterwards, so its proof styling is kept (and proofgraph still hooks that `proof` environment). If amsthm is already loaded, `\RequirePackage` is a no-op and `\proof` is left untouched.

```
\@ifpackageloaded{amsthm}{}{%
  \@ifundefined{proof}{}{%
    \let\pgph@class@proof\proof \let\pgph@class@endproof\endproof}%
  \let\proof\relax \let\endproof\relax}
\RequirePackage{amsthm}
\@ifundefined{pgph@class@proof}{}{%
  \let\proof\pgph@class@proof \let\endproof\pgph@class@endproof}
\RequirePackage{etoolbox}
\RequirePackage{xstring}
\RequirePackage{kvoptions}
\RequirePackage{pdftexcmds}
```

pdftexcmds provides `\pdf@filemdfivesum`, with which `autorun` recognises an unchanged graph and skips the Graphviz run (see `\pgph@maybrender`). On an engine that offers no file checksum the package still works, only without that optimisation.

graphicx is needed only to embed a rendered graph with `\proofgraph`; the core `.dot`-writing machinery does not use it. Rather than burden every document with it, we leave it unloaded and check for `\includegraphics` at the point of use (see `\proofgraph`). When the `hyperlinks` overlay is active tikz is loaded, which pulls graphicx in anyway.

`\pgph@trimsp` An expandable space-trimmer (labels inside a `\cite`/`\cref`/`\uses` list arrive with the space after each comma), built on the `expl3` primitive so it is robust to leading spaces and braces.

```
\ExplSyntaxOn
\cs_new:Npn \pgph@trimsp #1 { \tl_trim_spaces:n {#1} }
\ExplSyntaxOff
```

6.2 Options

```
\SetupKeyvalOptions{family=pgph,prefix=pgph@}
\DeclareStringOption[remove]{selfloops}
```

```
\DeclareBoolOption{autorun}
\DeclareStringOption[dot]{engine}
\DeclareStringOption[pdf]{format}
\DeclareStringOption[LR]{rankdir}
\DeclareStringOption[usedby]{direction}
\DeclareBoolOption{cite}
\DeclareStringOption[key]{citelabel}
\DeclareBoolOption[true]{statements}
\DeclareBoolOption[true]{hyperlinks}
\DeclareStringOption{file}
\ProcessKeyvalOptions*
\ifx\pgph@file\@empty\def\pgph@file{\jobname}\fi
```

6.3 Literal braces

We need catcode-12 braces to write the Graphviz `digraph { }` block.

```
\begingroup
  \catcode`\[=1 \catcode`\]=2
  \catcode`\{=12 \catcode`\}=12
  \gdef\pgph@ob[{]%
  \gdef\pgph@cb[}]%
\endgroup
```

6.4 Data structures

A global counter gives each result instance a unique numeric id. Nodes and edges are accumulated in list macros and emitted at end of document, so that editing commands (ignore, exclude, self-loops) can be applied to the complete graph regardless of where they appear.

```
\newcount\pgph@idcnt
\newcount\pgph@proofcnt
\newcount\pgph@citeslotcnt
\def\pgph@nodes{}
\def\pgph@edges{}
\def\pgph@ignores{}
\def\pgph@excludes{}
\let\pgph@curresult\@empty
\let\pgph@curproof\@empty
```

`\pgph@auxmap` The label-to-id map. `\label` inside a result records, both in memory and in the `.aux` file, that the user label maps to the current node id, so references resolve independently of hyperref.

```
\newcommand\pgph@auxmap[2]{\global\@namedef{pgph@map@#1}{#2}%
  \ifcsname pgph@rmap@#2\endcsname\else
    \global\@namedef{pgph@rmap@#2}{#1}\fi}
\newcommand\pgph@setmap[2]{%
  \edef\pgph@tmpid{#2}%
  \global\expandafter\edef\csname pgph@map@#1\endcsname{\pgph@tmpid}%
  \ifcsname pgph@rmap@\pgph@tmpid\endcsname\else
    \global\expandafter\def\csname pgph@rmap@\pgph@tmpid\endcsname{#1}%
  \fi
  \protected@write\@auxout{}{\string\pgph@auxmap{#1}{\pgph@tmpid}}%
}
```

6.5 Registering result environments

\pgph@register A result environment is tracked by installing a begin-hook that records a node. In every declaration command (\newtheorem, \spnewtheorem) the environment *name* is the first mandatory argument, so registration does not need to parse the remaining (varied) arguments. Each name met is queued as a *candidate* (deduplicated); the begin-hooks themselves are installed only at \begin{document}, once the inclusion and exclusion lists are settled, so the customisation commands below may appear anywhere in the preamble.

```
\let\pgph@candidates\@empty
\newcommand\pgph@register[1]{%
  \edef\pgph@thisname{#1}%
  \IfBeginWith\pgph@thisname{axp@}{}{%
    \ifcsname pgph@cand@\pgph@thisname\endcsname\else
      \global\@namedef{pgph@cand@\pgph@thisname}{}%
      \ifx\pgph@candidates\@empty
        \global\let\pgph@candidates\pgph@thisname
      \else
        \xdef\pgph@candidates{\pgph@candidates,\pgph@thisname}%
      \fi
    \fi
  }%
}
```

We wrap \newtheorem (and, for Springer classes, \spnewtheorem) to register each newly declared environment, replaying the original command so its own argument parsing is untouched. The starred (unnumbered) forms are *not* auto-registered: a results graph wants numbered statements, and these forms are commonly used for unnumbered repeats of a result (e.g., the appendix copies produced by apxproof and myproofs), which would otherwise appear as spurious unnumbered duplicate nodes. Use \proofgraphtrack to track an unnumbered environment deliberately.

```
\let\pgph@orig@newtheorem\newtheorem
\renewcommand\newtheorem{\@ifstar{\pgph@nt@star}{\pgph@nt@plain}}
\newcommand\pgph@nt@plain[1]{%
  \pgph@register{#1}\pgph@orig@newtheorem{#1}}
\newcommand\pgph@nt@star[1]{\pgph@orig@newtheorem*{#1}}
\ifdefined\spnewtheorem
  \let\pgph@orig@spnewtheorem\spnewtheorem
  \renewcommand\spnewtheorem{%
    \@ifstar{\pgph@spnt@star}{\pgph@spnt@plain}}
  \newcommand\pgph@spnt@plain[1]{%
    \pgph@register{#1}\pgph@orig@spnewtheorem{#1}}
  \newcommand\pgph@spnt@star[1]{\pgph@orig@spnewtheorem*{#1}}
\fi
```

\proofgraphtrack
\proofgraphuntrack
\proofgraphtrack
\proofgraphuntrack

Result environments are detected automatically, but the analysis can be tuned. \proofgraphtrack{⟨*names*⟩} forces the listed environments to be tracked, overriding any exclusion; \proofgraphuntrack{⟨*names*⟩} excludes the listed environment *types*. Expository environments are excluded by default: definition, example, remark, note. Both commands belong in the

preamble. `\pgph@defaultenvs` is the list of class-predefined names probed at `\begin{document}`; `\pgph@defaultexclude` seeds the exclusion set.

```
\newcommand\pgph@defaultenvs{theorem,lemma,proposition,corollary,%
  definition,conjecture,claim,fact,remark,example,observation,%
  notation,problem,question,property,assumption,hypothesis,principle}
\newcommand\pgph@defaultexclude{definition,example,remark,note}
\newcommand\proofgraphtrack[1]{%
  \@for\pgph@one:=#1\do{%
    \edef\pgph@one{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
    \csundef{pgph@xtype@\pgph@one}%
    \global\@namedef{pgph@force@\pgph@one}{}%
    \expandafter\pgph@register\expandafter{\pgph@one}%
  }%
}
\newcommand\proofgraphuntrack[1]{%
  \@for\pgph@one:=#1\do{%
    \edef\pgph@one{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
    \csundef{pgph@force@\pgph@one}%
    \global\@namedef{pgph@xtype@\pgph@one}{}%
  }%
}
\expandafter\proofgraphuntrack\expandafter{\pgph@defaultexclude}
```

Environments predefined by the document class (acmart, lipics, Springer classes) are declared before the package is loaded, so the wrappers above never see them. At `\begin{document}` we add every common result name that exists to the candidate list, then install a begin-hook for each candidate that is forced or not excluded.

```
\newcommand\pgph@dohook[1]{%
  \ifcsname pgph@hooked@#1\endcsname\else
    \global\@namedef{pgph@hooked@#1}{}%
    \AtBeginEnvironment{#1}{\pgph@beginresult{#1}}%
  \fi
}
\newcommand\pgph@maybehook[1]{%
  \ifcsname pgph@force@#1\endcsname
    \pgph@dohook{#1}%
  \else
    \ifcsname pgph@xtype@#1\endcsname\else\pgph@dohook{#1}\fi
  \fi
}
\newcommand\pgph@resolveregistration[1]{%
  \@for\pgph@one:=#1\do{%
    \ifcsname\pgph@one\endcsname%
      \expandafter\pgph@register\expandafter{\pgph@one}\fi}%
  \ifx\pgph@candidates\@empty\else
    \@for\pgph@one:=\pgph@candidates\do{%
      \expandafter\pgph@maybehook\expandafter{\pgph@one}}%
  \fi
}
\AtBeginDocument{%
  \expandafter\pgph@resolveregistration\expandafter{\pgph@defaultenvs}}
```

`\pgph@beginresult` `\pgph@hooknames` `\pgph@recordresult` Start a result: allocate an id and record the node. The node is keyed and styled by

the environment name (third argument, also a fallback label), but displayed by the result's *formatted* name ("Theorem", "Lemma"), captured from the heading. We get that name by locally wrapping the heading macros, whose first two arguments are the formatted name and the number. Which macro carries the *optional note* (where a restatement's cross-reference lives) depends on the setup: amsthm, which proofgraph always loads, routes every heading through `\@begintheorem`, defined as `#1#2[#3]` with the note as the bracketed `#3` (and leaves `\@opargbegintheorem \relax`); the LaTeX kernel instead uses a two-argument `\@begintheorem` with a separate three-argument `\@opargbegintheorem` for the note; the Springer classes use `\@spbegintheorem`/`\@spopargbegintheorem`. We detect the amsthm form (`\@opargbegintheorem` is `\relax`) and read the bracketed note there, otherwise we hook `\@opargbegintheorem`; either way the note goes to `\pgph@scanrefs`. These redefinitions are confined to the current environment group, so they revert at `\end`. We also locally redefine `\label` to populate the map.

The note read as a *delimited* argument is handed back to the original macro inside braces, `[{`⟨*note*⟩`}]` rather than `[`⟨*note*⟩`]`. TeX strips the outer braces of a delimited argument that is one brace group in its entirety, so a title written `[{\cite[p.~37]{a-l}}]` – the bracing the amsthm manual prescribes to hide the `]` of the citation note – would otherwise reach the original macro unbraced, whose own scan for `]` would then stop inside the citation and take the rest of the note for text. Rebracing restores exactly what was written: the added braces are stripped again by that scan.

Nodes are accumulated in *reverse* order of appearance (`\pgph@prependnode`). With `rankdir=LR`, Graphviz stacks disconnected components vertically in the order they are read, from the bottom up; emitting the results last-to-first therefore lays them out top-to-bottom in document order (first result on top), while connected results stay grouped. This is a layout tendency, not a guarantee, but needs no `pack` trickery.

Unless `statements` is switched off, the *body* of the result is treated like the body of a proof of that result (`\pgph@beginstatement`), so that a cross-reference or a citation made in the statement itself becomes a dependency. Those assignments are local to the environment group, as the ones above are, so they revert at `\end`.

A result opened *inside* a proof – a claim stated in the course of an argument, and proved there – is a step of that proof, so `\pgph@supportedge` records the proof as depending on it: the claim would otherwise be drawn with whatever its own proof uses hanging below it and nothing above, as a component detached from the result it was stated to establish. The edge is recorded against the enclosing proof rather than a result, so it follows the same chain of owners as any other, `\proofof` and nesting included, and it passes through the ordinary filters, so `\proofgraphignore` drops it like any other. Nothing is recorded for a claim the document leaves unnumbered: `\pgph@node` excludes such a result, and the edge to it goes with it.

```
\newcommand\pgph@supportedge{%
  \ifx\pgph@encproof\@empty\else
    \pgph@addedge{\pgph@encproof}{node:\pgph@curresult}%
  \fi
}
\newcommand\pgph@prependnode[1]{%
  \xdef\pgph@nodes{\unexpanded{#1}\unexpanded\expandafter{\pgph@nodes}}%
}
```

```
\newcommand\pgph@beginresult[1]{%
  \let\pgph@encproof\pgph@curproof
  \global\advance\pgph@idcnt\@ne
  \xdef\pgph@curresult{\the\pgph@idcnt}%
  \protected@edef\pgph@tmp{%
    \noexpand\pgph@node{\pgph@curresult}{#1}{#1}}%
  \expandafter\pgph@prependnode\expandafter{\pgph@tmp}%
  \pgph@supportedge
  \pgph@hooknames
  \ifx\label\pgph@maplabel\else\let\pgph@savedlabel\label\fi
  \let\label\pgph@maplabel
  \ifpgph@statements\pgph@beginstatement\fi
}
\newcommand\pgph@hooknames{%
  \ifdefined\@begintheorem
    \let\pgph@orig@bt\@begintheorem
    \ifx\@opargbegintheorem\relax
      \def\@begintheorem##1##2[##3]{%
        \pgph@recordresult{##1}{##2}\pgph@scanrefs{##3}%
        \pgph@orig@bt{##1}{##2}[{##3}]}%
    \else
      \def\@begintheorem##1##2{%
        \pgph@recordresult{##1}{##2}\pgph@orig@bt{##1}{##2}}%
    \fi
  \fi
  \ifdefined\@opargbegintheorem\ifx\@opargbegintheorem\relax\else
    \let\pgph@orig@obt\@opargbegintheorem
    \def\@opargbegintheorem##1##2##3{%
      \pgph@recordresult{##1}{##2}\pgph@scanrefs{##3}%
      \pgph@orig@obt{##1}{##2}{##3}}%
  \fi\fi
  \ifdefined\@spbegintheorem
    \let\pgph@orig@spbt\@spbegintheorem
    \def\@spbegintheorem##1##2##3##4{%
      \pgph@recordresult{##1}{##2}\pgph@orig@spbt{##1}{##2}{##3}{##4}}%
  \fi
  \ifdefined\@spopargbegintheorem
    \let\pgph@orig@spobt\@spopargbegintheorem
    \def\@spopargbegintheorem##1##2##3##4##5{%
      \pgph@recordresult{##1}{##2}\pgph@scanrefs{##3}%
      \pgph@orig@spobt{##1}{##2}{##3}{##4}{##5}}%
  \fi
}
```

`\pgph@scanrefs` A result whose optional title carries a cross-reference, e.g., a `result` whose title is `[(Corollary~\ref{cor:x})]`, is understood to restate or build on that result: we record an edge from the current result to each label referenced in the title. We capture these by typesetting the title once into a discarded box with the capturing commands installed and `\pgph@curproof` pointing at the current result. Everything happens inside a group and `\pgph@curproof` is only *read* while the box is built, so a local assignment suffices and nothing outside the title is affected; a global one would escape the enclosing environment, which under `statements` (where `\pgph@curproof` is already set on entering the result) would leave every

subsequent reference attributed to that result.

```
\newcommand\pgph@scanrefs[1]{%
  \ifx\pgph@curresult\@empty\else
    \begingroup
      \let\pgph@curproof\pgph@curresult
      \pgph@installcapture
      \setbox\z@\hbox{#1}%
    \endgroup
  \fi
}
```

The formatted name is the first argument of every heading macro (`\@begintheorem`/`\@opargbegintheorem` for `amsthm`/`\newtheorem`, `\@spbegintheorem`/`\@spopargbegintheorem` for Springer's `\spnewtheorem`); the second argument is the number. We use that second argument only to tell *whether* the result is numbered (it is empty for the starred, unnumbered forms): an unnumbered result records no number, so it is dropped at emission unless tracked explicitly. The number *value*, when present, is read from `\@currentlabel` rather than from that argument, which some classes pollute (e.g., lipics injects `\advance\par@deathcycles\@ne`). `\@currentlabel`, set by the result's `\refstepcounter` just before the heading, is the clean number and needs no `\label`. Values are reduced to plain strings now, with fragile wrappers neutralized (notably apxproof's forward-link). The first heading wins (guarded on the name), ignoring nested headings. The emptiness of the number argument is tested without expanding it, so a polluted (but non-empty) number does not trip the test.

```
\newcommand\pgph@recordresult[2]{%
  \ifcsname pgph@name@\pgph@curresult\endcsname\else
    \begingroup
      \let\protect\@empty
      \def\axp@forward@link##1##2{##2}%
      \edef\pgph@tmpname{#1}%
      \global\expandafter\let\csname pgph@name@\pgph@curresult\endcsname
                              \pgph@tmpname
      \global\@namedef{pgph@seen@\pgph@curresult}{}%
      \def\pgph@tmphn{#2}%
      \ifx\pgph@tmphn\@empty\else
        \edef\pgph@tmpnum{\@currentlabel}%
      \global\expandafter\let\csname pgph@num@\pgph@curresult%
        \endcsname\pgph@tmpnum
      \fi
    \endgroup
  \fi
}
```

`\pgph@maplabel` records the label-to-node mapping. As a fallback for classes whose headings pass through none of the hooked heading macros (so no heading was `seen`), it also captures the number from `\@currentlabel`. We do *not* do this once a heading was seen: there, an absent number means the result is genuinely unnumbered (a starred form), and `\@currentlabel` could be a stale value from an earlier result. The value is expanded to a plain string *now*, inside a group where fragile wrappers are neutralized, so that nothing danger-

ous survives to the emission pass: in particular, under apxproof forward-linking, `\@currentlabel` holds `\axp@forward@link{`⟨*target*⟩`}{`⟨*number*⟩`}`, whose hyperlink machinery must never be expanded in a non-typesetting context (it would run away). Reducing it to its number argument keeps it harmless.

`\pgph@maplabel`

```
\newcommand\pgph@maplabel[1]{%
  \pgph@setmap{#1}{\pgph@curresult}%
  \ifcsname pgph@num@\pgph@curresult\endcsname\else
   \ifcsname pgph@seen@\pgph@curresult\endcsname\else
    \begingroup
      \let\protect\@empty
      \def\axp@forward@link##1##2{##2}%
      \edef\pgph@tmpnum{\@currentlabel}%
      \global\expandafter\let\csname pgph@num@\pgph@curresult%
        \endcsname\pgph@tmpnum
    \endgroup
   \fi
  \fi
  \pgph@savedlabel{#1}%
}
```

6.6 Hooking proofs

Inside a proof we redefine the reference commands to capture edges from the proved result.

```
\newcommand\pgph@addedge[2]{%
  \protected@edef\pgph@tmpedge{\noexpand\pgph@edge{#1}{#2}}%
  \expandafter\g@addto@macro\expandafter\pgph@edges%
    \expandafter{\pgph@tmpedge}%
}
\newcommand\pgph@recordref[1]{%
  \ifpgph@nocapture\else
  \ifx\pgph@curproof\@empty\else
    \@for\pgph@one:=#1\do{%
      \edef\pgph@k{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
      \pgph@addedge{\pgph@curproof}{\pgph@k}}%
  \fi\fi
}
```

`\ifpgph@nocapture` The capture is installed for the whole of a proof, and a proof long enough to break the page has the page furniture built while it is open. The running head is set from the marks, so a cross-reference or a citation in a sectional title is *executed again* there, in the output routine, and would be recorded as a dependency of the proof the break fell in: a document whose section title names a result gained an edge from that result to whichever result the proof belonged to. Recording is therefore suspended while the page is assembled.

The kernel neutralizes `\label`, `\index` and `\glossary` at the same point, in `\@outputpage`, and offers the hook `build/page/reset` there, inside the group that encloses the shipped box, so a local switch set from the hook covers the head and foot and is undone by itself. Where the hook is not declared (a kernel older than the hook, a class with an output routine of its own that skips it), `\@outputpage`

is wrapped instead, which needs the switch to be set globally as the group ends inside the box being shipped.

```
\newif\ifpgph@nocapture
\newcommand\pgph@wrapoutputpage{%
  \let\pgph@orig@outputpage\@outputpage
  \def\@outputpage{%
    \global\pgph@nocapturetrue
    \pgph@orig@outputpage
    \global\pgph@nocapturefalse}%
}
\AtBeginDocument{%
  \ifdefined\IfHookExistsTF
    \IfHookExistsTF{build/page/reset}%
      {\AddToHook{build/page/reset}{\pgph@nocapturetrue}}%
      {\pgph@wrapoutputpage}%
  \else
    \pgph@wrapoutputpage
  \fi
}
```

`\pgph@beginproof` Attach the proof to the current result and install the capturing versions of the reference commands (only those that exist).

The proof is not recorded as the result it proves but as a *proof instance* of its own, `p`⟨*n*⟩, whose owning result is held in `\pgph@powner@p`⟨*n*⟩ and read only at emission. An edge captured in the proof therefore does not fix its source when it is seen, which is what lets `\proofof` pin the whole proof and not merely the references that follow it. Were the source fixed at capture, a proof opening with "By Theorem 1" and naming its own result only at the end would give that first reference to whichever result happens to precede the proof. The indirection also lets a proof be pinned to a result labelled later in the document, the label being resolved with all the others when the graph is written.

The capturing commands are `\protected` so that, when a reference appears in a moving argument inside a proof (e.g., a `\caption`), they do not expand into the `.aux`/`.lof` and leak internal tokens.

```
\protected\def\pgph@cap@ref#1{\pgph@recordref{#1}\pgph@saved@ref{#1}}
\protected\def\pgph@cap@cref#1{\pgph@recordref{#1}\pgph@saved@cref{#1}}
\protected\def\pgph@cap@Cref#1{\pgph@recordref{#1}\pgph@saved@Cref{#1}}
\protected\def\pgph@cap@autoref#1{%
  \pgph@recordref{#1}\pgph@saved@autoref{#1}}
\protected\def\pgph@cap@eqref#1{%
  \pgph@recordref{#1}\pgph@saved@eqref{#1}}
\protected\def\pgph@cap@vref#1{\pgph@recordref{#1}\pgph@saved@vref{#1}}
```

zref-clever's `\zcref` is the one captured command that is not simply ⟨*cs*⟩`{`⟨*labels*⟩`}`: it is `\zcref*[`⟨*options*⟩`]{`⟨*labels*⟩`}`, and both the star and the options have to be handed on unchanged. Its mandatory argument is a comma-separated list, which `\pgph@recordref` already splits.

```
\NewDocumentCommand\pgph@cap@zcref{sO{}m}{%
  \pgph@recordref{#3}%
  \IfBooleanTF{#1}{\pgph@saved@zcref*[#2]{#3}}{\pgph@saved@zcref[#2]{#3}}%
}
\newcommand\pgph@install[1]{%
```

```
  \ifcsname #1\endcsname
    \expandafter\ifx\csname #1%
      \expandafter\endcsname\csname pgph@cap@#1\endcsname
    \else
      \expandafter\let\csname pgph@saved@#1%
        \expandafter\endcsname\csname #1\endcsname
      \expandafter\let\csname #1%
        \expandafter\endcsname\csname pgph@cap@#1\endcsname
    \fi
  \fi
}
```

The set of captured commands is needed in three places (a proof, the optional title of a result, and an apxproof deferred proof), so it is named once here: adding a command to the list must not be a matter of remembering three call sites.

```
\newcommand\pgph@installrefs{%
  \pgph@install{ref}\pgph@install{cref}\pgph@install{Cref}%
  \pgph@install{autoref}\pgph@install{eqref}\pgph@install{vref}%
  \pgph@install{zcref}%
}
```

The cross-reference commands and, when the `cite` option is on, the citation commands are always installed together, so the pair has a name of its own: every place that opens a capturing context (a proof, the optional title of a result, and, with `statements`, the body of a result) installs exactly the same set.

```
\newcommand\pgph@installcapture{%
  \pgph@installrefs
  \ifpgph@cite\pgph@installcite\fi
}
```

A proof may sit inside another one, either because the proof of a claim stated in the course of a proof is itself a `proof` environment, or because the document writes structured proofs, whose steps are proofs nested in the proof of the whole (as pf2 does). The two cases want different owners, and what tells them apart is whether a *result* has been opened since the enclosing context began: the proof of a claim follows the claim, whereas the step of a structured proof follows nothing. So each capturing context records, in `\pgph@pres@`⟨*ctx*⟩, the result current when it began; a proof opening while that result is still the current one is a *sub-proof* and takes the enclosing context as its owner, and any other proof takes the current result, as before.

Owning the enclosing context rather than a result is what lets a `\proofof` in an outer proof reach the inner ones: the pin is read where the chain of owners is followed, at emission, so it applies to a nested reference just as it does to one made directly in the pinned proof. An inner `\proofof` still pins its own proof alone, which is what the detached proof of a claim needs.

```
\newcommand\pgph@setpres[1]{%
  \global\expandafter\let\csname pgph@pres@#1\endcsname\pgph@curresult
}
\newcommand\pgph@setpowner[2]{%
  \global\expandafter\let\csname pgph@powner@#1\endcsname#2%
}
\newcommand\pgph@beginproof{%
  \global\advance\pgph@proofcnt\@ne
  \edef\pgph@newproof{p\the\pgph@proofcnt}%
```

```
  \pgph@setpres\pgph@newproof
  \edef\pgph@ncur{\pgph@curresult}%
  \edef\pgph@npres{%
    \ifcsname pgph@pres@\pgph@curproof\endcsname
      \csname pgph@pres@\pgph@curproof\endcsname
    \else\pgph@nomatch\fi}%
  \ifx\pgph@ncur\pgph@npres
    \pgph@setpowner\pgph@newproof\pgph@curproof
  \else
    \pgph@setpowner\pgph@newproof\pgph@curresult
  \fi
  \let\pgph@curproof\pgph@newproof
  \pgph@installcapture
  \pgph@hookproofhead
}
```

The sentinel stands for "no enclosing context", and must be something no result id can be, so that the comparison fails rather than matching an empty `\pgph@curresult` outside any result.

```
\newcommand\pgph@nomatch{-}
```

`\pgph@beginstatement` Under the `statements` option the body of a result captures like the body of its proof. The proof-heading hook is deliberately *not* installed here: it turns a cross-reference in an optional title into a `\proofof`, which is right for a proof titled "of Theorem 1" but not for a result titled "(restates Theorem 1)", where the reference is an ordinary dependency.

```
\newcommand\pgph@beginstatement{%
  \let\pgph@curproof\pgph@curresult
  \pgph@setpres\pgph@curproof
  \pgph@installcapture
}
```

When the `proof` environment is itself a theorem-like environment (as with Springer's `\spnewtheorem*{proof}`), its optional title is set by `\@opargbegintheorem`/`\@spopargbegintheorem`. We wrap these for the duration of the proof so that a title such as `[of Theorem~\ref{thm:x}]` sets `\proofof{thm:x}`, attributing the proof to its result. As in `\pgph@hooknames`, a `\relax \@opargbegintheorem` is left alone: that is the amsthm setup, where the macro is never called, and giving it a meaning would make `\pgph@hooknames` read a result nested in the proof as having the kernel's two-argument heading rather than amsthm's.

```
\newcommand\pgph@hookproofhead{%
  \ifdefined\@opargbegintheorem\ifx\@opargbegintheorem\relax\else
    \let\pgph@orig@obtp\@opargbegintheorem
    \def\@opargbegintheorem##1##2##3{%
      \pgph@scanproofarg{##3}\pgph@orig@obtp{##1}{##2}{##3}}%
  \fi\fi
  \ifdefined\@spopargbegintheorem
    \let\pgph@orig@spobtp\@spopargbegintheorem
    \def\@spopargbegintheorem##1##2##3##4##5{%
      \pgph@scanproofarg{##3}%
      \pgph@orig@spobtp{##1}{##2}{##3}{##4}{##5}}%
  \fi
```

```
  \ifdefined\@Opargbegintheorem
    \let\pgph@orig@Obtp\@Opargbegintheorem
    \def\@Opargbegintheorem##1##2##3##4{%
      \pgph@scanproofarg{##2}\pgph@orig@Obtp{##1}{##2}{##3}{##4}}%
  \fi
}
```

`\pgph@scanproofarg` `\pgph@wrapproof` A proof whose optional title names its result (a detached proof) has its result given by that title, e.g., `thm:x` for an optional argument `[of Theorem~\ref{thm:x}]`, rather than by the most recently opened result. We capture this by wrapping the `proof` environment so that, when an optional title is given, the cross-reference it contains is turned into a `\proofof` (the title is typeset once into a discarded box with `\ref` and friends redefined). Without an optional title the original environment is used unchanged.

```
\NewDocumentCommand\pgph@zcproofof{sO{}m}{\proofof{#3}}
\newcommand\pgph@scanproofarg[1]{%
  \begingroup
    \let\protect\@empty
    \def\ref##1{\proofof{##1}}\def\cref##1{\proofof{##1}}%
    \def\Cref##1{\proofof{##1}}\def\autoref##1{\proofof{##1}}%
    \let\zcref\pgph@zcproofof
    \setbox\z@\hbox{#1}%
  \endgroup
}
```

To capture the optional title of a detached `amsthm` proof (an optional `[of Theorem~\ref{thm:x}]`) we redefine the outer `\proof` so that, when a title is given, it is scanned and then handed to the *saved* outer macro, which parses it as it always did. Delegating to the saved outer macro rather than to the inner one means we need to know neither the name of the inner macro nor the environment's default title, so the same wrapper serves any proof environment, whatever its default. No recursion occurs: the saved meaning is captured before the redefinition.

We must wrap *only* a genuine optional-argument environment: a package that captures the proof body verbatim (apxproof, myproofs) redefines `proof` with no optional argument, and wrapping it would break that capture. We recognise the optional-argument form by the `\@protected@testopt` in its definition. When the proof environment is instead a theorem-like environment (Springer `\spnewtheorem*{proof}`, or a `\newtheorem` of one's own) its title is already handled by `\pgph@hookproofhead`, so there we just install the begin-hook.

```
\newcommand\pgph@wrapproofone[1]{%
  \@ifundefined{#1}{}{%
    \edef\pgph@proofmeaning{\expandafter\meaning\csname #1\endcsname}%
    \edef\pgph@testoptstr{\detokenize{\@protected@testopt}}%
    \IfSubStr\pgph@proofmeaning\pgph@testoptstr
      {\pgph@redefproof{#1}}%
      {\AtBeginEnvironment{#1}{\pgph@beginproof}}%
  }%
}
\newcommand\pgph@redefproof[1]{%
  \expandafter\let\csname pgph@savedenv@#1\expandafter\endcsname
    \csname #1\endcsname
  \expandafter\def\csname #1\endcsname{\pgph@beginproof
```

```
    \@ifnextchar[{\pgph@proofopt{#1}}%
                 {\csname pgph@savedenv@#1\endcsname}}%
}
\def\pgph@proofopt#1[#2]{%
  \pgph@scanproofarg{#2}\csname pgph@savedenv@#1\endcsname[{#2}]}
```

\proofgraphproofenv

\proofgraphproofenv
\pgph@proofenvs

Which environments are proofs is a list, `proof` to begin with; `\proofgraphproofenv{`⟨*names*⟩`}` adds to it, for a document whose proofs live in an environment of another name (pf2's structured proofs under a name that avoids the clash with amsthm, a `\newtheorem{claimproof}` for the proofs of claims…). A proof is not a result, so the names are excluded from result tracking as well; `\proofgraphtrack` still overrides that for an environment one wants drawn as both.

```
\newcommand\pgph@proofenvs{proof}
\newcommand\proofgraphproofenv[1]{%
  \g@addto@macro\pgph@proofenvs{,#1}%
  \proofgraphuntrack{#1}%
}
```

Packages such as apxproof capture the body of a `proof` verbatim (to defer it to the appendix) and replay it there. Patching the `proof` environment then corrupts that capture, and the references are only executed at replay time anyway. So we wrap `proof` only when no such package is active; otherwise the deferred-proof capture (below) takes over. The exemption is for `proof` alone: an environment added with `\proofgraphproofenv` is not one apxproof defers, so it is wrapped as usual. This happens at `\begin{document}`, once we know what is loaded and what the preamble has added to the list.

```
\newif\ifpgph@apxproof
\newcommand\pgph@wrapproofs{%
  \@for\pgph@one:=\pgph@proofenvs\do{%
    \edef\pgph@pe{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
    \ifboolexpr{ bool {pgph@apxproof} and test {\ifdefstring{\pgph@pe}{proof}} }%
      {}{\expandafter\pgph@wrapproofone\expandafter{\pgph@pe}}%
  }%
}
\AtBeginDocument{%
  \@ifpackageloaded{apxproof}{\pgph@apxprooftrue\pgph@apxinit}{}%
  \pgph@wrapproofs
}
```

\pgph@apxinit
\pgph@apxproofcapture

Under apxproof, deferred proofs are typeset in the appendix, where their references finally execute. apxproof fires `\apxproofhook` inside each such proof, passing the `axp@r`⟨*n*⟩ label of the proved theorem, which (because apxproof also attaches that label to the theorem in the main text) is already in our map. We resolve it to the source node and install the reference-capturing commands for the duration of the proof.

```
\newcommand\pgph@apxinit{%
  \def\apxproofhook##1{\pgph@apxproofcapture{##1}}}
\newcommand\pgph@apxproofcapture[1]{%
  \edef\pgph@curproof{\pgph@resolve{#1}}%
  \ifx\pgph@curproof\@empty\else
```

```
    \pgph@setpres\pgph@curproof
    \pgph@installcapture
  \fi
}
```

`\pgph@hookcite` Optional citation capture, applied to `\cite` and, when they are defined (natbib, biblatex), to `\citet` and `\citep` as well, all sharing one capture chain parameterized by the command name: external references become `cite:`-prefixed target labels, drawn later as distinct nodes. The node identity is the pair (key, note): a `\cite[`⟨*note*⟩`]{`⟨*key*⟩`}` inside a proof is distinguished from a `\cite` of the same key with a *different* note, so that citing two different results of the same work (say two theorems of one book) yields two nodes rather than one mislabelled with whichever note was seen first. Each pair is assigned a numeric *slot* (`\pgph@citeslot`); the edge target is `cite:`⟨*slot*⟩, and the slot remembers the key (for the tag and the bibliography hyperlink) and the note (for the label). The note is reduced to a plain string at capture time (fragile wrappers neutralized, `~` normalised to a space), which also serves as the signature so `[Thm 1]` and `[Thm~1]` coincide. We peek for the optional argument with `\@ifnextchar` rather than declaring one, so that a note-less `\cite{`⟨*key*⟩`}` is passed on *as written*: re-emitting it as `\cite[]{`⟨*key*⟩`}` would make the typeset citation grow a spurious empty note ("[1,]"). A second optional argument is peeked for in the same way, to support the two-argument form `\cite[`⟨*pre-note*⟩`][`⟨*note*⟩`]{`⟨*key*⟩`}` of natbib and biblatex; there the *second* argument is the note recorded for the node (matching the one-argument semantics, where the lone argument is the post-note), while the pre-note ("see", "e.g.") qualifies the act of citing rather than identifying the cited result, and is only replayed to the saved `\cite`. Without this peek, the mandatory-argument scan would grab the bare token `[` as the key and typeset the rest of the citation as text, giving confusing errors ("Missing $ inserted") deep in the proof body. A star (`\citet*`, `\citep*`: full author list) is likewise peeked for and replayed; it affects rendering only, not the recorded (key, note) pair. As in `\pgph@hooknames`, the notes are replayed braced, `[{`⟨*note*⟩`}]`, so that a note written `[{p.~37, [sic]}]` to hide a `]` reaches the saved command as written rather than cut short at that `]`. The replay head (saved command plus optional star) is frozen in `\pgph@replay` by the star-peek before the arguments are scanned. As in `\pgph@install`, installation is skipped (per command) when the command is already the capture macro; otherwise a proof nested inside another proof would save the capture as the saved command, and the first citation in the inner proof would recurse forever.

```
\newcommand\pgph@gxdefcs[2]{%
  \global\expandafter\edef\csname#1\endcsname{#2}}
\newcommand\pgph@cap@citegen[1]{%
  \def\pgph@citecmd{#1}%
  \@ifstar{\pgph@cap@cite@x*}{\pgph@cap@cite@x{}}}
\def\pgph@cap@cite@x#1{%
  \edef\pgph@replay{%
    \expandafter\noexpand\csname pgph@saved@\pgph@citecmd\endcsname#1}%
  \@ifnextchar[\pgph@cap@cite@opt\pgph@cap@cite@noopt}
\def\pgph@cap@cite@opt[#1]{%
  \@ifnextchar[{\pgph@cap@cite@optii{#1}}{\pgph@cap@cite@opti{#1}}}
\def\pgph@cap@cite@opti#1#2{%
  \pgph@cap@citerec{#1}{#2}\pgph@replay[{#1}]{#2}}
\def\pgph@cap@cite@optii#1[#2]#3{%
```

```
  \pgph@cap@citerec{#2}{#3}\pgph@replay[{#1}][{#2}]{#3}}
\def\pgph@cap@cite@noopt#1{\pgph@cap@citerec{}{#1}\pgph@replay{#1}}
\newcommand\pgph@cap@citerec[2]{%
  \ifpgph@nocapture\else
  \ifx\pgph@curproof\@empty\else
    \@for\pgph@one:=#2\do{%
      \edef\pgph@k{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
      \pgph@citeslot{\pgph@k}{#1}%
      \pgph@addedge{\pgph@curproof}{cite:\pgph@curslot}}%
  \fi\fi
}
\newcommand\pgph@citeslot[2]{%
  \begingroup
    \pgph@citesanitize
    \xdef\pgph@gnotestr{#2}%
  \endgroup
  \edef\pgph@sig{#1@@\detokenize\expandafter{\pgph@gnotestr}}%
  \ifcsname pgph@cslot@\pgph@sig\endcsname
    \edef\pgph@curslot{\csname pgph@cslot@\pgph@sig\endcsname}%
  \else
    \global\advance\pgph@citeslotcnt\@ne
    \edef\pgph@curslot{\the\pgph@citeslotcnt}%
    \pgph@gxdefcs{pgph@cslot@\pgph@sig}{\pgph@curslot}%
    \pgph@gxdefcs{pgph@cskey@\pgph@curslot}{#1}%
    \ifx\pgph@gnotestr\@empty\else
      \pgph@gxdefcs{pgph@citenote@\pgph@curslot}{\pgph@gnotestr}%
    \fi
  \fi
}
```

`\pgph@citecmds` `\proofgraphcitecommand` `\pgph@installcite`

Every citation command with the signature `\cs*[`⟨*pre*⟩`][`⟨*post*⟩`] {`⟨*keys*⟩`}` is captured by the same `\pgph@cap@citegen`, so which commands are captured is a list rather than code. The default list holds the citation commands of LaTeX, natbib and biblatex that cite a work; `\proofgraphcitecommand` adds more. Naming a command that the document never defines costs nothing, as `\pgph@installciteone` installs only over an existing command, so the list may name every spelling a citation package might provide.

The commands printing only a *fragment* of a citation (`\citeauthor`, `\citeyear`...) are in the list as well. They commonly accompany a real citation of the same work, but a repetition is harmless: a (key, note) pair already seen reuses its slot, and `\pgph@writeedge` draws an edge once, so the second mention of a work adds nothing to the graph. Capturing them means that a proof invoking a work by author or by year alone, which does happen, is not silently left without its edge.

The biblatex *multicite* commands (`\cites`, `\parencites`, `\textcites`) cannot be captured at all: their `(`⟨*pre*⟩`)(`⟨*post*⟩`)` syntax and repeated key groups are a different grammar, which `\pgph@cap@citegen` does not parse.

```
\newcommand\pgph@citecmds{%
  cite,Cite,citet,Citet,citep,Citep,%
  citealt,Citealt,citealp,Citealp,%
  parencite,Parencite,textcite,Textcite,autocite,Autocite,%
  footcite,Footcite,footcitetext,smartcite,Smartcite,supercite,%
  fullcite,footfullcite,%
```

```
  citeauthor,Citeauthor,citefullauthor,citetitle,Citetitle,%
  citeyear,Citeyear,citeyearpar,citedate,Citedate,citeurl,citenum}
\newcommand\proofgraphcitecommand[1]{\g@addto@macro\pgph@citecmds{,#1}}
\newcommand\pgph@installcite{%
  \@for\pgph@conename:=\pgph@citecmds\do{%
    \edef\pgph@cone{\expandafter\pgph@trimsp\expandafter{\pgph@conename}}%
    \expandafter\pgph@installciteone\expandafter{\pgph@cone}}%
}
```

The capture macro of a command is built on first use rather than declared: it is the same `\pgph@cap@citegen` call for all of them, and building it only for commands the document actually defines keeps the unused names of the list from costing anything. It must be `\gdef`, the installation happening inside the group of a proof.

```
\newcommand\pgph@installciteone[1]{%
  \ifcsname #1\endcsname
    \ifcsname pgph@cap@#1\endcsname\else
      \expandafter\gdef\csname pgph@cap@#1\endcsname{\pgph@cap@citegen{#1}}%
    \fi
    \expandafter\ifx\csname #1\expandafter\endcsname
                    \csname pgph@cap@#1\endcsname
    \else
      \expandafter\let\csname pgph@saved@#1\expandafter\endcsname
                      \csname #1\endcsname
      \expandafter\let\csname #1\expandafter\endcsname
                      \csname pgph@cap@#1\endcsname
    \fi
  \fi
}
```

The biblatex recorder is registered at the end of the preamble, once biblatex has certainly been loaded (it is a preamble-only package) but while `\AtEveryCitekey` still accepts additions. It runs both at every citation and at every bibliography entry, so that a work only `\nocite`d, or reached by `\usescite` alone, is still seen. It is not conditioned on the `cite` option, which governs the automatic capture of `\cite` only: `\usescite` and `\proofgraphciteedge` draw citation nodes without it, and those nodes want their tag and their anchor just the same.

```
\newif\ifpgph@blx
\AtEndPreamble{%
  \@ifpackageloaded{biblatex}{%
    \pgph@blxtrue
    \AtEveryCitekey{\pgph@blxrecord}%
    \AtEveryBibitem{\pgph@blxrecord}%
  }{}%
}
```

6.7 User commands

```
\newcommand\uses[1]{\pgph@recordref{#1}}
```

`\proofof` names the result a proof establishes. Inside a proof instance it records the label against the instance, so that the pin covers the whole proof, wherever in it the command stands, and is resolved (with every other label) when the graph is written. Outside one – in an apxproof deferred proof, whose owning result the

\apxproofhook already gives, or in the body of a result under statements – there is no instance to pin, and the source is redirected on the spot instead.

```
\newcommand\proofof[1]{%
  \ifcsname pgph@powner@\pgph@curproof\endcsname
    \expandafter\gdef\csname pgph@pofl@\pgph@curproof\endcsname{#1}%
  \else
    \ifcsname pgph@map@#1\endcsname
      \xdef\pgph@curproof{\csname pgph@map@#1\endcsname}%
    \fi
  \fi
}
\newcommand\proofgraphedge[2]{%
  \g@addto@macro\pgph@edges{\pgph@labeledge{#1}{#2}}%
}
\newcommand\pgph@labeledge[2]{%
  \edef\pgph@ef{\pgph@resolve{#1}}%
  \ifx\pgph@ef\@empty\else
    \@for\pgph@one:=#2\do{%
      \edef\pgph@k{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
      \pgph@plainedge{\pgph@ef}{\pgph@k}}%
  \fi
}
```

\usescite
\proofgraphciteedge
\proofgraphciteedge
\pgph@citelabeledge

The two manual counterparts of the automatic \cite capture, for a dependency on external work that no citation in a proof expresses: \usescite, the \uses of citation nodes, records one from inside a proof, and \proofgraphciteedge, the \proofgraphedge of citation nodes, states one from anywhere. Both create the citation node if the work has no node yet, take a comma-separated list of BibTeX keys and accept as an optional argument the note that would have been the optional argument of \cite ([Thm~3]), which labels the node and, as in the automatic capture, distinguishes one result of a work from another.

Being explicit, they do not depend on the cite option, which only says whether a plain \cite is captured on its own. \usescite shares the recording path of the captured citation commands, so a work reached both ways still yields a single node.

```
\NewDocumentCommand\usescite{O{}m}{\pgph@cap@citerec{#1}{#2}}
\NewDocumentCommand\proofgraphciteedge{O{}mm}{%
  \g@addto@macro\pgph@edges{\pgph@citelabeledge{#1}{#2}{#3}}%
}
```

Resolution is deferred to emission, as for \proofgraphedge, so the result may be labelled anywhere in the document; the slot is therefore allocated then, which is harmless as slot numbers only have to be distinct. The edge is assembled by \edef rather than passed as macros, because \pgph@citeedge takes the cite:⟨*slot*⟩ target apart with xstring, which reads its argument literally.

```
\newcommand\pgph@citelabeledge[3]{%
  \edef\pgph@ef{\pgph@resolve{#2}}%
  \ifx\pgph@ef\@empty\else
    \@for\pgph@one:=#3\do{%
      \edef\pgph@k{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
      \pgph@citeslot{\pgph@k}{#1}%
      \edef\pgph@tmpce{%
```

```
        \noexpand\pgph@citeedge{\pgph@ef}{cite:\pgph@curslot}}%
      \pgph@tmpce}%
  \fi
}
```

\proofgraphignore
\proofgraphignorecite
\proofgraphexclude

The editing rules. Each undoes one of the recording commands and takes the arguments of the command it undoes: `\proofgraphignore` those of `\proofgraphedge`, the result first and what its proof uses second, and `\proofgraphignorecite` those of `\proofgraphciteedge`, the result and then the keys, with the note of the citation as an optional argument. So a rule is written by copying the dependency it is to remove, not by reading the arrow off the picture, which with the default `direction=usedby` runs the other way. The `direction` option is not consulted here at all: the filters work on the relation recorded, and `direction` settles only how it is drawn.

```
\newcommand\proofgraphignore[2]{%
  \g@addto@macro\pgph@ignores{\pgph@ignorerule{#1}{#2}}%
}
\NewDocumentCommand\proofgraphignorecite{O{}mm}{%
  \g@addto@macro\pgph@ignores{\pgph@citeignorerule{#1}{#2}{#3}}%
}
\newcommand\proofgraphexclude[1]{%
  \g@addto@macro\pgph@excludes{\pgph@excluderule{#1}}%
}

\newcommand\proofgraphstyle[2]{\global\@namedef{pgph@style@#1}{#2}}
```

External citation nodes (option `cite`) share one style, held in `\pgph@citestyle` and overridable with `\proofgraphstylecite`; it defaults to `shape=note`.

```
\newcommand\pgph@citestyle{shape=note}
\newcommand\proofgraphstylecite[1]{\renewcommand\pgph@citestyle{#1}}
```

6.8 Resolution helpers

These run at end of document, after the `.aux` has populated the map.

```
\newcommand\pgph@resolve[1]{%
  \ifcsname pgph@map@#1\endcsname\csname pgph@map@#1\endcsname\fi
}
% The key under which a filter is consulted is that of the relation recorded,
% whose source is the \emph{result}; the two names below say which end is which,
% so that reading a rule is not a matter of remembering an order.
%    \begin{macrocode}
\newcommand\pgph@ignorerule[2]{%
  \edef\pgph@ires{\pgph@resolve{#1}}\edef\pgph@idep{\pgph@resolve{#2}}%
  \ifx\pgph@ires\@empty\else\ifx\pgph@idep\@empty\else
    \@namedef{pgph@ig@\pgph@ires @\pgph@idep}{}%
  \fi\fi
}
```

A citation rule names the work depended on by key and note, the pair a citation node stands for, so it has to go through the slot allocator to learn which node that is. Asking for the slot of a work no proof cites allocates one, which costs nothing: slots become nodes only when an edge reaches them.

```
\newcommand\pgph@citeignorerule[3]{%
```

```
  \edef\pgph@ires{\pgph@resolve{#2}}%
  \ifx\pgph@ires\@empty\else
    \@for\pgph@one:=#3\do{%
      \edef\pgph@k{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
      \pgph@citeslot{\pgph@k}{#1}%
      \@namedef{pgph@igc@\pgph@ires @\pgph@curslot}{}}%
  \fi
}
\newcommand\pgph@excluderule[1]{%
  \edef\pgph@x{\pgph@resolve{#1}}%
  \ifx\pgph@x\@empty\else\@namedef{pgph@ex@\pgph@x}{}\fi
}
```

`\pgph@checkrules` An editing rule that matches nothing does nothing, and says nothing: the commonest way to get one is to write the two labels of `\proofgraphignore` in the order of the arrow drawn rather than of the dependency recorded, which leaves the graph exactly as it was, with no sign that a rule was even read. So the rules are replayed once the edges have been written, against the relations the document turned out to record, and each rule that met none is reported. When it is the swapped rule that would have matched, the report says so and gives the line to write, that being the mistake it almost always is.

The replay redefines the three rule macros and runs the same token lists again, so a rule cannot be checked in a way that differs from how it was applied, and adding a kind of rule cannot leave the check behind.

```
\newcommand\pgph@checkrules{%
  \let\pgph@ignorerule\pgph@ignorecheck
  \let\pgph@citeignorerule\pgph@citeignorecheck
  \let\pgph@excluderule\pgph@excludecheck
  \pgph@ignores
  \pgph@excludes
}
\newcommand\pgph@nolabelwarn[2]{%
  \PackageWarning{proofgraph}{%
    No such result in the graph:\MessageBreak
    \string#1\space names `#2', which is not\MessageBreak
    the label of any result drawn}%
}
\newcommand\pgph@ignorecheck[2]{%
  \edef\pgph@ires{\pgph@resolve{#1}}\edef\pgph@idep{\pgph@resolve{#2}}%
  \ifx\pgph@ires\@empty
    \pgph@nolabelwarn\proofgraphignore{#1}%
  \else\ifx\pgph@idep\@empty
    \pgph@nolabelwarn\proofgraphignore{#2}%
  \else\ifcsname pgph@rel@\pgph@ires @\pgph@idep\endcsname\else
    \ifcsname pgph@rel@\pgph@idep @\pgph@ires\endcsname
      \def\pgph@hint{%
        it is `#2' that depends on `#1'.\MessageBreak
        The two labels go the way the dependency is\MessageBreak
        recorded, the result first, and not the way\MessageBreak
        the arrow is drawn. Write instead\MessageBreak
        \string\proofgraphignore{#2}{#1}}%
    \else
      \def\pgph@hint{%
```

```
        nothing records that ‘#1’ depends on ‘#2’}%
    \fi
    \PackageWarning{proofgraph}{%
      No edge suppressed by\MessageBreak
      \string\proofgraphignore{#1}{#2}:\MessageBreak
      \pgph@hint}%
  \fi\fi\fi
}
\newcommand\pgph@citeignorecheck[3]{%
  \edef\pgph@ires{\pgph@resolve{#2}}%
  \ifx\pgph@ires\@empty
    \pgph@nolabelwarn\proofgraphignorecite{#2}%
  \else
    \edef\pgph@tmpnote{\detokenize{#1}}%
    \ifx\pgph@tmpnote\@empty
      \def\pgph@hint{}%
    \else
      \def\pgph@hint{\space with the note given}%
    \fi
    \@for\pgph@one:=#3\do{%
      \edef\pgph@k{\expandafter\pgph@trimsp\expandafter{\pgph@one}}%
      \pgph@citeslot{\pgph@k}{#1}%
      \ifcsname pgph@relc@\pgph@ires @\pgph@curslot\endcsname\else
        \PackageWarning{proofgraph}{%
          No edge suppressed by\MessageBreak
          \string\proofgraphignorecite{#2}{\pgph@k}:\MessageBreak
          nothing records that ‘#2’ cites that work\pgph@hint}%
      \fi}%
  \fi
}
\newcommand\pgph@excludecheck[1]{%
  \edef\pgph@x{\pgph@resolve{#1}}%
  \ifx\pgph@x\@empty\pgph@nolabelwarn\proofgraphexclude{#1}\fi
}
```

6.9 Emission

`\pgph@dotesc` Escape the two characters that are special in a Graphviz double-quoted string, so a theorem name, number or citation note containing them does not corrupt the `.dot`: a backslash is doubled and a double quote is backslash-escaped (in that order). `\pgph@bschar` and `\pgph@dqchar` hold the catcode-12 backslash and double quote to search for; xstring's expansion mode is saved and restored around the two substitutions.

```
\begingroup
  \catcode‘\|=0 \catcode‘\"=12 \catcode‘\\=12
  |gdef|pgph@bschar{\}%
  |gdef|pgph@dqchar{"}%
|endgroup
\newcommand\pgph@dotesc[1]{%
  \saveexpandmode\expandarg
  \StrSubstitute{#1}{\pgph@bschar}{\pgph@bschar\pgph@bschar}[#1]%
  \StrSubstitute{#1}{\pgph@dqchar}{\pgph@bschar\pgph@dqchar}[#1]%
  \restoreexpandmode}
```

`\pgph@node` Emit one node line, honouring exclusions and per-environment styling. External (`cite:`) targets are emitted separately as a pass over edges. An *unnumbered* result (no number captured) is omitted: a results graph is about numbered statements, and unnumbered theorem-like environments are typically expository or repeated copies (e.g., a class-predefined `\spnewtheorem*{claim}`). `\proofgraphtrack{`⟨*env*⟩`}` overrides this, so a deliberately unnumbered environment is still drawn.

```
\newcommand\pgph@node[3]{%
  \ifcsname pgph@ex@#1\endcsname\else
    \edef\pgph@nm{\ifcsname pgph@num@#1\endcsname%
      \csname pgph@num@#1\endcsname\fi}%
    \ifx\pgph@nm\@empty
      \ifcsname pgph@force@#2\endcsname
        \pgph@emitnode{#1}{#2}{#3}%
      \else
        \@namedef{pgph@ex@#1}{}%
      \fi
    \else
      \pgph@emitnode{#1}{#2}{#3}%
    \fi
  \fi
}
\newcommand\pgph@emitnode[3]{%
  \edef\pgph@s{\ifcsname pgph@style@#2\endcsname
    , \csname pgph@style@#2\endcsname\fi}%
  \edef\pgph@dn{\ifcsname pgph@name@#1\endcsname
    \csname pgph@name@#1\endcsname\else#3\fi}%
  \edef\pgph@lbl{\pgph@dn\space\pgph@nm}%
  \pgph@dotesc\pgph@lbl
  \immediate\write\pgph@out{%
    \space\space"n#1" [label="\pgph@lbl"\pgph@s];}%
  \ifcsname pgph@rmap@#1\endcsname
    \immediate\write\pgph@mapout{%
      \string\pgphnodemap{n#1}{\csname pgph@rmap@#1\endcsname}}%
  \fi
}
```

`\pgph@edge` `\pgph@setsrc` Emit one edge, resolving both ends and applying the self-loop, ignore and exclude filters. Targets beginning with `cite:` denote external nodes.

The source is resolved by `\pgph@setsrc`. An edge captured in a proof has that proof instance as its source, and the result it belongs to is decided here: the label `\proofof` pinned the proof to, when there is one and it resolves, and otherwise the result the proof followed. An unresolvable pin falls back on that result rather than dropping the edge, so a mistyped or never-labelled `\proofof` loses the attribution but not the dependency. An edge recorded outside a proof (the body or the title of a result, an apxproof deferred proof) already carries a result id, which is its own source.

A sub-proof owns not a result but the proof it is nested in, so the owner is followed as a chain: at each step the pin of the instance reached wins if it resolves, and otherwise the walk goes on to its owner. The nearest pin therefore decides, an inner `\proofof` overriding an outer one for the references of the inner proof alone, and an outer one covering every nested reference no inner pin claims. The

walk always terminates: an owner is either an instance opened strictly earlier or a result id, which owns nothing.

```
\newcommand\pgph@setsrc[1]{%
  \edef\pgph@esrc{#1}%
  \pgph@srcwalk
}
\newcommand\pgph@srcwalk{%
  \ifcsname pgph@powner@\pgph@esrc\endcsname
    \let\pgph@srcnext\pgph@srcstep
  \else
    \let\pgph@srcnext\relax
  \fi
  \pgph@srcnext
}
\newcommand\pgph@srcstep{%
  \let\pgph@epin\@empty
  \ifcsname pgph@pofl@\pgph@esrc\endcsname
    \edef\pgph@epin{\csname pgph@pofl@\pgph@esrc\endcsname}%
    \edef\pgph@epin{\pgph@resolve{\pgph@epin}}%
  \fi
  \ifx\pgph@epin\@empty
    \edef\pgph@esrc{\csname pgph@powner@\pgph@esrc\endcsname}%
    \pgph@srcwalk
  \else
    \let\pgph@esrc\pgph@epin
  \fi
}
\newcommand\pgph@edge[2]{%
  \pgph@setsrc{#1}%
  \ifx\pgph@esrc\@empty\else
    \expandafter\pgph@edgeone\expandafter{\pgph@esrc}{#2}%
  \fi
}
\newcommand\pgph@edgeone[2]{%
  \IfBeginWith{#2}{cite:}%
    {\pgph@citeedge{#1}{#2}}%
    {\IfBeginWith{#2}{node:}%
      {\StrBehind{#2}{node:}[\pgph@ntgt]%
       \pgph@plainedgeid{#1}{\pgph@ntgt}}%
      {\pgph@plainedge{#1}{#2}}}%
}
```

A target is normally a user label, resolved here; a `node:` one is a result id already, recorded by `\pgph@supportedge`, which knows the result it points at without its having to be labelled at all. Past that, the two are the same edge and take the same filters.

```
\newcommand\pgph@plainedge[2]{%
  \edef\pgph@ptgt{\pgph@resolve{#2}}%
  \ifx\pgph@ptgt\@empty\else
    \pgph@plainedgeid{#1}{\pgph@ptgt}%
  \fi
}
\newcommand\pgph@plainedgeid[2]{%
  \edef\pgph@t{#2}%
```

```
  \@namedef{pgph@rel@#1@\pgph@t}{}%
  \ifcsname pgph@ex@#1\endcsname\else
  \ifcsname pgph@ex@\pgph@t\endcsname\else
  \ifcsname pgph@ig@#1@\pgph@t\endcsname\else
    \ifboolexpr{ test {\ifnumequal{#1}{\pgph@t}}
                 and test {\ifdefstring{\pgph@selfloops}{remove}} }%
      {}{\pgph@emitedge{n#1}{n\pgph@t}}%
  \fi\fi\fi
}
```

`\pgph@emitedge` `\pgph@writeedge` The relation recorded is always "the result whose proof we are in (the first argument) uses the target (the second)". The `direction` option chooses the arrow orientation: `direction=usedby` (default) draws *target to result*, so an arrow `a->b` reads "`a` is used by `b`" (i.e. `b` relies on `a`); `direction=uses` draws *result to target* ("`a` uses `b`"). The filters above work on the recorded relation, so they are unaffected by the chosen orientation. `\pgph@writeedge` deduplicates.

```
\newcommand\pgph@emitedge[2]{%
  \ifdefstring{\pgph@direction}{uses}%
    {\pgph@writeedge{#1}{#2}}%
    {\pgph@writeedge{#2}{#1}}%
}
\newcommand\pgph@writeedge[2]{%
  \ifcsname pgph@drawn@#1@#2\endcsname\else
    \@namedef{pgph@drawn@#1@#2}{}%
    \immediate\write\pgph@out{\space\space"#1" -> "#2";}%
  \fi
}
```

`\pgph@citeedge` `\pgph@setcitelabel` A `cite:`⟨*slot*⟩ target: recover the slot's BibTeX key, declare the external node once per slot, then draw the edge. The node *label* is built by `\pgph@setcitelabel`: it is the key by default, or, with `citelabel=tag`, the printed citation tag (the "[42]"/"[Knu74]" text), recovered either from a tag recorded by `\pgph@blxrecord` under biblatex or from the `\b@`⟨*key*⟩ macro that `\bibcite` writes to the `.aux` (so two runs are needed, and it falls back to the key when no tag is known, e.g., on the first run or with citation packages neither path covers). The slot's `\cite` note, when present, is added inside the tag. The node-to-label map records the *anchor* of the cited work, so all slots of one work hyperlink to the same bibliography entry.

```
\newcommand\pgph@citeedge[2]{%
  \StrBehind{#2}{cite:}[\pgph@slot]%
  \@namedef{pgph@relc@#1@\pgph@slot}{}%
  \ifcsname pgph@igc@#1@\pgph@slot\endcsname\else
  \ifcsname pgph@ex@#1\endcsname\else
    \edef\pgph@key{\csname pgph@cskey@\pgph@slot\endcsname}%
    \ifcsname pgph@extnode@\pgph@slot\endcsname\else
      \@namedef{pgph@extnode@\pgph@slot}{}%
      \expandafter\pgph@setcitelabel\expandafter{\pgph@slot}%
      \pgph@dotesc\pgph@clabel
      \immediate\write\pgph@out{%
        \space\space"c@\pgph@slot"
        [label="\pgph@clabel",\pgph@citestyle];}%
      \immediate\write\pgph@mapout{%
        \string\pgphnodemapcite{c@\pgph@slot}{\pgph@citeanchor{\pgph@key}}}%
    \fi
```

```
    \pgph@emitedge{n#1}{c@\pgph@slot}%
  \fi\fi
}
\edef\pgph@obrace{\expandafter\@gobble\string\{}
\newif\ifpgph@natnum
\newcommand\pgph@nil{}
```

`\pgph@citesanitize` neutralizes the wrappers and spacing cruft that citation labels carry, so that an `\edef` of `\b@`⟨*key*⟩ yields plain text: hyperref's `\hyper@@link` (keep its printed last argument) and the boxing/penalty/`\ignorespaces` noise that natbib bakes into author lists.

```
\newcommand\pgph@citesanitize{%
  \let\protect\@empty
  \def\hyper@@link[##1]##2##3##4{##4}%
  \let\unskip\@empty \let\ignorespaces\@empty \let\unhbox\@empty
  \let\penalty\@gobble \let\@M\@empty \let\voidb@x\@empty
  \let\nobreakspace\space \let~\space
  \def\hbox##1{##1}\def\mbox##1{##1}%
}
```

A natbib `\b@`⟨*key*⟩ is `{`⟨*num*⟩`}{`⟨*year*⟩`}{ `⟨*short*⟩`}{`⟨*long*⟩`}`. In numbers mode the tag is the number; in author–year mode it is the short author list and the year.

```
\def\pgph@natpick#1#2#3#4\pgph@nil{%
  \ifpgph@natnum\def\pgph@x{#1}\else\def\pgph@x{#3 #2}\fi}
\newcommand\pgph@natdo{\expandafter\pgph@natpick\pgph@x{}{}{}\pgph@nil}
```

`\pgph@natnumset` Which of the two natbib modes is in force is read off `\ifNAT@numbers`, by comparing it with `\iftrue`. That comparison *must* live in a macro of its own rather than inline in `\pgph@setcitelabel`, because the code around it can be skipped by a false conditional: skipping does not expand anything, it merely counts conditional tokens, and the `\iftrue` here – an operand of `\ifx`, not a conditional – would be counted as one more `\if` to be closed. The skip would then eat one `\fi` too many and run past the end of `\pgph@setcitelabel` into its caller, silently swallowing the very writes that emit the citation node. Inside a macro the token is never seen by the skipping scanner.

```
\newcommand\pgph@natnumset{%
  \pgph@natnumfalse
  \@ifundefined{ifNAT@numbers}{}{%
    \expandafter\ifx\csname ifNAT@numbers\endcsname\iftrue
      \pgph@natnumtrue\fi}%
}
```

`\pgph@blxrecord` biblatex writes no `\b@`⟨*key*⟩: its tags are assembled while the citation is typeset, from data the `.bbl` holds. We therefore record them as they go by, from `\AtEveryCitekey` and `\AtEveryBibitem` (installed below), where the entry is the current one and its fields are readable. `labelnumber` carries the tag of the numeric styles and `labelalpha` that of the alphabetic ones; the author–year styles have neither, as their tag is a formatted name list rather than a field, and are left to the `key` fallback. A field that survives sanitizing with a backslash still in it is not plain text, and is likewise declined.

The same visit records the entry's *anchor*, the name of the hyperref destination its bibliography entry carries, which biblatex does not spell `cite.`⟨*key*⟩ as the LaTeX kernel and natbib do but `cite.`⟨*refsection*⟩`@`⟨*key*⟩, the reference section being part

of the name because one key may have an entry in each. Naming the destination wrongly is not a visible failure but a misleading one: hyperref sends an unknown destination to the end of the document, so the node would link somewhere rather than nowhere. The anchor is recorded whatever `citelabel` says, the hyperlink being drawn either way.

```
\newcommand\pgph@blxrecord{%
  \begingroup
    \pgph@citesanitize
    \edef\pgph@blxkey{\thefield{entrykey}}%
    \pgph@gxdefcs{pgph@blxanchor@\pgph@blxkey}%
      {\the\c@refsection @\pgph@blxkey}%
    \ifdefstring{\pgph@citelabel}{tag}{%
      \edef\pgph@x{\thefield{labelnumber}}%
      \ifx\pgph@x\@empty \edef\pgph@x{\thefield{labelalpha}}\fi
      \edef\pgph@dx{\detokenize\expandafter{\pgph@x}}%
      \ifx\pgph@x\@empty\else
        \IfSubStr\pgph@dx\@backslashchar{}{%
          \pgph@gxdefcs{pgph@blxtag@\pgph@blxkey}{\pgph@x}}%
      \fi
    }{}%
  \endgroup
}
```

The anchor of a key never met as a citation nor as a bibliography entry is not known; under biblatex we then assume the first reference section, which is the only one most documents have. A key absent from the bibliography has no destination under any name, so nothing is lost when the guess is wrong.

```
\newcommand\pgph@citeanchor[1]{%
  \ifcsname pgph@blxanchor@#1\endcsname
    \csname pgph@blxanchor@#1\endcsname
  \else
    \ifpgph@blx 0@\fi#1%
  \fi
}
```

A note is written the way a citation with a note is printed, inside the tag and after it: `[Knu68, pp. 23-27]`, not the tag after the note. Its dashes are the two ligatures LaTeX reads as dashes, `--` and `---`, which mean nothing to Graphviz; they are turned into the HTML character entities Graphviz decodes in an ordinary label, so that a page range set as `[pp.~23--27]` carries the same dash in the graph as in the document. The entities are built with `&` of category 12, as the escapes of `\pgph@dotesc` are, so that nothing in the label depends on the category codes in force where the graph is written.

```
\begingroup
  \catcode`\&=12
  \gdef\pgph@endash{&ndash;}%
  \gdef\pgph@emdash{&mdash;}%
\endgroup
\newcommand\pgph@dashesc[1]{%
  \saveexpandmode\expandarg
  \StrSubstitute{#1}{---}{\pgph@emdash}[#1]%
  \StrSubstitute{#1}{--}{\pgph@endash}[#1]%
  \restoreexpandmode}
```

```
\newif\ifpgph@ctag
\newcommand\pgph@setcitelabel[1]{%
  \edef\pgph@ckey{\csname pgph@cskey@#1\endcsname}%
  \edef\pgph@cbase{\pgph@ckey}%
  \pgph@ctagfalse
  \ifdefstring{\pgph@citelabel}{tag}{%
    \ifcsname pgph@blxtag@\pgph@ckey\endcsname
      \edef\pgph@cbase{\csname pgph@blxtag@\pgph@ckey\endcsname}%
      \pgph@ctagtrue
    \else
    \ifcsname b@\pgph@ckey\endcsname
      \global\let\pgph@cbaseg\relax
      \begingroup
        \pgph@citesanitize
        \edef\pgph@x{\csname b@\pgph@ckey\endcsname}%
        \edef\pgph@dx{\detokenize\expandafter{\pgph@x}}%
        \IfBeginWith\pgph@dx\pgph@obrace{%
          \pgph@natnumset
          \pgph@natdo
          \edef\pgph@dx{\detokenize\expandafter{\pgph@x}}%
        }{}%
        \ifx\pgph@x\@empty\else
          \IfSubStr\pgph@dx\@backslashchar{}{%
            \global\let\pgph@cbaseg\pgph@x}%
        \fi
      \endgroup
      \ifx\pgph@cbaseg\relax\else
        \let\pgph@cbase\pgph@cbaseg
        \pgph@ctagtrue
      \fi
    \fi\fi
  }{}%
  \edef\pgph@cnote{%
    \ifcsname pgph@citenote@#1\endcsname
      , \csname pgph@citenote@#1\endcsname\fi}%
  \ifx\pgph@cnote\@empty\else\pgph@dashesc\pgph@cnote\fi
  \edef\pgph@clabel{%
    \ifpgph@ctag[\fi\pgph@cbase\pgph@cnote\ifpgph@ctag]\fi}%
}
```

`\pgph@render` `\pgph@maybrender` `\pgph@lastrender`

autorun must not re-render a graph that has not changed. Rendering unconditionally is not merely wasteful: the image Graphviz produces is not guaranteed to be reproducible (its cairo-based outputs, `pdf` among them, carry a creation date), so an unconditional run makes the embedded image differ at every compilation. A build tool such as latexmk then sees an input file that never settles and reruns until it gives up, reporting that too many passes were needed.

We therefore render only when something relevant has changed. The signature of a rendering is the checksum of the `.dot` file just written, together with the engine and the format, which the `.dot` does not record; it is stored in the `.aux` by `\pgph@lastrender` and compared with the signature of the next run. Graphviz is run when the signature differs, when no signature is known yet (first run), when either output file is missing, and, as a safe fallback, when the engine provides no file checksum.

Both signatures are \detokenized before being compared. The checksum comes out of \pdf@filemdfivesum as characters of category *other*, while the same string read back from the .aux has its letters as category *letter*, so an \ifx on the raw strings would never see two runs as equal.

```
\let\pgph@prevsig\relax
\newcommand\pgph@lastrender[1]{\xdef\pgph@prevsig{\detokenize{#1}}}
\newcommand\pgph@render{%
  \immediate\write18{\pgph@engine\space -T\pgph@format\space
    \pgph@file.dot -o \pgph@file-proofgraph.\pgph@format}%
  \immediate\write18{\pgph@engine\space -Tplain\space
    \pgph@file.dot -o \pgph@file-proofgraph.plain}%
}
\newcommand\pgph@maybrender{%
  \edef\pgph@sig{\ifdefined\pdf@filemdfivesum
    \pdf@filemdfivesum{\pgph@file.dot}\fi}%
  \ifx\pgph@sig\@empty
    \pgph@render
  \else
    \edef\pgph@sig{\pgph@sig-\pgph@engine-\pgph@format}%
    \edef\pgph@sig{\expandafter\detokenize\expandafter{\pgph@sig}}%
    \IfFileExists{\pgph@file-proofgraph.\pgph@format}%
      {\IfFileExists{\pgph@file-proofgraph.plain}%
        {\ifx\pgph@sig\pgph@prevsig\else\pgph@render\fi}%
        {\pgph@render}}%
      {\pgph@render}%
```

The signature is recorded whether or not we have just rendered: either way the output files on disk are the ones this .dot produces. The write must be \immediate, as \pgph@emit runs at end of document, where a deferred write would have no shipout left to carry it.

```
    \if@filesw
      \immediate\write\@auxout{\string\pgph@lastrender{\pgph@sig}}%
    \fi
  \fi
}
```

\pgph@emit Open the file, resolve editing rules, write nodes then edges, close, and optionally run Graphviz.

```
\newwrite\pgph@out
\newwrite\pgph@mapout
\newcommand\pgph@emit{%
  \immediate\openout\pgph@out=\pgph@file.dot\relax
  \immediate\openout\pgph@mapout=\pgph@file-proofgraph.map\relax
  \immediate\write\pgph@out{digraph proofgraph \pgph@ob}%
  \immediate\write\pgph@out{\space\space rankdir="\pgph@rankdir";}%
  \begingroup
    \pgph@ignores
    \pgph@excludes
    \pgph@nodes
    \pgph@edges
    \pgph@checkrules
  \endgroup
  \immediate\write\pgph@out{\pgph@cb}%
  \immediate\closeout\pgph@out
```

```
  \immediate\closeout\pgph@mapout
  \ifpgph@autorun\pgph@maybrender\fi
}
```

The emission is registered at `\begin{document}` rather than at load time so that it is added to the end-document hook *after* packages loaded later (notably apxproof, whose appendix, with the deferred proofs and their references, is built at end of document). This way the graph is written only once that material has been typeset, and the emission does not disturb the verbatim replay of deferred proofs.

```
\AtBeginDocument{\AtEndDocument{\pgph@emit}}
```

If TikZ is available, load its `calc` library now (at `\begin{document}`, where category codes are normal) so that the hyperlink overlay in `\proofgraph` can use it without reading a file mid-document.

```
\AtBeginDocument{\@ifpackageloaded{tikz}{\usetikzlibrary{calc}}{}}
```

`\pgphnodemap` `\pgphnodemapcite` One entry of the node-to-target map written by `\pgph@emit`: it records, for a Graphviz node identifier, what the node should link to. `\pgphnodemap` records the user *label* of the result a node represents; `\pgphnodemapcite` records the *anchor* of an external citation node, so that (when the cited work is in the same document) the node can link to its bibliography entry, hyperref's `cite.`⟨*anchor*⟩ destination, the anchor being the BibTeX key except under biblatex (see `\pgph@citeanchor`). The map is read back when the graph is embedded, to turn each node into a hyperlink.

```
\newcommand\pgphnodemap[2]{\global\@namedef{pgph@lbl@#1}{#2}}
\newcommand\pgphnodemapcite[2]{\global\@namedef{pgph@clbl@#1}{#2}}
```

`\proofgraph` Include the rendered graph (from a previous run) in `autorun` mode. When hyperref and TikZ are both available (and `hyperlinks` is not switched off), each node is made into an internal hyperlink to what it represents: a result node links to the result (its `\label`), and an external citation node (from the `cite` option) links to the cited work's bibliography entry, the `cite.`⟨*anchor*⟩ destination hyperref gave it, when that work is cited in the same document. Graphviz produces a `.pdf` whose link annotations would be lost by `\includegraphics` (they sit inside a form XOBJECT), so instead we overlay transparent link areas at the node positions, which we read from the `-Tplain` output. Without those packages we fall back to a plain inclusion.

```
\newif\ifpgph@canlink
\newif\ifpgph@link
\newsavebox\pgph@imgbox
\newsavebox\pgph@natbox
\newread\pgph@plainin
\def\pgph@stop{}
\newcommand\proofgraph[1][]{%
  \@ifundefined{includegraphics}%
    {\PackageError{proofgraph}{\string\proofgraph\space requires the
       graphicx package}{Add \string\usepackage{graphicx} to your preamble
       to embed the rendered graph.}}%
    {\IfFileExists{\pgph@file-proofgraph.\pgph@format}%
      {\pgph@includegraph{#1}}%
      {\PackageWarning{proofgraph}{Rendered graph
         \pgph@file-proofgraph.\pgph@format\space not found; compile with
         shell-escape and the autorun option, then re-run}}%
```

```
}
\newcommand\pgph@includegraph[1]{%
  \pgph@canlinkfalse
  \ifpgph@hyperlinks
    \@ifpackageloaded{hyperref}%
      {\@ifpackageloaded{tikz}%
         {\IfFileExists{\pgph@file-proofgraph.plain}%
            {\pgph@canlinktrue}{}}{}}{}%
  \fi
  \ifpgph@canlink
    \pgph@graphlinked{#1}%
  \else
    \includegraphics[#1]{\pgph@file-proofgraph.\pgph@format}%
  \fi
}
```

The linked version. We read the node-to-target map under safe category codes (a label may contain _ or, under babel, an active :), measure the rendered image, place it in a TikZ node, then read the `-Tplain` file and drop one transparent link box on each mapped node. Graphviz draws the graph inset by a margin, so the image is larger than the layout bounding box the `-Tplain` coordinates live in; we recover that margin by also measuring the image at its *natural* size and comparing it with the `-Tplain` graph size (the `graph` line gives it in inches, hence the factor 72). Node positions are then expressed as fractions of the natural image, so they line up with the drawn nodes whatever size the image is finally scaled to.

```
\newcommand\pgph@graphlinked[1]{%
  \begingroup
    \catcode`\:=12 \catcode`\_=12 \catcode`\;=12 \catcode`\!=12
    \catcode`\?=12 \catcode`\&=12 \catcode`\^=12 \catcode`\~=12
    \InputIfFileExists{\pgph@file-proofgraph.map}{}{}%
  \endgroup
  \sbox\pgph@imgbox{%
    \includegraphics[#1]{\pgph@file-proofgraph.\pgph@format}}%
  \sbox\pgph@natbox{%
    \includegraphics{\pgph@file-proofgraph.\pgph@format}}%
  \edef\pgph@imgw{\the\wd\pgph@imgbox}%
  \edef\pgph@imgh{\the\dimexpr\ht\pgph@imgbox+\dp\pgph@imgbox\relax}%
  \edef\pgph@natw{\the\wd\pgph@natbox}%
  \edef\pgph@nath{\the\dimexpr\ht\pgph@natbox+\dp\pgph@natbox\relax}%
  \begin{tikzpicture}
    \node[inner sep=\z@,outer sep=\z@](pgph@P){\usebox\pgph@imgbox};
    \openin\pgph@plainin=\pgph@file-proofgraph.plain\relax
    \pgph@loopplain
    \closein\pgph@plainin
  \end{tikzpicture}%
}
\newcommand\pgph@loopplain{%
  \unless\ifeof\pgph@plainin
    \read\pgph@plainin to \pgph@line
    \pgph@procline
    \expandafter\pgph@loopplain
  \fi
}
\newcommand\pgph@procline{%
```

```
  \IfBeginWith{\pgph@line}{graph }%
    {\expandafter\pgph@dograph\pgph@line\pgph@stop}%
    {\IfBeginWith{\pgph@line}{node }%
       {\expandafter\pgph@donode\pgph@line\pgph@stop}{}}%
}
\def\pgph@dograph graph #1 #2 #3 #4\pgph@stop{%
  \def\pgph@gw{#2}\def\pgph@gh{#3}}
\def\pgph@donode node #1 #2 #3 #4 #5 #6\pgph@stop{%
  \StrDel{#1}{"}[\pgph@nid]%
  \pgph@linkfalse
  \ifcsname pgph@lbl@\pgph@nid\endcsname
    \edef\pgph@tgt{\csname pgph@lbl@\pgph@nid\endcsname}%
    \edef\pgph@thislink{\noexpand\hyperref[\pgph@tgt]}%
    \pgph@linktrue
  \else
    \ifcsname pgph@clbl@\pgph@nid\endcsname
      \edef\pgph@tgt{\csname pgph@clbl@\pgph@nid\endcsname}%
      \edef\pgph@thislink{\noexpand\hyperlink{cite.\pgph@tgt}}%
      \pgph@linktrue
    \fi
  \fi
  \ifpgph@link
    % Per-side Graphviz margin: half the natural-minus-layout size.
    \pgfmathsetmacro\pgph@mx{(\pgph@natw-\pgph@gw*72)/2}%
    \pgfmathsetmacro\pgph@my{(\pgph@nath-\pgph@gh*72)/2}%
    % Node centre as a fraction of the natural image.
    \pgfmathsetmacro\pgph@fx{(\pgph@mx+#2*72)/\pgph@natw}%
    \pgfmathsetmacro\pgph@fy{(\pgph@my+#3*72)/\pgph@nath}%
    \pgfmathsetlengthmacro\pgph@bw{#4*72/\pgph@natw*\pgph@imgw*0.92}%
    \pgfmathsetlengthmacro\pgph@bh{#5*72/\pgph@nath*\pgph@imgh*0.9}%
    \node[anchor=center]
      at ($($(pgph@P.south west)!\pgph@fx!(pgph@P.south east)$)%
      !\pgph@fy!($(pgph@P.north west)!\pgph@fx!(pgph@P.north east)$)$)
      {\pgph@thislink{\phantom{\rule{\pgph@bw}{\pgph@bh}}}};%
  \fi
}
```

⟨/package⟩

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

Change History