Tinymist Documentation (v0.14.10)

Tinymist Documentation (v0.14.10)

An integrated language service for Typst.

Visit tinymist repository: main branch, or v0.14.10.

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   “License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   “Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   “Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   “You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

   “Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   “Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   “Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   “Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   “Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

   “Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY,

or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Contents

Introduction

Tinymist ['taɪni mɪst] is an integrated language service for Typst [taɪpst]. You can also call it "微霭" [wēi ǎi] in Chinese.

It contains:

- an analyzing library for Typst, see tinymist-query.
- a CLI for Typst, see tinymist.
  - which provides a language server for Typst, see Language Features.
  - which provides a preview server for Typst, see Preview Feature.
- a VSCode extension for Typst, see Tinymist VSCode Extension.

1. Features

Language service (LSP) features:

- Semantic highlighting
  - The "semantic highlighting" is supplementary to "syntax highlighting".
- Code actions
  - Also known as "quick fixes" or "refactorings".
- Formatting (Reformatting)
  - Provide the user with support for formatting whole documents, using typstfmt or typstyle.
- Document highlight
  - Highlight all break points in a loop context.
  - (Todo) Highlight all exit points in a function context.
  - (Todo) Highlight all captures in a closure context.
  - (Todo) Highlight all occurrences of a symbol in a document.
- Document links
  - Renders path or link references in the document, such as `image("path.png")` or `bibliography(style: "path.csl")`.
- Document symbols
  - Also known as "document outline" or "table of contents" *in Typst*.
- Folding ranges
  - You can collapse code/content blocks and headings.
- Goto definitions
  - Right-click on a symbol and select "Go to Definition".
  - Or ctrl+click on a symbol.
- References
  - Right-click on a symbol and select "Go to References" or "Find References".
  - Or ctrl+click on a symbol.
- Hover tips
  - Also known as "hovering tooltip".
  - Render docs according to tidy style.
- Inlay hints
  - Inlay hints are special markers that appear in the editor and provide you with additional information about your code, like the names of the parameters that a called method expects.
- Color Provider
  - View all inlay colorful label for color literals in your document.
  - Change the color literal's value by a color picker or its code presentation.

- Code Lens
  - Should give contextual buttons along with code. For example, a button for exporting your document to various formats at the start of the document.
- Rename symbols and embedded paths
- Help with function and method signatures
- Workspace Symbols
- Code Action
  - Increasing/Decreasing heading levels.
  - Turn equation into "inline", "block" or "multiple-line block" styles.
- experimental/onEnter
  - `Enter` inside triple-slash comments automatically inserts `///`
  - `Enter` in the middle or after a trailing space in `//` inserts `//`
  - `Enter` inside `//!` doc comments automatically inserts `//!`
  - `Enter` inside equation markups automatically inserts indents.

Extra features:

- Compiles to PDF on save (configurable to as-you-type, or other options). Check Docs: Exporting Documents.
- Also compiles to SVG, PNG, HTML, Markdown, Text, and other formats by commands, vscode tasks, or code lenses.
- Provides test, benchmark, coverage collecting on documents and modules. Check Docs: Testing Features.
- Provides builtin linting. Check Docs: Linting Features.
- Provides a status bar item to show the current document's compilation status and words count.
- Editor tools:
  - View a list of templates in template gallery. (`tinymist.showTemplateGallery`)
  - Click a button in template gallery to initialize a new project with a template. (`tinymist.initTemplate` and `tinymist.initTemplateInPlace`)
  - Trace execution in current document (`tinymist.profileCurrentFile`).

2. Versioning and Release Cycle

Tinymist's versions follow the Semantic Versioning scheme, in format of `MAJOR.MINOR.PATCH`. Besides, tinymist follows special rules for the version number:

- If a version is suffixed with `-rcN` ($N > 0$), e.g. `0.11.0-rc1` and `0.12.1-rc1`, it means this version is a release candidate. It is used to test publish script and E2E functionalities. These versions will not be published to the marketplace.
- If the `PATCH` number is odd, e.g. `0.11.1` and `0.12.3`, it means this version is a nightly release. The nightly release will use both tinymist and typst at **main branch**. They will be published as prerelease version to the marketplace. Note that in nightly releases, we change `#sys.version` to the next minor release to help develop documents with nightly features. For example, in tinymist nightly v0.12.1 or v0.12.3, the `#sys.version` is changed to `version(0, 13, 0)`.
- Otherwise, if the `PATCH` number is even, e.g. `0.11.0` and `0.12.2`, it means this version is a regular release. The regular release will always use the recent stable version of tinymist and typst.

The release cycle is as follows:

- If there is a typst version update, a new major or minor version will be released intermediately. This means tinymist will always align the minor version with typst.
- If there is at least a bug or feature added this week, a new patch version will be released.

3. Installation

Follow the instructions to enable tinymist in your favorite editor.

- VS Cod(e,ium)
- Neovim
- Emacs
- Sublime Text
- Helix
- Zed

4. Installing Regular/Nightly Prebuilds from GitHub

Note: if you are not knowing what is a regular/nightly release, please don't follow this section.

Besides published releases specific for each editors, you can also download the latest regular/nightly prebuilts from GitHub and install them manually.

- Regular prebuilts can be found in GitHub Releases.
- Nightly prebuilts can be found in GitHub Actions.
  - (Suggested) Use the tinymist-nightly-installer to install the nightly prebuilts automatically.
    - Unix (Bash):

      ```
      curl -sSL https://github.com/hongjr03/tinymist-nightly-installer/releases/latest/download/run.sh | bash
      ```

    - Windows (PowerShell):

      ```
      iwr https://github.com/hongjr03/tinymist-nightly-installer/releases/latest/download/run.ps1 -UseBasicParsing | iex
      ```

  - The prebuilts for other revisions can also be found manually. For example, if you are seeking a nightly release for the featured PR: build: bump version to 0.11.17-rc1, you could click and go to the action page run for the related commits and download the artifacts.

To install extension file (the file with `.vsix` extension) manually, please `Ctrl+Shift+X` in the editor window and drop the downloaded vsix file into the opened extensions view.

5. Documentation

See Online Documentation.

6. Packaging

Stable Channel:

https://repology.org/project/tinymist/versions

Nightly Channel:

https://repology.org/project/tinymist-nightly/versions

7. Roadmap

7.1. Short Terms

To encourage contributions, we create many Pull Requests in draft to navigate short-term plans. They give you a hint of what or where to start in this large repository.

7.2. Long Terms

We are planning to implement the following features in typst v0.14.0 or spare time in weekend:

- Type checking: complete the type checker.
- Periscope renderer: It is disabled since vscode reject to render SVGs containing foreignObjects.
- Inlay hint: It is disabled *by default* because of performance issues.
- Find references of dictionary fields and named function arguments.
- Improve symbol view's appearance.
- Improve package view.
  - Navigate to symbols by clicking on the symbol name in the view.
  - Automatically locate the symbol item in the view when viewing local documentation.
  - Remember the recently invoked package commands, e.g. "Open Docs of @preview/cetz:0.3.1", "Open directory of @preview/touying:0.5.3".
- Improve label view.
  - Group labels.
  - Search labels.
  - Keep (persist) group preferences.
- Improve Typst Preview.
  - Pin drop-down: Set the file to preview in the drop-down for clients that doesn't support passing arguments to the preview command.
  - Render in web worker (another thread) to reduce overhead on the electron's main thread.
- ~~Spell checking: There is already a branch but no suitable (default) spell checking library is found.~~
  - typos is great for typst. harper looks promise.

If you are interested by any above features, please feel free to send Issues to discuss or PRs to implement to GitHub.

8. Contributing

Please read the CONTRIBUTING.md file for contribution guidelines.

9. Sponsoring

Tinymist thrives on community love and remains proudly independent. While we don't accept direct project funding, we warmly welcome support for our maintainers' personal efforts. Please go to Maintainers Page and Contributors Page and find their personal pages for more information. It is also welcomed to directly ask questions about sponsoring on the GitHub Issues.

10. Acknowledgements

- Partially code is inherited from typst-lsp
- The integrating **offline** handwritten-stroke recognizer is powered by Detypify.
- The integrating preview service is powered by typst-preview.
- The integrating local package management functions are adopted from vscode-typst-sync.

Part 1. Editor Integration

Configurations

All editors share the same server-side configurations. Besides, the VS Code extension has some additional configurations.

Please check the following chapters:

- VS Cod(e,ium) Specific Configurations.
- Server-Side Configurations.

VS Code Extension Configurations

1. `tinymist.compileStatus`

In VSCode, enable compile status meaning that the extension will show the compilation status in the status bar. Since Neovim and Helix don't have a such feature, it is disabled by default at the language server label.

- **Type**: `string`
- **Valid Values**:
  - `"enable"`
  - `"disable"`
- **Default**: `"enable"`

2. `tinymist.completion.postfix`

Whether to enable postfix code completion. For example, `[A].box|` will be completed to `box[A]|`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

3. `tinymist.completion.postfixUfcs`

Whether to enable UFCS-style completion. For example, `[A].box|` will be completed to `box[A]|`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

4. `tinymist.completion.postfixUfcsLeft`

Whether to enable left-variant UFCS-style completion. For example, `[A].table|` will be completed to `table(|)[A]`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

5. `tinymist.completion.postfixUfcsRight`

Whether to enable right-variant UFCS-style completion. For example, `[A].table|` will be completed to `table([A], |)`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

6. `tinymist.completion.symbol`

Whether to make symbol completion stepless. For example, `$ar|$` will be completed to `$arrow.r$`. Hint: Restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:

- `"step"` : Complete symbols step by step
- `"stepless"` : Complete symbols steplessly

- **Default**: `"step"`

7. `tinymist.completion.triggerOnSnippetPlaceholders`

Whether to trigger completions on arguments (placeholders) of snippets. For example, `box` will be completed to `box(|)`, and server will request the editor (lsp client) to request completion after moving cursor to the placeholder in the snippet. Note: this has no effect if the editor doesn't support `editor.action.triggerSuggest` or `tinymist.triggerSuggestAndParameterHints` command. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `false`

8. `tinymist.configureDefaultWordSeparator`

Whether to configure default word separators on startup

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Override the default word separators on startup
  - `"disable"` : Do not override the default word separators on startup
- **Default**: `"disable"`

9. `tinymist.convertExtension`

This configuration specifies how to let the editor use Typst templates to convert other format files into PDF documents.

This configuration item can be one of following types:

-
  - **Type**: `array`
- **Default**: `[]`

10. `tinymist.copyAndPaste`

Whether to handle paste of resources into the editing typst document. Note: restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Enable copy-and-paste.
  - `"disable"` : Disable copy-and-paste.
- **Default**: `"enable"`

11. `tinymist.dragAndDrop`

Whether to handle drag-and-drop of resources into the editing typst document. Note: restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Enable drag-and-drop.

  - `"disable"` : Disable drag-and-drop.
- **Default**: `"enable"`

12. `tinymist.exportPdf`

The extension can export PDFs of your Typst files. This setting controls whether this feature is enabled and how often it runs.

- **Type**: `string`
- **Valid Values**:
  - `"never"` : Never export PDFs, you will manually run typst.
  - `"onSave"` : Export PDFs when you save a file.
  - `"onType"` : Export PDFs as you type in a file.
  - `"onDocumentHasTitle"` : (Deprecated) Export PDFs when a document has a title (and save a file), which is useful to filter out template files.
- **Default**: `"never"`

13. `tinymist.exportTarget`

The target to export the document to. Defaults to `paged`. Note: you can still export PDF when it is set to `html`. This configuration only affects how the language server completes your code.

- **Type**: `string`
- **Valid Values**:
  - `"paged"` : The current export target is for PDF, PNG, and SVG export.
  - `"html"` : The current export target is for HTML export.
- **Default**: `"paged"`

14. `tinymist.fontPaths`

A list of file or directory path to fonts. Note: The configuration source in higher priority will **override** the configuration source in lower priority. The order of precedence is: Configuration `tinymist.fontPaths` > Configuration `tinymist.typstExtraArgs.fontPaths` > LSP's CLI Argument `--font-path` > The environment variable `TYPST_FONT_PATHS` (a path list separated by `;` (on Windows) or `:` (Otherwise)). Note: If the path to fonts is a relative path, it will be resolved based on the root directory. Note: In VSCode, you can use VSCode variables in the path, e.g. `${workspaceFolder}/fonts`.

- **Type**: array | null

15. `tinymist.formatterIndentSize`

Sets the indent size (using space) for the formatter.

- **Type**: `number`
- **Default**: `2`

16. `tinymist.formatterMode`

The extension can format Typst files using typstfmt or typstyle.

- **Type**: `string`
- **Valid Values**:
  - `"disable"` : Formatter is not activated.
  - `"typstyle"` : Use typstyle formatter.

  - `"typstfmt"` : Use typstfmt formatter.
- **Default**: `"typstyle"`

17. `tinymist.formatterPrintWidth`

Sets the print width for the formatter, which is a **soft limit** of characters per line. See the definition of *Print Width*. Note: this has lower priority than the formatter's specific configurations.

- **Type**: `number`
- **Default**: `120`

18. `tinymist.formatterProseWrap`

Controls how the formatter handles prose line wrapping. If enabled, the formatter will insert hard line breaks at the specified print width. If disabled, the formatter keeps the original line breaks and spaces.

- **Type**: `boolean`
- **Default**: `false`

19. `tinymist.lint.enabled`

Enable or disable lint checks. Note: restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `false`

20. `tinymist.lint.when`

Configure when to perform lint checks. Note: restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:
  - `"onSave"` : Perform lint checks on save
  - `"onType"` : Perform lint checks on type
- **Default**: `"onSave"`

21. `tinymist.onEnterEvent`

Enable or disable experimental/onEnter (LSP onEnter feature) to allow automatic insertion of characters on enter, such as `///` for comments. Note: restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

22. `tinymist.onPaste`

The script to be executed when pasting resources into the editing typst document. If the script code starts with `{` and ends with `}`, it will be evaluated as a typst code expression, e.g. `$root/x/$dir/$name` evaluated as `/path/to/root/x/dir/main`, otherwise it will be evaluated as a path pattern, e.g. `{ join(root, "x", dir, if name.ends-with(".png") ("imgs"), name) }` evaluated as `/path/to/root/x/dir/imgs/main`. The extra valid definitions are `root`, `dir`, `name`, and `join`. To learn more about

the paste script, please visit Script Hooks. Hint: you could import `@local` packages in the paste script. Note: restarting the editor is required to change this setting.

- **Type**: `string`
- **Default**: `"$root/assets"`

23. `tinymist.outputPath`

The path pattern to store Typst artifacts, you can use `$root` or `$dir` or `$name` to do magic configuration, e.g. `$dir/$name` (default) and `$root/target/$dir/$name`.

- **Type**: `string`
- **Default**: `""`

24. `tinymist.preview.background.args`

The arguments that the background preview server used for. It is only used when `tinymist.preview.background` is enabled. Check `tinymist preview` to see the allowed arguments.

- **Type**: `array`
- **Properties**:
  - `"type"`:
    - **Type**: `string`
- **Default**: `[`

```
"--data-plane-host=127.0.0.1:23635",
"--invert-colors=auto"
]
```

25. `tinymist.preview.background.enabled`

This configuration is only used for the editors that doesn't support lsp well, e.g. helix and zed. When it is enabled, the preview server listens a specific tcp port in the background. You can discover the background previewers in the preview panel.

- **Type**: `boolean`
- **Default**: `false`

26. `tinymist.preview.browsing.args`

The arguments used by `tinymist.startDefaultPreview` command. Check `tinymist preview` to see the allowed arguments.

- **Type**: `array`
- **Properties**:
  - `"type"`:
    - **Type**: `string`
- **Default**: `[`

```
"--data-plane-host=127.0.0.1:0",
"--invert-colors=auto",
"--open"
]
```

27. `tinymist.preview.cursorIndicator`

(Experimental) Show typst cursor indicator in preview.

- **Type**: `boolean`
- **Default**: `false`

28. `tinymist.preview.invertColors`

Invert colors of the preview (useful for dark themes without cost). Please note you could see the origin colors when you hover elements in the preview. It is also possible to specify strategy to each element kind by an object map in JSON format.

This configuration item can be one of following types:

- Invert colors of the preview.
  - **Type**: `string`
  - **Valid Values**:
    - `"never"` : Never
    - `"auto"` : Inferring from the browser settings
    - `"always"` : Always
- Invert colors of the preview in a more fine-grained way.
  - **Type**: `object`
  - **Properties**:
    - `"rest"`: Invert colors of the rest of the elements.
      - **Type**: `string`
      - **Valid Values**:
        - `"never"` : Never
        - `"auto"` : Inferring from the browser settings
        - `"always"` : Always
      - **Default**: `"never"`
    - `"image"`: Invert colors of the images.
      - **Type**: `string`
      - **Valid Values**:
        - `"never"` : Never
        - `"auto"` : Inferring from the browser settings
        - `"always"` : Always
      - **Default**: `"never"`
- **Default**: `"never"`

29. `tinymist.preview.partialRendering`

Only render visible part of the document. This can improve performance but still being experimental.

- **Type**: `boolean`
- **Default**: `true`

30. `tinymist.preview.refresh`

Refresh preview when the document is saved or when the document is changed

- **Type**: `string`
- **Valid Values**:
  - `"onSave"` : Refresh preview on save
  - `"onType"` : Refresh preview on type
- **Default**: `"onType"`

31. `tinymist.preview.scrollSync`

Configure scroll sync mode.

- **Type**: `string`
- **Valid Values**:
  - `"never"` : Disable automatic scroll sync
  - `"onSelectionChangeByMouse"` : Scroll preview to current cursor position when selection changes by mouse
  - `"onSelectionChange"` : Scroll preview to current cursor position when selection changes by mouse or keyboard (any source)
- **Default**: `"onSelectionChangeByMouse"`

32. `tinymist.previewFeature`

Enable or disable preview features of Typst. Note: restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Enable preview features.
  - `"disable"` : Disable preview features.
- **Default**: `"enable"`

33. `tinymist.projectResolution`

This configuration specifies the way to resolved projects.

- **Type**: `string`
- **Valid Values**:
  - `"singleFile"` : Manage typst documents like what we did in Markdown. Each single file is an individual document and no project resolution is needed.
  - `"lockDatabase"` : Manage typst documents like what we did in Rust. For each workspace, tinymist tracks your preview and compilation history, and stores the information in a lock file. Tinymist will automatically selects the main file to use according to the lock file. This also allows other tools push preview and export tasks to language server by updating the lock file.
- **Default**: `"singleFile"`

34. `tinymist.renderDocs`

(Experimental) Whether to render typst elements in (hover) docs. In VS Code, when this feature is enabled, tinymist will store rendered results in the filesystem's temporary storage to show them in the hover content. Note: Please disable this feature if the editor doesn't support/handle image previewing in docs.

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Enable render docs.
  - `"disable"` : Disable render docs.
- **Default**: `"enable"`

35. `tinymist.rootPath`

Configure the root for absolute paths in typst. Note: for Neovim users, if it complains root not found, you must set `require("lspconfig")["tinymist"].setup { root_dir }` as well, see tinymist#528.

- **Type**: string | null

36. `tinymist.semanticTokens`

Enable or disable semantic tokens (LSP syntax highlighting)

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Use semantic tokens for syntax highlighting
  - `"disable"` : Do not use semantic tokens for syntax highlighting
- **Default**: `"enable"`

37. `tinymist.serverPath`

The extension can use a local tinymist executable instead of the one bundled with the extension. This setting controls the path to the executable. The string “tinymist” means look up Tinymist in PATH.

- **Type**: string | null

38. `tinymist.showExportFileIn`

Configures way of opening exported files, e.g. inside of editor tabs or using system application.

This configuration item can be one of following types:

- Configures way of opening exported files, e.g. inside of editor tabs or using system application.
  - **Type**: `string`
  - **Valid Values**:
    - `"editorTab"` : Show the exported files in editor tabs.
    - `"systemDefault"` : Show the exported files by system default application.
  - **Default**: `"editorTab"`

39. `tinymist.statusBarFormat`

Set format string of the server status. For example, `{compileStatusIcon}{wordCount} [{fileName}]` will format the status as `$(check) 123 words [main]`. Valid placeholders are:

- `{compileStatusIcon}`: Icon indicating the compile status
- `{wordCount}`: Number of words in the document
- `{charCount}`: Number of characters in the document
- `{fileName}`: Name of the file being compiled

Note: The status bar will be hidden if the format string is empty.

- **Type**: `string`
- **Default**: `"{compileStatusIcon} {wordCount} [{fileName}]"`

40. `tinymist.syntaxOnly`

Configure whether to enable syntax-only mode for the language server. In syntax-only mode, the language server will only provide syntax checking and basic code completion, but will not perform

full document compilation or code analysis. This can be useful for improving performance on low-end devices or when working with large documents.

- **Type**: `string`
- **Valid Values**:
  - `"auto"` : Always disable syntax-only mode. The strategy may be changed in the future, for example, automatically enable syntax-only mode when the system is in power-saving mode.
  - `"onPowerSaving"` : Enable syntax-only mode when the system is in power-saving mode.
  - `"enable"` : Always enable syntax-only mode.
  - `"disable"` : Always disable syntax-only mode.
- **Default**: `"auto"`

41. `tinymist.systemFonts`

A flag that determines whether to load system fonts for Typst compiler, which is useful for ensuring reproducible compilation. If set to null or not set, the extension will use the default behavior of the Typst compiler. Note: You need to restart LSP to change this options.

- **Type**: `boolean`
- **Default**: `true`

42. `tinymist.trace.server`

Traces the communication between VS Code and the language server.

- **Type**: `string`
- **Valid Values**:
  - `"off"`
  - `"messages"`
  - `"verbose"`
- **Default**: `"off"`

43. `tinymist.typingContinueCommentsOnNewline`

Whether to prefix newlines after comments with the corresponding comment prefix.

- **Type**: `boolean`
- **Default**: `true`

44. `tinymist.typstExtraArgs`

You can pass any arguments as you like, and we will try to follow behaviors of the **same version** of typst-cli. Note: the arguments may be overridden by other settings. For example, `--font-path` will be overridden by `tinymist.fontPaths`.

- **Type**: `array`
- **Default**: `[]`

Server-Side Configurations

45. `compileStatus`

In VSCode, enable compile status meaning that the extension will show the compilation status in the status bar. Since Neovim and Helix don't have a such feature, it is disabled by default at the language server label.

- **Type**: `string`
- **Valid Values**:
  - `"enable"`
  - `"disable"`
- **Default**: `"enable"`

46. `completion.postfix`

Whether to enable postfix code completion. For example, `[A].box|` will be completed to `box[A]|`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

47. `completion.postfixUfcs`

Whether to enable UFCS-style completion. For example, `[A].box|` will be completed to `box[A]|`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

48. `completion.postfixUfcsLeft`

Whether to enable left-variant UFCS-style completion. For example, `[A].table|` will be completed to `table(|)[A]`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

49. `completion.postfixUfcsRight`

Whether to enable right-variant UFCS-style completion. For example, `[A].table|` will be completed to `table([A], |)`. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `true`

50. `completion.symbol`

Whether to make symbol completion stepless. For example, `$ar|$` will be completed to `$arrow.r$`. Hint: Restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:

  - `"step"` : Complete symbols step by step
  - `"stepless"` : Complete symbols steplessly
- **Default**: `"step"`

51. `completion.triggerOnSnippetPlaceholders`

Whether to trigger completions on arguments (placeholders) of snippets. For example, `box` will be completed to `box(|)`, and server will request the editor (lsp client) to request completion after moving cursor to the placeholder in the snippet. Note: this has no effect if the editor doesn't support `editor.action.triggerSuggest` or `tinymist.triggerSuggestAndParameterHints` command. Hint: Restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `false`

52. `exportPdf`

The extension can export PDFs of your Typst files. This setting controls whether this feature is enabled and how often it runs.

- **Type**: `string`
- **Valid Values**:
  - `"never"` : Never export PDFs, you will manually run typst.
  - `"onSave"` : Export PDFs when you save a file.
  - `"onType"` : Export PDFs as you type in a file.
  - `"onDocumentHasTitle"` : (Deprecated) Export PDFs when a document has a title (and save a file), which is useful to filter out template files.
- **Default**: `"never"`

53. `exportTarget`

The target to export the document to. Defaults to `paged`. Note: you can still export PDF when it is set to `html`. This configuration only affects how the language server completes your code.

- **Type**: `string`
- **Valid Values**:
  - `"paged"` : The current export target is for PDF, PNG, and SVG export.
  - `"html"` : The current export target is for HTML export.
- **Default**: `"paged"`

54. `fontPaths`

A list of file or directory path to fonts. Note: The configuration source in higher priority will **override** the configuration source in lower priority. The order of precedence is: Configuration `tinymist.fontPaths` > Configuration `tinymist.typstExtraArgs.fontPaths` > LSP's CLI Argument `--font-path` > The environment variable `TYPST_FONT_PATHS` (a path list separated by `;` (on Windows) or `:` (Otherwise)). Note: If the path to fonts is a relative path, it will be resolved based on the root directory. Note: In VSCode, you can use VSCode variables in the path, e.g. `${workspaceFolder}/fonts`.

- **Type**: array | null

55. `formatterIndentSize`

Sets the indent size (using space) for the formatter.

- **Type**: `number`
- **Default**: `2`

56. `formatterMode`

The extension can format Typst files using typstfmt or typstyle.

- **Type**: `string`
- **Valid Values**:
  - `"disable"` : Formatter is not activated.
  - `"typstyle"` : Use typstyle formatter.
  - `"typstfmt"` : Use typstfmt formatter.
- **Default**: `"typstyle"`

57. `formatterPrintWidth`

Sets the print width for the formatter, which is a **soft limit** of characters per line. See the definition of *Print Width*. Note: this has lower priority than the formatter's specific configurations.

- **Type**: `number`
- **Default**: `120`

58. `formatterProseWrap`

Controls how the formatter handles prose line wrapping. If enabled, the formatter will insert hard line breaks at the specified print width. If disabled, the formatter keeps the original line breaks and spaces.

- **Type**: `boolean`
- **Default**: `false`

59. `lint.enabled`

Enable or disable lint checks. Note: restarting the editor is required to change this setting.

- **Type**: `boolean`
- **Default**: `false`

60. `lint.when`

Configure when to perform lint checks. Note: restarting the editor is required to change this setting.

- **Type**: `string`
- **Valid Values**:
  - `"onSave"` : Perform lint checks on save
  - `"onType"` : Perform lint checks on type
- **Default**: `"onSave"`

61. `outputPath`

The path pattern to store Typst artifacts, you can use `$root` or `$dir` or `$name` to do magic configuration, e.g. `$dir/$name` (default) and `$root/target/$dir/$name`.

- **Type**: `string`
- **Default**: `""`

62. `preview.background.args`

The arguments that the background preview server used for. It is only used when `tinymist.preview.background` is enabled. Check `tinymist preview` to see the allowed arguments.

- **Type**: `array`
- **Properties**:
  - `"type"`:
    - **Type**: `string`
- **Default**: `[`

```
"--data-plane-host=127.0.0.1:23635",
"--invert-colors=auto"
]
```

63. `preview.background.enabled`

This configuration is only used for the editors that doesn't support lsp well, e.g. helix and zed. When it is enabled, the preview server listens a specific tcp port in the background. You can discover the background previewers in the preview panel.

- **Type**: `boolean`
- **Default**: `false`

64. `preview.browsing.args`

The arguments used by `tinymist.startDefaultPreview` command. Check `tinymist preview` to see the allowed arguments.

- **Type**: `array`
- **Properties**:
  - `"type"`:
    - **Type**: `string`
- **Default**: `[`

```
"--data-plane-host=127.0.0.1:0",
"--invert-colors=auto",
"--open"
]
```

65. `preview.invertColors`

Invert colors of the preview (useful for dark themes without cost). Please note you could see the origin colors when you hover elements in the preview. It is also possible to specify strategy to each element kind by an object map in JSON format.

This configuration item can be one of following types:

- Invert colors of the preview.

    - **Type**: `string`
    - **Valid Values**:
      - `"never"` : Never
      - `"auto"` : Inferring from the browser settings
      - `"always"` : Always
- Invert colors of the preview in a more fine-grained way.
    - **Type**: `object`
    - **Properties**:
      - `"rest"`: Invert colors of the rest of the elements.
        - **Type**: `string`
        - **Valid Values**:
          - `"never"` : Never
          - `"auto"` : Inferring from the browser settings
          - `"always"` : Always
        - **Default**: `"never"`
      - `"image"`: Invert colors of the images.
        - **Type**: `string`
        - **Valid Values**:
          - `"never"` : Never
          - `"auto"` : Inferring from the browser settings
          - `"always"` : Always
        - **Default**: `"never"`
- **Default**: `"never"`

66. `preview.partialRendering`

Only render visible part of the document. This can improve performance but still being experimental.

- **Type**: `boolean`
- **Default**: `true`

67. `preview.refresh`

Refresh preview when the document is saved or when the document is changed

- **Type**: `string`
- **Valid Values**:
    - `"onSave"` : Refresh preview on save
    - `"onType"` : Refresh preview on type
- **Default**: `"onType"`

68. `projectResolution`

This configuration specifies the way to resolved projects.

- **Type**: `string`
- **Valid Values**:
    - `"singleFile"` : Manage typst documents like what we did in Markdown. Each single file is an individual document and no project resolution is needed.
    - `"lockDatabase"` : Manage typst documents like what we did in Rust. For each workspace, tinymist tracks your preview and compilation history, and stores the information in a lock file.

Tinymist will automatically selects the main file to use according to the lock file. This also allows other tools push preview and export tasks to language server by updating the lock file.

- **Default**: `"singleFile"`

69. `rootPath`

Configure the root for absolute paths in typst. Note: for Neovim users, if it complains root not found, you must set `require("lspconfig")["tinymist"].setup { root_dir }` as well, see tinymist#528.

- **Type**: string | null

70. `semanticTokens`

Enable or disable semantic tokens (LSP syntax highlighting)

- **Type**: `string`
- **Valid Values**:
  - `"enable"` : Use semantic tokens for syntax highlighting
  - `"disable"` : Do not use semantic tokens for syntax highlighting
- **Default**: `"enable"`

71. `syntaxOnly`

Configure whether to enable syntax-only mode for the language server. In syntax-only mode, the language server will only provide syntax checking and basic code completion, but will not perform full document compilation or code analysis. This can be useful for improving performance on low-end devices or when working with large documents.

- **Type**: `string`
- **Valid Values**:
  - `"auto"` : Always disable syntax-only mode. The strategy may be changed in the future, for example, automatically enable syntax-only mode when the system is in power-saving mode.
  - `"onPowerSaving"` : Enable syntax-only mode when the system is in power-saving mode.
  - `"enable"` : Always enable syntax-only mode.
  - `"disable"` : Always disable syntax-only mode.
- **Default**: `"auto"`

72. `systemFonts`

A flag that determines whether to load system fonts for Typst compiler, which is useful for ensuring reproducible compilation. If set to null or not set, the extension will use the default behavior of the Typst compiler. Note: You need to restart LSP to change this options.

- **Type**: `boolean`
- **Default**: `true`

73. `typstExtraArgs`

You can pass any arguments as you like, and we will try to follow behaviors of the **same version** of typst-cli. Note: the arguments may be overridden by other settings. For example, `--font-path` will be overridden by `tinymist.fontPaths`.

- **Type**: `array`
- **Default**: `[]`

Editor Frontends

Leveraging the interface of LSP, tinymist provides frontends to each editor, located in the editor folders. They are minimal, meaning that LSP should finish its main LSP features as many as possible without help of editor frontends. The editor frontends just enhances your code experience. For example, the vscode frontend takes responsibility on providing some nice editor tools. It is recommended to install these editors frontend for your editors.

Check the following chapters for uses:

- VS Cod(e,ium)
- Neovim
- Emacs
- Sublime Text
- Helix
- Zed

Tinymist Typst VS Code Extension

A VS Code or VS Codium extension for Typst. You can find the extension on:

- Stable and pre-release versions available at Visual Studio Marketplace and Open VSX.
- The artifacts built per commits at GitHub Actions.
- The artifacts built per release at GitHub Releases.

74. Features

See Tinymist Features for a list of features.

75. LLM-Assisted Documentation

The Software Specification contains user interface provided by this extension, You could find solutions by text search in the docs or ask LLMs with plugging the docs.

76. Configuration Reference

Configuration Reference.

77. Usage Tips

77.1. Initializing with a Template

To initialize a Typst project:

- Use command `Typst init template` (tinymist.initTemplate) to initialize a new Typst project based on a template.
- Use command `Typst show template` (tinymist.showTemplateGallery) to show available Typst templates for picking up a template to initialize.

🎉 If your template contains only a single file, you can also insert the template content in place with command:

- Use command `Typst template in place` (tinymist.initTemplateInPlace) and input a template specifier for initialization.

77.2. Configuring LSP-Enhanced Formatters

1. Open settings.
2. Search for “Tinymist Formatter” and modify the value.
   - Use `"formatterMode": "typstyle"` for typstyle.
   - Use `"formatterMode": "typstfmt"` for typstfmt.

Tips: to enable formatting on save, you should add extra settings for typst language:

```
{
  "[typst]": {
    "editor.formatOnSave": true
  }
}
```

77.3. Configuring Linter

1. Open settings.
2. Search for “Tinymist Lint” and modify the value.

1. Toggle “Enabled” to enable or disable the linter.
2. Change “When” to configure when the linter runs.
   - (Default) `onSave` run linting when you save the file.
   - `onType` run linting as you type.

77.4. Configuring/Using Tinymist’s Activity Bar (Sidebar)

If you don’t like the activity bar, you can right-click on the activity bar and uncheck “Tinymist” to hide it.

77.4.1. Symbol View

- Search symbols by keywords, descriptions, or handwriting.
- See symbols grouped by categories.
- Click on a symbol, then it will be inserted into the editor.

77.4.2. Tool View

- Template Gallery: Show available Typst templates for picking up a template to initialize.
- Document Summary: Show a summary of the current document.
- Symbols: Show symbols in the current document.
- Fonts: Show fonts in the current document.
- Profiling: Profile the current document.

77.4.3. Package View

- Create or open some local typst packages.
- Show a list of available typst packages and invoke associated commands.

77.4.4. Content View

- Show thumbnail content of the current document, which is useful for creating slides.

77.4.5. Label View

- Show labels in the current workspace.

77.4.6. Outline View

- Show outline of exported document, viewing typst as a markup language.
  - This is slightly different from the LSP-provided document outline, which shows the syntax structure of the document, viewing typst as a programming language.

77.5. Preview Command

Open command palette (Ctrl+Shift+P), and type `>Typst Preview:`.

You can also use the shortcut (Ctrl+K V).

77.6. Theme-aware template (previewing)

In short, there is a `sys.inputs` item added to the compiler when your document is under the context of *user editing or previewing task*. You can use it to configure your template:

```
#let preview-args = json.decode(sys.inputs.at("x-preview", default: "{}"))
// One is previewing the document.
#let is-preview = sys.inputs.has("x-preview")
// `dark` or `light`
#let preview-theme = preview-args.at("theme", default: "light")
```

For details, please check Preview’s sys.inputs.

77.7. Configuring path to search fonts

To configure path to search fonts:

1. Open settings.
   - File -> Preferences -> Settings (Linux, Windows).
   - Code -> Preferences -> Settings (Mac).
2. Search for "Tinymist Font Paths" for providing paths to search fonts order-by-order.
3. Search for "Tinymist System Fonts" for disabling system fonts to be searched, which is useful for reproducible rendering your PDF documents.
4. Reload the window or restart the vscode editor to make the settings take effect.

Note: you must provide absolute paths.

Note: you can use vscode variables in the settings, see vscode-variables for more information.

77.8. Configuring path to root directory

To configure the root path resolved for Typst compiler:

1. Open settings.
2. Search for "Tinymist Root Path" and modify the value.
3. Reload the window or restart the vscode editor to make the settings take effect. **Note:** you must provide absolute paths.

77.9. Managing Local Packages

1. Use `Typst: Create Typst Local Package` command to create a local package.
2. Use `Typst: Open Typst Local Package` command to open a local package.
3. View and manage a list of available local packages in the "PACKAGE" view in the activity bar.

77.10. Compiling PDF

This extension compiles to PDF, but it doesn't have a PDF viewer yet. To view the output as you work, install a PDF viewer extension, such as `vscode-pdf`.

To find a way to compile PDF:

- Click the code len `Export PDF` at the top of document, or use command `Typst Show PDF ...`, to show the current document to PDF.
- Use command `Typst Export PDF` to export the current document to PDF.
- There are code lens buttons at the start of the document to export your document to PDF or other formats.

To configure when PDFs are compiled:

1. Open settings.
2. Search for "Tinymist Export PDF".
3. Change the "Export PDF" setting.
   - `onSave` makes a PDF after saving the Typst file.
   - `onType` makes PDF files live, as you type.
   - `never` disables PDF compilation.
   - `onDocumentHasTitle` makes a PDF when the document has a title and, as you save.

To configure where PDFs are saved:

1. Open settings.
2. Search for "Tinymist Output Path".
3. Change the "Output Path" setting. This is the path pattern to store artifacts, you can use `$root` or `$dir` or `$name` to do magic configuration
   - e.g. `$root/$dir/$name` (default) for `$root/path/to/main.pdf`.

- e.g. `$root/target/$dir/$name` for `$root/target/path/to/main.pdf`.
- e.g. `$root/target/foo` for `$root/target/foo.pdf`. This will ensure that the output is always output to `target/foo.pdf`.

Note: the output path should be substituted as an absolute path.

77.11. Exporting to Other Formats

You can export your documents to various other formats by lsp as well. Currently, the following formats are supported:

- Official svg, png, and pdf.
- Unofficial html, md (typlite), and txt
- Query Results (into json, yaml, or txt), and pdfpc (by `typst query --selector <pdfpc-file>`, for Touying)

See Docs: Exporting Documents for more information.

77.12. Working under Power-Saving Mode or with Resource-consumed Projects

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting, by:

1. Open settings.
2. Search for "Tinymist Syntax Only Mode".
3. Toggle the "Syntax Only Mode" setting to always enable, disable syntax only mode, or enable on power-saving mode.

For more information about power-saving mode, see Syntax-Only Mode.

77.13. Working with Multiple-File Projects

You can pin a main file by command.

- Use command `Typst Pin Main` (tinymist.pinMainToCurrent) to set the current file as the main file.
- Use command `Typst Unpin Main` (tinymist.unpinMain) to unset the main file.

`tinymist.pinMain` is a stateful command, and tinymist doesn't remember it between sessions (closing and opening the editor).

77.14. Passing Extra CLI Arguments

There is a **global** configuration `tinymist.typstExtraArgs` to pass extra arguments to tinymist LSP, like what you usually do with `typst-cli` CLI. For example, you can set it to `["--input=awa=1", "--input=abaaba=2", "main.typ"]` to configure `sys.inputs` and entry for compiler, which is equivalent to make LSP run like a `typst-cli` with such arguments:

```
typst watch --input=awa=1 --input=abaaba=2 main.typ
```

Supported arguments:

- entry file: The last string in the array will be treated as the entry file.
  - This is used to specify the **default** entry file for the compiler, which may be overridden by other settings.
- `--input`: Add a string key-value pair visible through `sys.inputs`.
- `--font-path` (environment variable: `TYPST_FONT_PATHS`), Font paths, maybe overridden by `tinymist.fontPaths`.

- `--ignore-system-fonts`: Ensures system fonts won't be searched, maybe overridden by `tinymist.systemFonts`.
- `--creation-timestamp` (environment variable: `SOURCE_DATE_EPOCH`): The document's creation date formatted as a UNIX timestamp.
- `--cert` (environment variable: `TYPST_CERT`): Path to CA certificate file for network access, especially for downloading typst packages.

Note: Fix entry to `main.typ` may help multiple-file projects but you may loss diagnostics and autocompletions in unrelated files.

Note: the arguments has quite low priority, and that may be overridden by other settings.

78. Contributing

You can submit issues or make PRs to GitHub.

Tinymist Neovim Support for Typst

Run and configure `tinymist` in Neovim with support for all major distros and package managers.

79. Feature Integration

- **Language service** (completion, definitions, etc.)
- **Code Formatting**
- **Live Web Preview** with typst-preview.

Work for full parity for all `tinymist` features is underway. This will include: exporting to different file types, template preview, and multifile support. Neovim integration is behind VS Code currently but should be caught up in the near future.

80. Installation

- (Recommended) mason.nvim.

```
{
  "mason-org/mason.nvim",
  opts = {
    ensure_installed = {
      "tinymist",
    },
  },
}
```

- Or manually:

  To enable LSP, you must install `tinymist`. You can find `tinymist` by:

  - Night versions available at GitHub Actions.
  - Stable versions available at GitHub Releases.
    If you are using the latest version of typst-ts-mode, then you can use command `typst-ts-lsp-download-binary` to download the latest stable binary of `tinymist` at `typst-ts-lsp-download-path`.
  - Build from source by cargo. You can also compile and install **latest** `tinymist` by Cargo.

    ```
    cargo install --git https://github.com/Myriad-Dreamin/tinymist --locked tinymist-cli
    ```

81. Configuration

- With `lspconfig`:

```
require("lspconfig")["tinymist"].setup {
    settings = {
        formatterMode = "typstyle",
        exportPdf = "onType",
        semanticTokens = "disable"
    }
}
```

- Or with `Coc.nvim`:

```
{
  "languageserver": {
    "tinymist": {
      "command": "tinymist",
      "filetypes": ["typst"],
      "settings": { ... }
    }
  }
}
```

- Or finally with the builtin lsp protocol:

```
vim.lsp.config["tinymist"] = {
    cmd = { "tinymist" },
    filetypes = { "typst" },
    settings = {
        -- ...
    }
}
```

For a full list of available settings see Tinymist Server Configuration.

82. Formatting

Either `typstyle` or `typstfmt` can be used. Both are now included in `tinymist`, so you do not need to install them separately if you only intend to use them through `tinymist`. You can select the one you prefer using the `formatterMode` setting. The following snippet shows all formatting settings that you can put in the `settings` table of the snippets from the previous section:

```
formatterMode = "typstyle", -- or "typstfmt"
formatterProseWrap = true, -- wrap lines in content mode
formatterPrintWidth = 80,  -- limit line length to 80 if possible
formatterIndentSize = 4,   -- indentation width
```

Note that, since this Neovim PR, neovim uses LSP servers (hence `tinymist` in our case) by default when formatting code. This applies to calls to `vim.lsp.buf.format()` (which formats the whole current buffer) and also to the `gq` command.

83. Live Preview

Live preview can be achieved with either a web preview or a pdf reader that supports automatic reloading (zathura is good).

Web Preview

```
-- lazy.nvim
{
  'chomosuke/typst-preview.nvim',
  lazy = false, -- or ft = 'typst'
  version = '1.*',
  opts = {}, -- lazy.nvim will implicitly calls `setup {}`
}
```

See typst-preview for more installation and configuration options.

Pdf Preview

This preview method is slower because of compilation delays, and additional delays in the pdf reader refreshing.

It is often useful to have a command that opens the current file in the reader.

```lua
vim.api.nvim_create_user_command("OpenPdf", function()
  local filepath = vim.api.nvim_buf_get_name(0)
  if filepath:match("%.typ$") then
    local pdf_path = filepath:gsub("%.typ$", ".pdf")
    vim.system({ "open", pdf_path })
  end
end, {})
```

For Neovim prior to v0.9.5, `os.execute` can be used instead. This is not suggested. See Issue #1606 for more information.

Make sure to change `exportPdf` to "onType" or "onSave".

83.1. Working under Power-Saving Mode or with Resource-consumed Projects

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting by setting the `tinymist.syntaxOnly` to `enable` or `onPowerSaving` in the configuration.

For more information about power-saving mode, see Syntax-Only Mode.

83.2. Working with Multiple-Files Projects

Tinymist cannot know the main file of a multiple-files project if you don't tell it explicitly. This causes the well-known label error when editing the `/sub.typ` file in a project like that:

```typst
// in file: /sub.typ
// Error: unknown label 'label-in-main'
@label-in-main
// in file: /main.typ
#include "sub.typ"
= Heading <label-in-main>
```

The solution is a bit internal, which should get further improvement, but you can pin a main file by command.

```lua
require("lspconfig")["tinymist"].setup { -- Alternatively, can be used `vim.lsp.config["tinymist"]`
    -- ...
    on_attach = function(client, bufnr)
        vim.keymap.set("n", "<leader>tp", function()
            client:exec_cmd({
                title = "pin",
                command = "tinymist.pinMain",
                arguments = { vim.api.nvim_buf_get_name(0) },
            }, { bufnr = bufnr })
        end, { desc = "[T]inymist [P]in", noremap = true })

        vim.keymap.set("n", "<leader>tu", function()
            client:exec_cmd({
                title = "unpin",
                command = "tinymist.pinMain",
                arguments = { vim.v.null },
            }, { bufnr = bufnr })
        end, { desc = "[T]inymist [U]npin", noremap = true })
    end,
}
```

Note that `vim.v.null` should be used instead of `nil` in the `arguments` table when unpinning. See issue #1595.

For Neovim versions prior to 0.11.0, `vim.lsp.buf.execute_command` should be used instead:

```
-- pin the main file
vim.lsp.buf.execute_command({ command = 'tinymist.pinMain', arguments = { vim.api.nvim_buf_get_name(0) } })
-- unpin the main file
vim.lsp.buf.execute_command({ command = 'tinymist.pinMain', arguments = { vim.v.null } })
```

It also doesn't remember the pinned main file across sessions, so you may need to run the command again after restarting Neovim.

This could be improved in the future.

84. Troubleshooting

Generally you can find in depth information via the `:mes` command. `:checkhealth` and `LspInfo` can also provide valuable information. Tinymist also creates a debug log that is usually at `~/.local/state/nvim/lsp.log`. Reporting bugs is welcome.

84.1. tinymist not starting when creating/opening files

This is most commonly due to nvim not recognizing the `.typ` file extension as a `typst` source file. In most cases is can be resolved with:

```
:set filetype=typst
```

In older versions of Neovim an autocommand may be necessary.

```
autocmd BufNewFile,BufRead *.typ setfiletype typst
```

85. Contributing

Please check the contributing guide for more information on how to contribute to the project.

Tinymist Emacs Support for Typst

Run and configure tinymist in Emacs for Typst.

86. Features

See Tinymist Features for a list of features.

87. Finding Executable

To enable LSP, you must install `tinymist`. You can find `tinymist` by:

- Night versions available at GitHub Actions.
- Stable versions available at GitHub Releases.
  If you are using the latest version of typst-ts-mode, then you can use command `typst-ts-lsp-download-binary` to download the latest stable binary of `tinymist` at `typst-ts-lsp-download-path`.
- Build from source by cargo. You can also compile and install **latest** `tinymist` by Cargo.

```
cargo install --git https://github.com/Myriad-Dreamin/tinymist --locked tinymist-cli
```

88. Setup Server

```
(with-eval-after-load 'eglot
  (with-eval-after-load 'typst-ts-mode
    (add-to-list 'eglot-server-programs
                 `((typst-ts-mode) .
                   ,(eglot-alternatives `(,typst-ts-lsp-download-path
                                          "tinymist"
                                          "typst-lsp"))))))
```

Above code adds `tinymist` downloaded by `typst-ts-lsp-download-binary`, `tinymist` in your PATH and `typst-lsp` in your `PATH` to the `typst-ts-mode` entry of `eglot-server-programs`.

89. Extra Settings

89.1. Configuring Language Server

You can either use `eglot-workspace-configuration` or specifying launch arguments for `tinymist`.

89.2. Working under Power-Saving Mode or with Resource-consumed Projects

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting by setting the `tinymist.syntaxOnly` to `enable` or `onPowerSaving` in the configuration.

For more information about power-saving mode, see Syntax-Only Mode.

89.2.1. eglot-workspace-configuration

For example, if you want to export PDF on save:

```
  (setq-default eglot-workspace-configuration
                '(:tinymist (:exportPdf "onSave")))
```

You can also have configuration per directory. Be sure to look at the documentation of `eglot-workspace-configuration` by `describe-symbol`..

See Tinymist Server Configuration for references.

89.2.2. Launch Arguments

For example:

```
(with-eval-after-load 'eglot
  (with-eval-after-load 'typst-ts-mode
    (add-to-list 'eglot-server-programs
                 `((typst-ts-mode) .
                   ,(eglot-alternatives `((,typst-ts-lsp-download-path "--font-path" "<your-font-path>")
                                          ("tinymist" "--font-path" "<your-font-path>")
                                          "typst-lsp"))))))
```

You can run command `tinymist help lsp` to view all available launch arguments for configuration.

Tinymist Sublime Support for Typst

Follow the instructions in the sublimelsp documentation to make it work.

90. Getting Preview Feature

Default Preview Feature and Background Preview Feature are suitable in Sublime Text.

90.1. Working under Power-Saving Mode or with Resource-consumed Projects

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting by setting the `tinymist.syntaxOnly` to `enable` or `onPowerSaving` in the configuration.

For more information about power-saving mode, see Syntax-Only Mode.

Tinymist Helix Support for Typst

Run and configure tinymist in helix for Typst.

91. Features

See Tinymist Features for a list of features.

92. Finding Executable

To enable LSP, you must install `tinymist`. You can find `tinymist` by:

- Night versions available at GitHub Actions.
- Stable versions available at GitHub Releases.
  If you are using the latest version of typst-ts-mode, then you can use command `typst-ts-lsp-download-binary` to download the latest stable binary of `tinymist` at `typst-ts-lsp-download-path`.
- Build from source by cargo. You can also compile and install **latest** `tinymist` by Cargo.

  ```
  cargo install --git https://github.com/Myriad-Dreamin/tinymist --locked tinymist-cli
  ```

93. Setup Server

Update `.config/helix/languages.toml` to use tinymist.

```
[language-server.tinymist]
command = "tinymist"

[[language]]
name = "typst"
language-servers = ["tinymist"]
```

94. Tips

94.1. Getting Preview Feature

Default Preview Feature and Background Preview Feature are suitable in helix.

94.2. Working under Power-Saving Mode or with Resource-consumed Projects

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting by setting the `tinymist.syntaxOnly` to `enable` or `onPowerSaving` in the configuration.

For more information about power-saving mode, see Syntax-Only Mode.

94.3. Working with Multiple-File Projects

There is a way in Neovim, but you cannot invoke related commands with arguments by :lsp-workspace-command in helix. As a candidate solution, assuming your having following directory layout:

```
├── .helix
│   └── languages.toml
└── main.typ
```

You could create .helix/languages.toml in the project folder with the following contents:

```
[language-server.tinymist.config]
typstExtraArgs = ["main.typ"]
```

Then all diagnostics and autocompletion will be computed according to the `main.typ`.

Note: With that configuration, if you're seeing a file that is not reachable by `main.typ`, you will not get diagnostics and autocompletion correctly in that file.

95. Extra Settings

To configure the language server, edit the `language-server.tinymist` section. For example, if you want to export PDF on typing and output files in `$root_dir/target` directory:

```
[language-server.tinymist]
command = "tinymist"
config = { exportPdf = "onType", outputPath = "$root/target/$dir/$name" }
```

To enable a live preview you can use the `preview.background`:

```
[language-server.tinymist]
command = "tinymist"
config = { preview.background.enabled = true, preview.background.args = ["--data-plane-host=127.0.0.1:23635",
"--invert-colors=never", "--open"] }
```

See Tinymist Server Configuration for references.

Tinymist Zed Support for Typst

See typst.zed.

96. Getting Preview Feature

Background Preview Feature is suitable in Zed.

96.1. Working under Power-Saving Mode or with Resource-consumed Projects

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting by setting the `tinymist.syntaxOnly` to `enable` or `onPowerSaving` in the configuration.

For more information about power-saving mode, see Syntax-Only Mode.

Part 2. Features

Command Line Interface (CLI)

The difference between typst-cli and tinymist-cli is that the latter one focuses on the features requiring code analysis or helping the language server. For example, `tinymist-cli` also provides a `compile` command, but it doesn't provide a `query` or `watch` command, which are provided by `typst-cli`. This is because `tinymist compile` also collects and saves the compilation commands needed by the language server.

1. Servers

1.1. Starting a Language Server Following LSP Protocol

To start a language server following the Language Server Protocol, please use the following command:

```
tinymist lsp
```

Or simply runs the CLI without any arguments:

```
tinymist
```

1.2. Starting a Preview Server

To start a preview server, please use the following command:

```
tinymist preview path/to/main.typ
```

See Arguments.

1.3. Starting a debug adapter Server Following DAP Protocol

To start a debug adapter following the Debug Adapter Protocol, please use the following command:

```
tinymist dap
```

2. Commands

2.1. Compiling a Document

The `tinymist compile` command is compatible with `typst compile`:

```
tinymist compile path/to/main.typ
```

To save the compilation command to the lock file:

```
tinymist compile --save-lock path/to/main.typ
```

To save the compilation command to the lock file at the path `some/tinymist.lock`:

```
tinymist compile --lockfile some/tinymist.lock path/to/main.typ
```

The lock file feature is in development. It is to help the language server to understand the structure of your projects. See Configuration: tinymist.projectResolution.

2.2. Running Tests

To run tests, you can use the `test` command, which is also compatible with `typst compile`:

```
tinymist test path/to/main.typ
```

The `test` command will defaultly run all the functions whose names are staring with `test-` related the the main file:

```
#let test-it() = []
```

See Docs: Testing Features for more information.

2.3. Generating shell completion script

To generate a bash-compatible completion script:

```
tinymist completion bash
```

Available values for the shell parameter are `bash`, `elvish`, `fig`, `fish`, `powershell`, `zsh`, and `nushell`.

Syntax-Only Mode

The syntax-only mode is available since `tinymist` v0.14.2.

When working under power-saving mode or with resource-consumed projects, typst compilations costs too much CPU and memory resources. From a simple test on a typst document with 200 pages, containing complex figures and WASM plugin calls, editing a large `.typ` file on a windows laptop (i9-12900H), the CPU and memory usage are as follows:

Mode	CPU Usage	Memory Usage (Cold Compilation)	Memory Usage (Incremental Compilation)
Normal Mode	5% ~ 12%	2.72 GB	6.62~8.73GB
Syntax-Only Mode	0% ~ 0.6%	15.0 MB	15.1~16.0 MB

You can configure the extension to run in syntax only mode, i.e. only performing elementary tasks, like syntax checking, syntax-only code analysis and formatting by setting the `tinymist.syntaxOnly` to `enable` or `onPowerSaving` in the configuration.

The syntax-only mode is known to disable or limit the functionality of the following features:

- typst preview feature.
- compilation diagnostics.
- label completion.

The syntax-only mode will be able to work with following features:

- export PDF or other formats.
- label completion.

If there are any other features that you find it work abnormally, please report issues to the GitHub Issues.

Code Documentation

Tinymist will read the documentation from the source code and display it in the editor. For example, you can hover over a identifier to see its documentation, usually the content of the comments above the identifier's definition. The format of the documentation follows this guideline.

Note: the feature is not yet officially supported.

3. Status of the Feature

- √ Syntax of Docstring's Content: We have reached consensus on the syntax of content. It MUST be written in Typst.
- ? Annotations in Docstring's Content: We check the annotations in docstring by tidy style (**< v0.4.0**). It's not an official standard.
  - Related Issue: Tidy #53: Long-term compatibility with `tinymist`.
- ✗ Syntax of Docstring: We haven't reached consensus on the syntax of docstring. It's not clear whether we should distinguish the docstring from regular comments.

4. Format of Docstring

A docstring is an object in source code associating with some typst definition, whose content is the documentation information of the definition. Documentation is placed on consecutive special comments using three forward slashes `///` and an optional space. These are called doc comments.

While the `DocCommentMatcher` matches doc comments in a looser way, we recommend using the strict syntax mentioned in the following sections.

4.1. Example 1

The content MUST follow typst syntax instead of markdown syntax.

```
/// You can use *typst markup* in docstring.
#let foo = 1;
```

Explanation: The documentation of `foo` is "You can use **typst markup** in docstring."

4.2. Example 2

The comments SHOULD be **line** comments starting with **three** forward slashes `///` and an optional space.

```
/* I'm a regular comment */
#let foo = 1;
// I'm a regular comment.
#let foo = 1;
//// I'm a regular comment.
#let foo = 1;
```

Explanation: There SHOULD be no documentation for `foo` in the three cases. The first comment is not a line comment, the second and the third one don't start with exact three forward slashes. However, the language server will regard them as doc comments loosely.

4.3. Example 3

The comments SHOULD be consecutively and exactly placed aside the associating definition.

```
/// 1
/// 2
#let foo = 1;
```

Explanation: The documentation of `foo` is `"1\\n2"`.

```
/// 1

/// 2
#let bar = 1;
```

Explanation: The documentation of `bar` is `"2"`, because there is a space between `/// 1` and `/// 2`.

```
/// 1
/// 2

#let baz = 1;
```

Explanation: There SHOULD be no documentation for `baz`, because the comments is not exactly placed before the let statement of the `baz`.

4.4. Module-Level Docstring

A module-level appears at the beginning of the module (file).

4.5. Example 4

Given a file `foo.typ` containing code:

```
/// 1

/// 2
#let baz = 1;
```

Explanation: The documentation of the module `foo` (`foo.typ`) is `"1"`. It is not `"1\n2"`, because there is a space between `/// 1` and `/// 2`.

4.6. Example 5

Given a file `foo.typ` containing code:

```
// License: Apache 2.0
/// 1
```

Explanation: The documentation of the module `foo` (`foo.typ`) is `"1"`. It is not `"License: Apache 2.0\n1"`, because `// License: Apache 2.0` is not a strict doc comment.

4.7. Variable Docstring

A variable appears exactly before some let statement (the ast starting with `#let` or `let`). BNF Syntax:

```
VAR_DOCSTRING_CONTENT ::= MARKUP { VAR_SUB_ANNOATATION } [ VAR_INIT_ANNOATATION ]
```

4.8. Example 6

You can use an arrow `->` following a type annotation to mark the type of the *initializer expression* of the let statement. The *initializer expression* is the expression at the right side of the equal marker in the let statement. BNF Syntax:

```
VAR_INIT_ANNOATATION ::= '-> ' TYPE_ANNOATATION
```

```
/// -> int
#let f(x) = { /* code */ };
```

Explanation: The docstring tells that the type of `{ /* code */ }` is `int`. Thus, the **resultant type** of the function `f` is also annotated as `int`.

```
/// -> float
#let G = { /* code */ };
```

Explanation: The docstring tells that the type of `{ /* code */ }` is `float`. Thus, the **type** of the variable `G` is also annotated as `float`.

4.9. Example 7

You can use a list item `- name (type): description` to document the related variable at the left side of the let statement. BNF Syntax:

```
VAR_SUB_ANNOATATION ::= '- ' NAME '(' TYPE_ANNOATATION ')' ':' MARKUP
```

```
/// - x (int): The input of the function `f`.
#let f(x, y) = { /* code */ };
```

Explanation: The docstring tells that the type of `x` is `int` and the documentation of `x` is "The input of the function `f`."

```
/// - x (any): The swapped value from `y`.
#let (x, y) = (y, x);
```

Explanation: The docstring tells that the type of `x` at the left side is `any` and its documentation is "The swapped value from `y`." The variables at the right side of the let statement are not documented by the docstring.

4.10. Examples in Docstrings

You can use `#example` function to provide examples in docstrings.

4.11. Example 8

```
#example(`
$ sum f(x) = 10 $
`)
```

The docstring tells that there is an associated example in the docstring. It will be rendered as a code block following the rendered result when possible:

```
$ sum f(x) = 10 $
```

$$\sum f(x) = 10$$

4.12. Type Annotations in Docstrings

A type annotation is a comma separated list containing types. BNF Syntax:

```
TYPE_ANNOATATION ::= TYPE { ',' TYPE }
```

Currently, only built-in types and the generic array type are supported in docstrings.

The list of built-in types:

- `any`
- `content`
- `none`
- `auto`
- `bool` or `boolean`
- `false`
- `true`

- `int` or `integer`
- `float`
- `length`
- `angle`
- `ratio`
- `relative`
- `fraction`
- `str` or `string`
- `color`
- `gradient`
- `pattern`
- `symbol`
- `version`
- `bytes`
- `label`
- `datetime`
- `duration`
- `styles`
- `array`
- `dictionary`
- `function`
- `arguments`
- `type`
- `module`
- `plugin`

Code Completion

5. Using LSP-Based Completion

LSP will serve completion if you enter *trigger characters* in the editor. Currently, the trigger characters are:

1. any valid identifier character, like `'a'` or `'Z'`.
2. `'#'`, `'('`, `'<'`, `'.'`, `':'`, `'/'`, `'"'`, `'@'`, which is configured by LSP server.

5.0.1. VSCode:

Besides, you can trigger the completion manually by pressing `Ctrl+Space` in the editor.

If `Ctrl+Space` doesn't work, please check your IME settings or keybindings.

When an item is selected, it will be committed if some character is typed.

1. press `Esc` to avoid commit.
1. press `Enter` to commit one.
2. press `'.'` to commit one for those that can interact with the dot operator.
3. press `';'` to commit one in code mode.
4. press `','` to commit one in list.

5.1. Label Completion

The LSP will keep watching and compiling your documents to get available labels for completion. Thus, if it takes a long time to compile your document, there will be an expected delay after each editing labels in document.

A frequently asked question is how to completing labels in sub files when writing in a multiple-file project. By default, you will not get labels from other files, e.g. bibiliography configured in other files. This is because the "main file" will be tracked when your are switching the focused files. Hence, the solution is to set up the main file correctly for the multi-file project.

5.1.1. VSCode:

See VS Code: Working with Multiple File Projects.

5.1.2. Neovim:

See Heovim: Working with Multiple File Projects.

5.1.3. Helix:

See Helix: Working with Multiple File Projects.

6. Using Snippet-Based Completion

6.0.1. VSCode:

We suggest to use snippet extensions powered by TextMate Scopes. For example, HyperSnips provides context-sensitive snippet completion.

Exporting Documents

You can export your documents to various formats using the `export` feature.

7. Export from Query Result

7.1. Hello World Example (VSCode Tasks)

You can export the result of a query as text using the `export` command.

Given a code:

```
#println("Hello World!")
#println("Hello World! Again...")
```

LSP should export the result of the query as text with the following content:

```
Hello World!
Hello World! Again...
```

This requires the following configuration in your `tasks.json` file:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Query as Text",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "query",
        "query.format": "txt",
        "query.outputExtension": "out",
        "query.field": "value",
        "query.selector": "<print-effect>",
        "query.one": true
      }
    },
  ]
}
```

See the Sample Workspace: print-state for more details.

7.2. IEEE Example (VSCode Tasks)

This workspace gives an example to create and prepare IEEE papers using Typst, which exports typst documents to LaTeX.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Export to LaTeX (IEEE)",
      "type": "typst",
      "command": "export",
      "export": {
        "format": ["tex"],
        // It is totally legal to use the processor in the current workspace,
        // but we suggest make a local package and use the package spec instead.
        // like: "package": "@local/ieee-tex:0.1.0"
        "processor": "/ieee-tex.typ",
        "assetsPath": "${workspaceFolder}/target"
      }
    },
    // todo: not working
    {
      "label": "Export to Word (IEEE)",
      "type": "typst",
      "command": "export",
      "export": {
        "format": ["word"]
      }
    }
  ]
}
```

It uses `export` to export the document to LaTeX.

See the Sample Workspace: IEEE Paper for more details.

7.3. Pdfpc Example (VSCode Tasks)

A more practical example is exporting the result of a query as a pdfpc file. You can use the following configuration in your `tasks.json` file to export the result of a query as a pdfpc file, which is adapted by Touying Slides.

```
{
  "label": "Query as Pdfpc",
  "type": "typst",
  "command": "export",
  "export": {
    "format": "query",
    "query.format": "json",
    "query.outputExtension": "pdfpc",
    "query.selector": "<pdfpc-file>",
    "query.field": "value",
    "query.one": true
  }
}
```

To simplify configuration,

```
{
  "label": "Query as Pdfpc",
  "type": "typst",
  "command": "export",
  "export": {
    "format": "pdfpc"
  }
}
```

8. VSCode: Task Configuration

You can configure tasks in your `tasks.json` file to “persist” the arguments for exporting documents.

Example:

```jsonc
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Export as Html",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "html"
      }
    },
    {
      "label": "Export as Markdown",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "markdown"
      }
    },
    {
      "label": "Export as Plain Text",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "html"
      }
    },
    {
      "label": "Export as SVG",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "svg",
        "merged": true
      }
    },
    {
      "label": "Export as PNG",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "png",
        // Default fill is white, but you can set it to transparent.
        "fill": "#00000000",
        "merged": true
      }
    },
    {
      "label": "Query as Pdfpc",
      "type": "typst",
      "command": "export",
      "export": {
        "format": "pdfpc"
      }
    },
    {
      "label": "Export as PNG and SVG",
      "type": "typst",
      "command": "export",
      "export": {
        // You can export multiple formats at once.
        "format": ["png", "svg"],
        // To make a visual effect, we set an obvious low resolution.
        // For a nice result, you should set a higher resolution like 288.
        "png.ppi": 24,
        "merged": true,
        // To make a visual effect, we set an obvious huge gap.
        // For a nice result, you should set a smaller gap like 10pt.
        "merged.gap": "100pt"
      }
    }
  ]
}
```

todo: documenting export options.

```
[
  {
    "type": "typst",
    "required": [
      "command"
    ],
    "properties": {
      "command": {
        "type": "string",
        "default": "export",
        "description": "The command to run.",
        "enum": [
          "export"
        ],
        "enumDescriptions": [
          "Export the document to specific format."
        ]
      },
      "export": {
        "type": "object",
        "description": "Arguments for `export` command.",
        "properties": {
          "format": {
            "description": "The format(s) to export the document to.",
            "oneOf": [
              {
                "type": "string",
                "description": "The format to export the document to.",
                "enum": [
                  "pdf",
                  "png",
                  "svg",
                  "html",
                  "markdown",
                  "tex",
                  "text",
                  "query",
                  "pdfpc"
                ],
                "enumDescriptions": [
                  "PDF",
                  "PNG",
                  "SVG",
                  "HTML",
                  "Markdown",
                  "TeX",
                  "Plain Text",
                  "Query Result",
                  "Pdfpc (From Query)"
                ],
                "default": "pdf"
              },
              {
                "type": "array",
                "description": "The formats to export the document to.",
                "items": {
                  "type": "string",
                  "description": "The format to export the document to.",
                  "enum": [
                    "pdf",
                    "png",
                    "svg",
                    "html",
                    "markdown",
                    "tex",
                    "text",
                    "query",
                    "pdfpc"
                  ],
                  "enumDescriptions": [
                    "PDF",
                    "PNG",
                    "SVG",
                    "HTML",
                    "Markdown",
                    "TeX",
                    "Plain Text",
                    "Query Result",
                    "Pdfpc (From Query)"
                  ],
                  "default": "pdf"
                }
```

After configuring the tasks, you can run them using the command palette.

1. Press `Ctrl+Shift+P` to open the command palette.
2. Type `Run Task` and select the task you want to run.
3. Select the task you want to run.

9. Neovim: Export Commands

You can call the following export commands.

- `tinymist.exportSvg`
- `tinymist.exportPng`
- `tinymist.exportPdf`
- `tinymist.exportHtml`
- `tinymist.exportMarkdown`
- `tinymist.exportTeX`
- `tinymist.exportText`
- `tinymist.exportQuery`

The first argument is the path to the file you want to export and the second argument is an object containing additional options.

Exporting to Other Markup Formats

This feature is currently in early development.

typlite is a pure Rust library for converting Typst documents to other markup formats.

typlite's goal is to convert docstrings in typst packages to LSP docs (Markdown format). To achieve this, it runs HTML export and extracts semantic information from the HTML document for markup conversion.

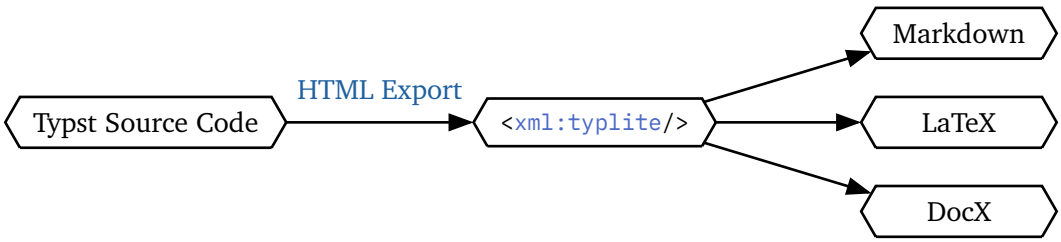

Figure 1: The conversion path from typst source code to other markup formats, by typlite.

10. TodoList

- [ ] Renders figures into PDF instead of SVG.
- [ ] Converts typst equations, might use texmath or mathyml plus pmml2tex.

11. Example: Writing README in typst

To export to Markdown, run the command:

```
typlite README.typ README.md --assets-path assets
```

11.1. Assets Path

You might love to use `html.frame` to render typst examples in `README.md`. By default, the examples are embedded in the output by data url. To externalize them, please specify the `--assets-path` option.

11.2. Implementing target-aware functions

typlite will set `sys.inputs.x-target` to `md` if it is exporting to Markdown. You can use this variable to implement target-aware functions in your typst documents.

```
#let x-target = sys.inputs.at("x-target", default: "pdf")
#let is-md-target = x-target == "md"
```

For example, you can implement a GitHub link function, which determines the link based on the target:

```
#let current-revision = read("/.git/" + read("/.git/HEAD").trim().slice(5)).trim()

#let github-link(path, body, kind: none) = {
  let dest = if is-md-target {
    path
  } else {
    if kind == none {
      kind = if path.ends-with("/") { "tree" } else { "blob" }
    }
    (remote, kind, current-revision, path).join("/")
  }

  link(dest, body)
}
```

12. Example: Styling a Typst Document by IEEE LaTeX Template

The `main.typ` in the Sample Workspace: IEEE Paper can be converted perfectly.

- Run the command:

  ```
  typlite main.typ main.tex --processor "/ieee-tex.typ"
  ```

- Create a project on Overleaf, using the IEEE LaTeX Template.
- Upload the `main.tex` file and exported PDF assets and it will get rendered and ready to submit.

12.1. Processor Scripts

The CLI command in the previous example uses the `--processor "/ieee-tex.typ"` option, which is not a flag of the official `typst-cli`. The option tells typlite to use a processor script to process the HTML export result for LaTeX export.

typlite will show your main documents with the `article` function obtained from the processor file.

```
#let verbatim(body) = {
  show raw.where(lang: "tex"): it => html.elem("m1verbatim", attrs: (src: it.text))
  body
}
#let article(body) = {
  verbatim(```tex
  %% Generated by typlite
  %% Please report to https://github.com/Myriad-Dreamin/tinymist if you find any bugs.

  \begin{document}
  Hey,
  ```)
  body
  verbatim(```tex
  \end{document}
  ```)
}
```

Currently, `html.elem("m1verbatim")` is the only `xml` element can be used by processor scripts. When seeing a `<m1verbatim/>` element, typlite writes the inner content to output directly.

It will output like that:

```
%% Generated by typlite
%% Please report to https://github.com/Myriad-Dreamin/tinymist if you find any bugs.

\begin{document}
Hey, [CONTENT generated according to body]
\end{document}
```

You can implement the `article` function for different markup formats, such as LaTeX, Markdown, DocX, and Plain Text.

12.2. Using Processor Packages

The processor script can be not only a file, but also a package:

- From current workspace: `"/ieee-tex.typ"` is the file relative to the current workspace root.
- From a package: `"@local/ieee-tex:0.1.0"` or `"@preview/ieee-tex:0.1.0"` can be used to get functions from local packages or typst universe.

13. Perfect Conversion

typlite is called "-ite" because it only ensures that nice docstrings are converted perfectly. Similarly, if your document looks nice, typlite can also convert it to other markup formats perfectly.

This introduces concept of *Semantic Typst*. To help conversion, you should separate styling scripts and semantic content in your typst documents.

A good example in HTML is `<strong>` v.s. `<b>`. Written in typst,

```
#strong[Good Content]
#text(weight: 700)[Bad Content]
```

typlite can convert "Good Content" perfectly, but not "Bad Content". This is because we can attach markup-specific styles to "Good Content" then, but "Bad Content" may be broken by some reasons, such as failing to find font weight when rendering the content.

To style your typst documents, rewriting the bad content by show rule is suggested:

```
#show strong: it => if is-md-target {
  // style for Markdown target, for example:
  html.span(class: "my-strong", it.body)
} else { // style for builtin Typst targets:
  text(weight: 700, it.body)
}
#strong[Bad Content]
```

typlite will feel happy and make perfect conversion if you keep aware of keep pure semantics of `main.typ` documents in the above ways.

14. Implementing `abstract` for IEEE LaTeX Template

Let's explain how `/ieee-tex.typ` works by the `abstract` example. First, we edit `/ieee-template.typ` to store the abstract in state:

```
#let abstract-state = state("tex:abstract", "")
#let abstract(body) = if is-md-target {
  abstract-state.update(_ => body)
} else {
    // fallback to regular abstract
}
```

`is-md-target` already distinguishes regular typst PDF export and typlite export (which uses HTML export). We haven't decide a way to let your template aware of LaTeX export.

Luckily, typst has `sys.input` mechanism so you can distinguish it by yourself:

```
// typst compile or typlite main.typ --input exporter=typlite-tex
#let exporter = sys.inputs.at("exporter", default: "typst")
#exporter // "typst" or "typlite-tex"
```

Or define a state that shares between the template and the processor script:

```
#let is-typlite = state("is-typlite", false)
// processor.typ
#let article(body) = {
  is-typlite.update(_ => true)
  body
}
// template.typ
#let is-typlite = ...
#let abstract(body) = context if is-typlite {
  abstract-state.update(_ => body)
} else {
  // fallback to regular abstract
}
```

Next, in the `ieee-tex.typ`, we can get the `abstract` material from the state and render it in the LaTeX template:

```
#let abstract = context {
  let abstract-body = state("tex:abstract", "").final()
  verbatim(```tex
  % As a general rule, do not put math, special symbols or citations
  % in the abstract
  \begin{abstract}
  ```)
  abstract-body
  verbatim(```tex
  \end{abstract}
  ```)
}
```

We don't extracts content from `abstract-body`. Inteadly, we put the body directly in the document, to let typlite process the equations in `abstract-body` and convert them to LaTeX.

Hook Scripts

The hook script feature is available since `tinymist` v0.14.2.

Hook Scripts allow you to hook and customize certain behaviors of tinymist by providing code snippets that will be executed at specific events.

The hook scripts are run as typst scripts with some predefined variables. Since typst is sandboxed, the hook scripts cannot access system directly. However, you can still bind lsp commands to perform complex operations.

The following example demonstrates how to customize the paste behavior when pasting resources into the editing typst document.

- First, the editor client (VS Code) invokes the `tinymist.onPaste` and gets the action $\mathcal{A}$ to perform. the action $\mathcal{A}$ contains two fields `dir`, and `onConflict`, where `dir` is the directory to store the pasted resources, and `onConflict` is the script to execute when a conflict occurs (i.e., the file already exists). Note that `import` statements are allowed in the paste script.
- Then, the editor client checks the physical file system. If it detects conflict when creating a resource file, it again runs the `onConflict` script to determine next behavior.
- After several iterations, the editor client finally creates the resource files and inserts the corresponding Typst code into the document.

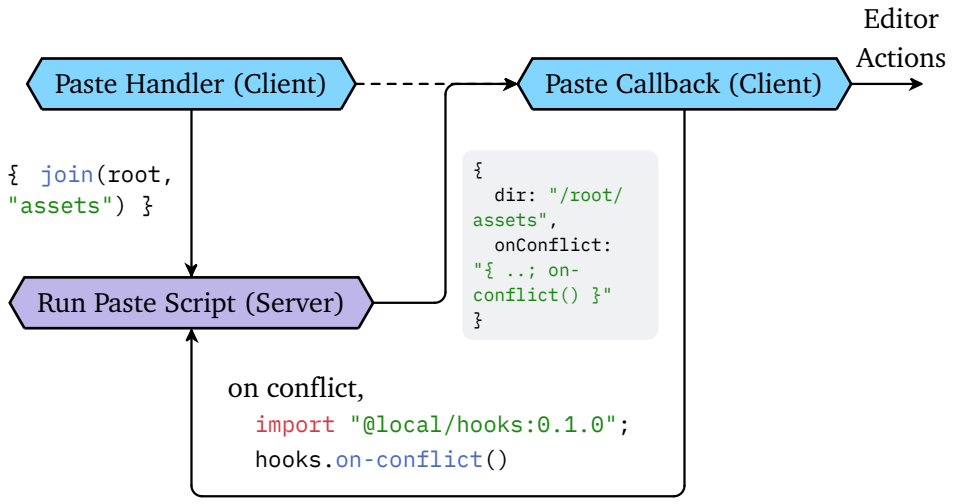

Figure 2: The workflow of running `tinymist.onPaste`

Specifically, three script hooks will be supported:

- Hook on Paste: customize the paste behavior when pasting resources into the editing typst document.
- Hook on Watch: customize the watch behavior when a file change is detected in the workspaces.
- Hook on Generating Code Actions and Lenses: adding additional code actions by typst scripting.

15. Customizing Paste Behavior (for VS Code)

You could configure `tinymist.onPaste` to customize the paste behavior. It will be executed when pasting resources into the editing typst document. Two kinds of script code are supported:

- If the script code starts with `{` and ends with `}`, it will be evaluated as a typst code expression.

- Otherwise, it will be evaluated as a path pattern.

15.1. Path Pattern (Stable)

When evaluated as a path pattern, the path variables could be used:

- `root`: the root of the workspace.
- `dir`: the directory of the current file.
- `name`: the name of the current file.

For example, the following path pattern

```
$root/assets
```

is evaluated as `/path/to/root/assets` when pasting `main.typ` in `/path/to/root/dir/main.typ`.

15.2. Code Expression (Experimental)

When evaluated as a typst code expression, the script could use following predefined definitions:

- `root`: the root of the workspace.
- `dir`: the directory of the current file.
- `name`: the name of the current file.
- `join`: a function to join path segments, e.g. `join("a", "b", "c")` returns `a/b/c` on Unix-like systems.

For example, the following paste script

```
{ join(root, "x", dir, if name.ends-with(".png") ("imgs"), name) }
```

is evaluated as `/path/to/root/dir/imgs/main` when pasting `main.png` in `/path/to/root/dir/main.typ`.

The result of the paste script could also be a dictionary with the following fields:

- `dir`: the directory to store the pasted resources.

If the result is a string, it will be treated as the `dir` field, i.e. `{ dir: <result> }` and the editor client will creates the pasted resource files in the specified directory.

More fields will be supported in the future. If you have any suggestions, please feel free to open an issue.

16. Customizing Export Behavior (for all Editors, Experimental)

Note: this is not implemented yet in current version.

You could configure `tinymist.onWatch` to customize the watch behavior. It will be executed when a file change is detected in the workspace.

For example, debouncing time:

```
{ debounce("100ms") }
```

For example, debounce and

- export by a custom handler and postprocess using `ghostscript`,
- export cover (first page) as SVG.

```
{ if debounce("1000ms") { (
  ( command: "myExtension.pdfWithGhostScript" ),
  ( export: "svg", pages: "1" ),
) } }
```

And define a custom command at client side (VS Code):

```
async function pdfWithGhostScript() {
  const pdfPath = await vscode.commands.execute("tinymist.exportPdf");
  const outputPath = pdfPath.replace(/\.pdf$/, "-gs.pdf");
  return new Promise((resolve, reject) => {
    exec(`gs ... -sOutputFile=${outputPath} ${pdfPath}`, (error, stdout, stderr) => {
      if (error) {
        reject(error);
      } else {
        resolve(outputPath);
      }
    });
  });
}
```

Hint: you could create your own vscode extension to define such custom commands.

17. Providing Package-Specific Code Actions (for all Editors, Experimental)

Note: this is not implemented yet in current version.

You could configure `tinymist.onCodeAction` to provide package-specific code actions. It will be executed when requesting code actions in the editing typst document.

For example, matching a table element and providing a code lens to open it in a Microsoft Excel:

```
{
  code-lens(
    if: ```typc is-func and func.name == "table"```,
    title: "Open in Excel",
    command: "myExtension.openTableInExcel",
    arguments: (sys.inputs,),
  )
}
```

Preview Feature

Two ways of previewing a Typst document are provided:

- PDF Preview: let lsp export your PDF on typed, and open related PDF by your favorite PDF viewer.
- Web (SVG) Preview: use builtin preview feature.

Whenever you can get a web preview feature, it is recommended since it is much faster than PDF preview and provides bidirectional navigation feature, allowing jumping between the source code and the preview by clicking or lsp commands.

18. PDF Preview

For non-vscode clients, Neovim client as an example. One who uses `nvim-lspconfig` can place their configuration in the `servers.tinymist.settings` section. If you want to export PDF on typing and output files in `$root_dir/target` directory, please configure it like that:

```
return {
  -- add tinymist to lspconfig
  {
    "neovim/nvim-lspconfig",
    opts = {
      servers = {
        tinymist = {
          settings = {
            exportPdf = "onType",
            outputPath = "$root/target/$dir/$name",
          }
        },
      },
    },
  },
}
```

18.0.1. VSCode:

The corresponding configuration should be placed in the `settings.json` file. For example:

```
{
  "tinymist.exportPdf": "onType",
  "tinymist.outputPath": "$root/target/$dir/$name"
}
```

Also see:

- VS Cod(e,ium): Tinymist Server Configuration
- Neovim: Tinymist Server Configuration

19. Builtin Preview Feature

19.1. Using `tinymist.startDefaultPreview` Command (Since Tinymist v0.13.6)

You can use `tinymist.startDefaultPreview` command to start a preview instance without arguments. This is used for the case where a client cannot pass arguments to the preview command, e.g. helix. Default Behaviors:

- The preview server listens on a random port.
- The colors are inverted according to the browser (usually also system) settings.
- The preview follows an inferred focused file from the requests from the client.

- The preview is opened in the default browser.

You can set the arguments to used by configuration `tinymist.preview.browsing.args` to **override** the default behavior. The default value is `["--data-plane-host=127.0.0.1:0", "--invert-colors=auto", "--open"]`. Intentionally, the name of the configuration is **not** `tinymist.defaultPreviewArgs` or `tinymist.preview.defaultArgs` to avoid confusion.

19.2. Running preview server in background (Since Tinymist v0.13.6)

You can start a preview instance in background with configuration:

```
{
  "tinymist.preview.background.enabled": true,
}
```

Default Behaviors:

- The preview server listens on `127.0.0.1:23635`.
- The colors are inverted according to the browser (usually also system) settings.
- The preview follows an inferred focused file from the requests from the client.

You can set the arguments to used by configuration `tinymist.preview.background.args` to **override** the default behavior. The default value is `["--data-plane-host=127.0.0.1:23635", "--invert-colors=auto"]`. Example:

```
{
  "tinymist.preview.background.args": ["--data-plane-host=127.0.0.1:23635", "--invert-colors=never"],
}
```

19.3. CLI Integration

```
typst-preview /abs-path/to/main.typ --partial-rendering
```

is equivalent to

```
tinymist preview /abs-path/to/main.typ --partial-rendering
```

19.4. Editor Integration

19.4.1. VSCode:

The preview feature is integrated into the tinymist extension.

19.4.2. Neovim:

You may seek typst-preview.nvim for the preview feature.

19.4.3. Emacs:

You may seek typst-preview.el for the preview feature.

19.5. `sys.inputs`

If the document is compiled by lsp, you can use `sys.inputs` to get the preview arguments:

```
#let preview-args = json.decode(sys.inputs.at("x-preview", default: "{}"))
```

There is a `version` field in the `preview-args` object, which will increase when the scheme of the preview arguments is changed.

```
#let version = preview-args.at("version", default: 0)
#if version <= 1 {
  assert(preview-args.at("theme", default: "light") in ("light", "dark"))
}
```

19.6. Developer Guide

See Typst-Preview Developer Guide.

19.6.1. Theme-aware template

The only two abstracted theme kinds are supported: `light` and `dark`. You can use the following code to get the theme:

```
#let preview-theme = preview-args.at("theme", default: "light")
```

Testing Feature

The testing feature is available since `tinymist` v0.13.10.

20. IDE Support

You can run tests and check coverage in the IDE or CLI.

21. Test Discovery

Given a file, tinymist will try to discover tests related to the file.

- All dependent files in the same workspace will be checked.
  - For example, if file `a.typ` contains `import "b.typ"` or `include "b.typ"`, tinymist will check `b.typ` for tests as well.
- For each file including the entry file itself, tinymist will check the file for tests.
  - If a file is named `example-*.typ`, it is considered an **example document** and will be compiled using `typst_shim::compile_opt`.
    - Both png export and html export may be called.
    - For now, png export is always called for each example file.
    - If the label `<test-html-example>` can be found in the example file, html export will be called.
  - Top-level functions will be checked for tests.
    - If a function is named `test-*`, it is considered a test function and will be called directly.
    - If a function is named `bench-*`, it is considered a benchmark function and will be called once to collect coverage.
    - If a function is named `panic-on-*`, it will only pass the test if a panic occurs during execution.

Example Entry File:

```
#import "example-hello-world.typ"

#let test-it() = {
  "test"
}

#let panic-on-panic() = {
  panic("this is a panic")
}
```

Example Output:

```
Running example(example-hello-world)
Running test(test-it)
Running test(panic-on-panic)
 Passed test(test-it)
 Passed test(panic-on-panic)
 Passed example(example-hello-world)
   Info All test cases passed...
```

22. Benchmarking

Since it requires some heavy framework to run benchmarks, a standalone tool is provided to run benchmarks.

Check crityp for more information.

23. Visualizing Coverage

- Run and collect file coverage using command `tinymist.profileCurrentFileCoverage` in VS Cod(e,ium).
- Run and collect test coverage using command `tinymist.profileCurrentTestCoverage` in VS Cod(e,ium).
  - Check Test Discovery to learn how tinymist discovers tests.

VS Cod(e,ium) will show the overall coverage in the editor.

24. CLI Support

You can run tests and check coverage in the CLI.

```
tinymist test tests/main.typ
...
  Info: All test cases passed...
```

You can pass same arguments as `typst compile` to `tinymist test`.

To watch for changes and run tests automatically, use the `--watch` option:

```
tinymist test --watch tests/main.typ
```

When running in the watch mode, input command lines to control the test runner:

```
   Hint Press 'h' for help
$ h
h/r/u/c/q: help/run/update/quit
```

For example, according to the help message, update the reference files using the command `u` (update).

25. Collecting Coverage with CLI

You can collect coverage using the `--coverage` option.

```
tinymist test tests/main.typ --coverage
...
   Info Written coverage to target/coverage.json ...
    Cov Coverage Summary 9/10 (90.00%)
   Info All test cases passed...
```

Use `--print-coverage=full` to print the coverage of each file.

```
tinymist test tests/main.typ --coverage --print-coverage=full
...
    Cov  6     / 6     (100.00%)  tests/example-hello-world.typ
    Cov  3     / 4     ( 75.00%)  tests/main.typ
    Cov Coverage Summary 9/10 (90.00%)
   Info All test cases passed...
```

26. Debugging tests with CLI

If any test fails, the CLI will return a non-zero exit code.

```
tinymist test tests/main.typ
...
 Fatal: Some test cases failed...
```

To update the reference files, you can run:

```
tinymist test tests/main.typ --update
```

To get image files to diff you can use grep to find the image files to update:

```
tinymist test tests/main.typ 2> >(grep Hint) > >(grep "compare image")
   Hint example(example-hello-world): compare image at refs/png/example-hello-world.png
   Hint example(example-other): compare image at refs/png/example-other.png
```

You can use your favorite image `diff` tool to compare the images, e.g. `magick compare`.

27. Tips: Reproducible Rendering

To ensure that the rendering is reproducible, you can ignore system fonts.

```
tinymist test tests/main.typ --ignore-system-fonts
```

Adds font paths using the `--font-paths` option if you want to use custom fonts:

```
tinymist test tests/main.typ --font-paths /path/to/fonts
```

28. Continuous Integration

`tinymist test` only compares hash files to check whether content is changed. Therefore, you can ignore rendered files and only keep the hash files to compare them on CI. Putting the following content in `.gitignore` will help you to ignore the files:

```
# png files
refs/png/**/*.png
# html files
refs/html/**/*.html
# hash files
!refs/**/*.hash
```

Install `tinymist` on CI and run `tinymist test` to check whether the content is changed.

```
- name: Install tinymist
  env:
    TINYMIST_VERSION: 0.13.x # to test with typst compiler v0.13.x, tinymist v0.14.x for typst v0.14.x, and so
on.
  run: curl --proto '=https' --tlsv1.2 -LsSf https://github.com/Myriad-Dreamin/tinymist/releases/download/
${TINYMIST_VERSION}/tinymist-installer.sh | sh
- name: Run tests (Typst)
  run: tinymist test tests/main.typ --root . --ppi 144 --ignore-system-fonts
- name: Upload artifacts
  uses: actions/upload-artifact@v4
  with:
    name: refs
    path: refs
```

Linting Feature

The linting feature is available since `tinymist` v0.13.12.

If enabled, the linter will run on save or on type, depending on your configuration. When it finishes, the language server will send the results along with the compilation diagnostics to the editor.

29. Configuring in VS Code

1. Open settings.
2. Search for “Tinymist Lint” and modify the value.
    1. Toggle “Enabled” to enable or disable the linter.
    2. Change “When” to configure when the linter runs.
        - (Default) `onSave` run linting when you save the file.
        - `onType` run linting as you type.

29.1. Configuring in Other Editors

1. Change configuration `tinymist.lint.enabled` to `true` to enable the linter.
2. Change configuration `tinymist.lint.when` to `onSave` or `onType` to configure when the linter runs.
    - (Default) `onSave` run linting when you save the file.
    - `onType` run linting as you type.

Project Model

This section documents the experimental project management feature. Implementation may change in future releases.

30. The Core Configuration: `tinymist.projectResolution`

This setting controls how Tinymist resolves projects:

- **`singleFile` (Default)**: Treats each Typst file as an independent document (similar to Markdown workflows). No lock files or project caches are generated, which is suitable for most people who work with single Typst files or small projects.
- **`lockDatabase`**: Mimics Rust's project management by tracking compilation/preview history. Stores data in lock files (`tinymist.lock`) and cache directories, enabling automatic main file selection based on historical context.

31. The Challenge to Handle Multiple-File Projects

When working with multiple-file projects, Tinymist faces the challenge of determining which file to use as the main file for compilation and preview. This is because:

1. All project files (entries, includes, imports) share `.typ` extensions.
2. No inherent distinction exists between entry files and dependencies.
3. Automatic detection is ambiguous without context.

This resembles the situation in C++, where the language server also struggles to determine the header files and source files in a project. In C++, the language servers and IDEs relies on the `compile_commands.json` file to understand the compilation context.

Inspired by C++ and Rust, we introduced the `lockDatabase` resolution method to relieve pain of handling multiple-file projects.

32. The classic way: `singleFile`

This is the default resolution method and has been used for years. Despite using `singleFile`, you can still work with multiple files:

- The language server will serve the previewed file as the main file when previewing a file.
- Pinning a main file manually by commands is possible:
  - Use command `Typst Pin Main` (tinymist.pinMainToCurrent) to set the current file as the main file.
  - Use command `Typst Unpin Main` (tinymist.unpinMain) to unset the main file.

33. A Sample Usage of `lockDatabase`

This feature is in early development stage, and may contain bugs or incomplete features. The following sample demonstrates how to use the `lockDatabase` resolution method. Here is the related test.

1. Set `projectResolution = "lockDatabase"` in LSP settings.
2. Like `scripts/test-lock.sh`, compile a file using tinymist CLI with `--save-lock` flag: `tinymist compile --save-lock main.typ`. This will create a `tinymist.lock` file in the current directory, which contains the *Compilation History* and project routes.
3. back to the editor, editing `chapters/chapter1.typ` will trigger PDF export of `main.typ` automatically.

Please report issue on GitHub if you find any bugs, missing features, or have any questions about this feature.

34. Stability Notice: `tinymist.lock`

We have been aware of backward compatibility issues, but any change of the schema of `tinymist.lock` may corrupt the `tinymist.lock` intentionally or unintentionally. The schema is unstable and in beta stage. You have to remove the `tinymist.lock` file to recovery from errors, and you could open an issue to discuss with us. To reliably keep compilation commands, please put `tinymist compile` commands in build system such as `Makefile` or `justfile`.

35. Compilation History

The *Project Model* only relies on the concept of *Compilation History* and we will explain how it works and how to use.

The *Compilation History* (`tinymist.lock`) is a set of records. Each record contains the following information about compilation:

- **Input Args**: Main file path, fonts, features (e.g., HTML/Paged as export target).
- **Export Args**: Output path, PDF standard to use.

The source of *Compilation History*:

- (Implemented) CLI commands: `tinymist compile/preview --save-lock`, suitable for all the editor clients.
- (Not Implemented) LSP commands: `tinymist.exportXXX/previewXXX`, suitable for vscode or neovim clients, which allows client-side extension.
- (Not Implemented) External tools: Tools that update the lock file directly. For example, the tytanic could update all example documents once executed test commands. The official typst could also do this to tell whether a test case is compiled targeting HTML or PDF.

35.1. Utilizing Compilation History

There are several features that can be implemented using the *Compilation History*:

- Correct Entry Detection: When a user runs the compile or preview commands, Tinymist will save the *Compilation History* to the lock file and the language server will identify the main file correctly.
- Dynamic Entry Switching: When a user runs another compile command, the newer command will have higher priorit, and Tinymist will "switch" the main file accordingly.
- Per-Document Flags: Some documents are compiled to HTML, while others are compiled to PDF. The users can specify more compile flags for each document.
- Session Persistence: Users can open the recently previewed file and continue editing it. More state such as the scroll position could be remembered in future.
- Sharing and VCS: The lock file can be shared with other users by tools like git, allowing them to compile the same project with the same settings.

35.2. Storing Compilation History

- **Storage in file system**: stored in `tinymist.lock` (TOML) files. When resolving a depended file, the nearest lock file will be used to determine the compilation arguments.
- **Storage in memory**: The language server also maintains a *Compilation History* and project routes in memory. We may enable in-memory *Compilation History* by default in the future, which will allow Tinymist to resolve projects smarter.

36. Project Route

The language server will load route entries from disk or memory, combine, and perform entry lookup based on the route table. Specificially, **The depended files** of a single compilation will be stored as route entries in the cache directory after compilation. A single route entry is a triple, (Dependent Path, Project ID, Priority), where:

- "Dependent Path" is an absoulte dependent file path, like paths to assets and source files in packages.
- "Project ID" is the project id (main file) indexing an entry in the *Compilation History* (`tinymist.lock`).
- "Priority" is a priority number.

And the language server determines a project id associating some dependent file by the following rules:

1. Highest priority routes take precedence.
2. Most recent updated projects in *Compilation History* prioritized automatically.

The cache directory contains cache of project routes. Currently, we haven't implemented a way clean up or garbage collect the project route cache, and disk cache may be deprecated in future. It is safe to remove all the project routes in the cache directory, as Tinymist will regenerate them when needed.

Language and Editor Features

Language service (LSP) features:

- Semantic highlighting
  - The “semantic highlighting” is supplementary to “syntax highlighting”.
- Code actions
  - Also known as “quick fixes” or “refactorings”.
- Formatting (Reformatting)
  - Provide the user with support for formatting whole documents, using typstfmt or typstyle.
- Document highlight
  - Highlight all break points in a loop context.
  - (Todo) Highlight all exit points in a function context.
  - (Todo) Highlight all captures in a closure context.
  - (Todo) Highlight all occurrences of a symbol in a document.
- Document links
  - Renders path or link references in the document, such as `image("path.png")` or `bibliography(style: "path.csl")`.
- Document symbols
  - Also known as “document outline” or “table of contents” *in Typst*.
- Folding ranges
  - You can collapse code/content blocks and headings.
- Goto definitions
  - Right-click on a symbol and select “Go to Definition”.
  - Or ctrl+click on a symbol.
- References
  - Right-click on a symbol and select “Go to References” or “Find References”.
  - Or ctrl+click on a symbol.
- Hover tips
  - Also known as “hovering tooltip”.
  - Render docs according to tidy style.
- Inlay hints
  - Inlay hints are special markers that appear in the editor and provide you with additional information about your code, like the names of the parameters that a called method expects.
- Color Provider
  - View all inlay colorful label for color literals in your document.
  - Change the color literal’s value by a color picker or its code presentation.
- Code Lens
  - Should give contextual buttons along with code. For example, a button for exporting your document to various formats at the start of the document.
- Rename symbols and embedded paths
- Help with function and method signatures
- Workspace Symbols
- Code Action
  - Increasing/Decreasing heading levels.
  - Turn equation into “inline”, “block” or “multiple-line block” styles.
- experimental/onEnter
  - `Enter` inside triple-slash comments automatically inserts `///`

  - `Enter` in the middle or after a trailing space in `//` inserts `//`
  - `Enter` inside `//!` doc comments automatically inserts `//!`
  - `Enter` inside equation markups automatically inserts indents.

Extra features:

- Compiles to PDF on save (configurable to as-you-type, or other options). Check Docs: Exporting Documents.
- Also compiles to SVG, PNG, HTML, Markdown, Text, and other formats by commands, vscode tasks, or code lenses.
- Provides test, benchmark, coverage collecting on documents and modules. Check Docs: Testing Features.
- Provides builtin linting. Check Docs: Linting Features.
- Provides a status bar item to show the current document's compilation status and words count.
- Editor tools:
  - View a list of templates in template gallery. (`tinymist.showTemplateGallery`)
  - Click a button in template gallery to initialize a new project with a template. (`tinymist.initTemplate` and `tinymist.initTemplateInPlace`)
  - Trace execution in current document (`tinymist.profileCurrentFile`).

Part 3. Service Overview

Overview of Service

This document gives an overview of tinymist service, which provides a single integrated language service for Typst. This document doesn't dive in details unless necessary.

1. Principles

Four principles are followed, as detailed in Principles.

- Multiple Actors
- Multi-level Analysis
- Optional Non-LSP Features
- Minimal Editor Frontends

2. Command System

The extra features are exposed via LSP's `workspace/executeCommand` request, forming a command system. They are detailed in Command System.

3. Additional Concepts for Typst Language

3.1. AST Matchers

Many analyzers don't check AST node relationships directly. The AST matchers provide some indirect structure for analyzers.

- Most code checks the syntax object matched by `get_deref_target` or `get_check_target`.
- The folding range analyzer and def-use analyzer check the source file on the structure named *lexical hierarchy*.
- The type checker checks constraint collected by a trivial node-to-type converter.

3.2. Type System

Check Type System for more details.

4. Notes on Implementing Language Features

Five basic analysis like *lexical hierarchy*, *def use info* and *type check info* are implemented first. And all rest Language features are implemented based on basic analysis. Check Analyses for more details.

Principles

Four principles are followed.

5. Multiple Actors

The main component, tinymist, starts as a thread or process, obeying the Language Server Protocol. tinymist will bootstrap multiple actors, each of which provides some typst feature.

The each actor holds and maintains some resources exclusively. For example, the compile server actor holds the well known `trait World` resource.

The actors communicate with each other by channels. An actor should own many receivers as its input, and many senders as output. The actor will take input from receivers *sequentially*. For example, when some LSP request or notification is coming as an LSP event, multiple actors serve the event collaboratively, as shown in Figure 3.

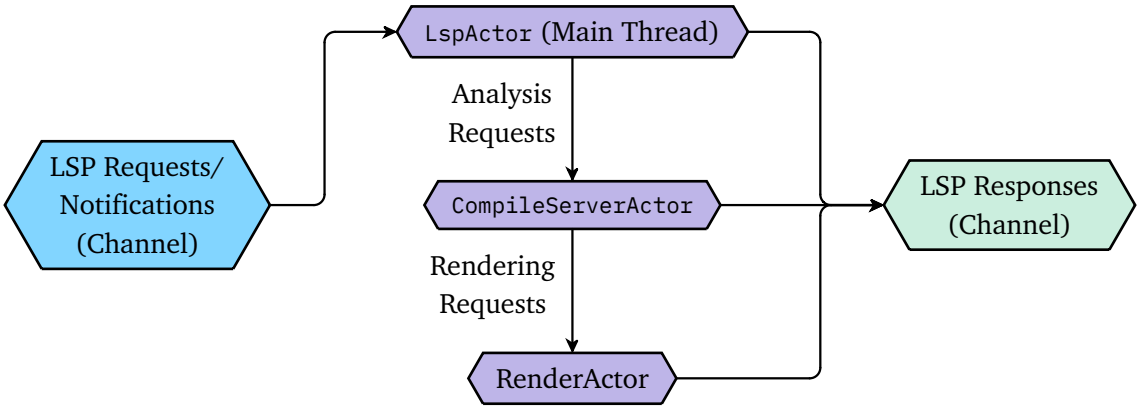

Figure 3: The IO Graph of actors serving a LSP request or notification

A *Hover* request is taken as example of that events.

A global unique `LspActor` takes the event and *mutates* a global server state by the event. If the event requires some additional code analysis, it is converted into an analysis request, `struct CompilerQueryRequest`, and pushed to the actors owning compiler resources. Otherwise, `LspActor` responds to the event directly. Obviously, the *Hover* on code request requires code analysis.

The `CompileServerActors` are created for workspaces and main entries (files/documents) in workspaces. When a compiler query is coming, a subset of that actors will take it and give project-specific responses, combining into a final concluded LSP response. Some analysis requests even require rendering features, and those requests will be pushed to the actors owning rendering resources. If you enable the periscope feature, a `Hover` on content request requires rendering on documents.

The `RenderActors` don't do compilations, but own project-specific rendering cache. They are designed for rendering documents in *low latency*. This is the last sink of `Hover` requests. A `RenderActor` will receive an additional compiled `Document` object, and render the compiled frames in needed. After finishing rendering, a response attached with the rendered picture is sent to the LSP response channel intermediately.

6. Multi-level Analysis

he most critical features are lsp functions, built on the tinymist-query crate. To achieve higher concurrency, functions are classified into different levels of analysis.

1. `query_source` – `SyntaxRequest` – locks and accesses a single source unit.
2. `query_world` – `SemanticRequest` – locks and accesses multiple source units.
3. `query_state` – `StatefulRequest` – acquires to accesses a specific version of compile results.

When an analysis request is coming, tinymist *upgrades* it to a suitable level as needed, as shown in Figure 4. A higher level requires to hold more resources and takes longer time to prepare.

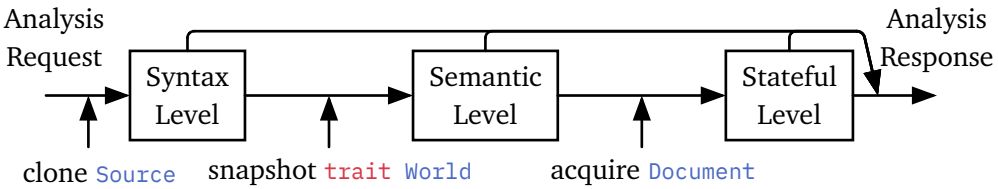

Figure 4: The analyzer upgrades the level to acquire necessary resources

7. Optional Non-LSP Features

All non-LSP features in tinymist are **optional**. They are optional, as they can be disabled **totally** on compiling the tinymist binary. The significant features are enabled by default, but you can disable them with feature flags. For example, `tinymist` provides preview server features powered by `typst-preview`.

8. Minimal Editor Frontends

Leveraging the interface of LSP, tinymist provides frontends to each editor, located in the editor folders. They are minimal, meaning that LSP should finish its main LSP features as many as possible without help of editor frontends. The editor frontends just enhances your code experience. For example, the vscode frontend takes responsibility on providing some nice editor tools. It is recommended to install these editors frontend for your editors.

Command System

The extra features are exposed via LSP's `workspace/executeCommand` request, forming a command system. The commands in the system share a name convention.

- `exportFmt`. these commands perform export on some document, with a specific format (`Fmt`), e.g. `exportPdf`.
- `interactCodeContext({kind}[])`. The code context requests are useful for *Editor Frontends* to extend some semantic actions. A batch of requests are sent at the same time, to get code context *atomically*.
- `getResources("path/to/resource/", opts)`. The resources required by *Editor Frontends* should be arranged in paths. A second arguments can be passed as options to request a resource. This resemebles a restful `POST` action to LSP, with a url path and a HTTP body, or a RPC with a method name and params.

  Note you can also hide some commands in list of commands in UI by putting them in `getResources` command.
- `doXxx`. these commands are internally for *Editor Frontends*, and you'd better not to invoke them directly. You can still invoke them manually, as long as you know what would happen.
- The rest commands are public and tend to be user-friendly.

8.1. Code Context

The code context requests are useful for *Editor Frontends* to check syntax and semantic the multiple positions. For example an editor frontend can filter some completion list by acquire the code context at current position.

LSP Inputs

9. Prefer to Using LSP Configurations

Though tinymist doesn't refuse to keep state in your disk, it actually doesn't have any data to write to disk yet. All customized behaviors (user settings) are passed to the server by LSP configurations. This is a good practice to keep the server state clean and simple.

10. Handling Compiler Input Events

The compilation triggers many side effects, but the behavior of compiler actor is still easy to predicate. This is achieved by accepting all compile inputs by events.

Let us take reading files from physical file system as example of processing compile inputs, as shown in Figure 5. The upper access models take precedence over the lower access models. The memory access model is updated *sequentially* by `LspActor` receiving source change notifications, assigned with logical ticks $t_{L,n}$. The notify access model is also updated in same way by `NotifyActor`. When there is an absent access, the system access model initiates the request for the file system directly. The read contents from fs are assigned with logical access time $t_{M,n}$.

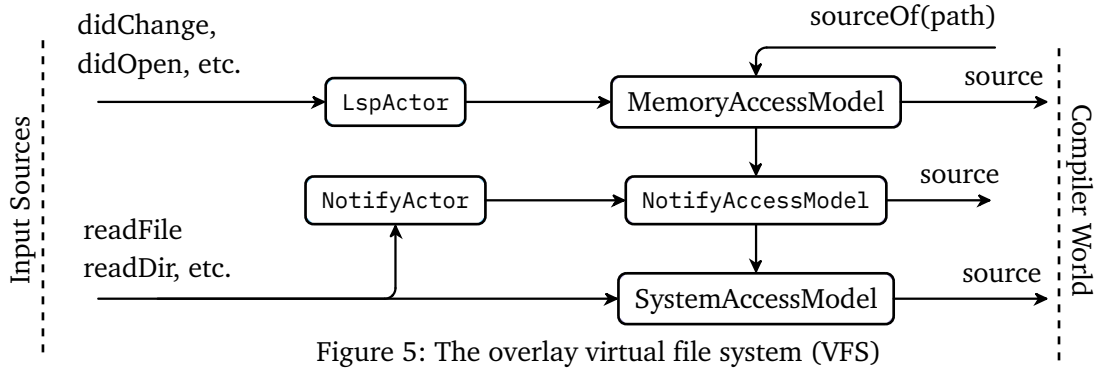

Figure 5: The overlay virtual file system (VFS)

The problem is to ensure that the compiler can read the content correctly from access models at the time.

If there is an only active input source in a *small time window*, we can know the problem is solved, as the logical ticks $t_{L,n}$ and $t_{M,n}$ keep increasing, enforced by actors. For example, if there is only `LspActor` active at the *small time window*, the memory access model receives the source changes in the order of $t_{L,n}$, i.e. the *sequential* order of receiving notifications. The cases of two rest access models is more complicated, but are also ensured that compiler reads content in order of $t_{M,n}$.

Otherwise, the two input sources are both active in a *small time window* on a **same file**. However, this indicates that, the file is in already the memory access model at most time. Since the precedence, the compiler reads content in order of $t_{L,n}$ at the time.

The only bad case can happen is that: When the two input sources are both active in a *small time window* δ on a **same file**:

- first `LspActor` removes the file from the memory access model, then compiler doesn't read content from file system in time δ.
- first `NotifyActor` inserts the file from the inotify thread, then the LSP client (editor) overlays an older content in time δ.

This is handled by tinymist by some tricks.

10.1. Record and Replay

Tinymist can record these input events with assigned the logic ticks. By replaying the events, tinymist can reproduce the server state for debugging. This technique is learned from the well-known LSP, clangd, and the well known emulator, QEMU. This concrete usage is documented in Language Server: Debugging with input mirroring.

Type System

The underlying techniques are not easy to understand, but there are some links:

- bidirectional type checking: https://jaked.org/blog/2021-09-15-Reconstructing-TypeScript-part-1
- type system borrowed here: https://github.com/hkust-taco/mlscript

Some tricks are taken for help reducing the complexity of code:

First, the array literals are identified as tuple type, that each cell of the array has type individually.

Second, the sig and the argument type are reused frequently.

- the tup type is notated as $(\tau_1, .., \tau_n)$, and the arr type is a special tuple type $\mathsf{arr} ::= \mathsf{arr}(\tau)$.
- the rec type is imported from mlscript, notated as $\{a_1 = \tau_1, .., a_n = \tau_n\}$.
- the sig type consists of:
  - a positional argument list, in tup type.
  - a named argument list, in rec type.
  - an optional rest argument, in arr type.
  - an **optional** body, in any type.

  notated as $\mathsf{sig} := \mathsf{sig}(\mathsf{tup}(\tau_1, .., \tau_n), \mathsf{rec}(a_1 = \tau_{n+1}, .., a_m = \tau_{n+m}), ..\mathsf{arr}; (\tau_{n+m+1})) \to \psi$
- the argument is a signature without rest and body.

  $\mathsf{args} := \mathsf{args}(\mathsf{sig}(..))$

With aboving constructors, we soonly get typst's type checker.

- it checks array or dictionary literals by converting them with a corresponding sig and args.
- it performs the getting element operation by calls a corresponding sig.
- the closure is converted into a typed lambda, in sig type.
- the pattern destructing are converted to array and dictionary constrains.

Part 4. Service Development

Crate Docs

Tinymist Crate Docs (for Developers)

Language Server

1. Architecture

Tinymist binary has multiple modes, and it may runs multiple actors in background. The actors could run as an async task, in a single thread, or in an isolated process.

The main process of tinymist runs the program as a language server, through stdin and stdout. A main process will fork:

- rendering actors to provide PDF export with watching.
- preview actors that give a document/outline preview over some typst source file.
- compiler actors to provide language APIs.

From the directory structure of `crates/tinymist`, the `main.rs` file parses the command line arguments and starts commands:

```
match args.command.unwrap_or_default() {
    Commands::Query(query_cmds) => query_main(query_cmds),
    Commands::Lsp(args) => lsp_main(args),
    Commands::TraceLsp(args) => trace_lsp_main(args),
    Commands::Preview(args) => tokio_runtime.block_on(preview_main(args)),
    Commands::Probe => Ok(()),
}
```

The `query` subcommand contains the query commands, which are used to perform language queries via cli, which is convenient for debugging and profiling single query of the language server.

There are three servers in the `server` directory:

- `lsp` provides the language server, initialized by `lsp_main` in `main.rs`.
- `trace` provides the trace server (profiling typst programs), initialized by `trace_lsp_main` in `main.rs`.
- `preview` provides a `typst-preview` compatible preview server, initialized by `preview_main` in `tool/preview.rs`.

The long-running servers are contributed by the `ServerState` in the `server.rs` file.

They will bootstrap actors in the `actor` directory and start tasks in the `task` directory.

They can construct and return resources in the `resource` directory.

They may invoke tools in the `tool` directory.

2. Debugging with input mirroring

You can record the input during running the editors with Tinymist. You can then replay the input to debug the language server.

```
# Record the input
tinymist lsp --mirror input.txt
# Replay the input
tinymist lsp --replay input.txt
```

3. Analyze memory usage with DHAT

You can build the program with `dhat-heap` feature to collect memory usage with DHAT. The DHAT will instrument the allocator dynamically, so it will slow down the program significantly.

```
cargo build --release --bin tinymist --features dhat-heap
```

The instrumented program is nothing different from the normal program, so you can mine the specific memory usage with a lsp session (recorded with `--mirror`) by replaying the input.

```
./target/release/tinymist lsp --replay input.txt
...
dhat: Total:     740,668,176 bytes in 1,646,987 blocks
dhat: At t-gmax: 264,604,009 bytes in 317,241 blocks
dhat: At t-end:  259,597,420 bytes in 313,588 blocks
dhat: The data has been saved to dhat-heap.json, and is viewable with dhat/dh_view.html
```

Once you have the `dhat-heap.json`, you can visualize the memory usage with the DHAT viewer.

4. Server-Level Profiling

In VS Code, you can get the profiling data of the language server by searching and running the "Typst: Profile server" command.

> To use this feature in other LSP clients, please check `/editors/vscode/src/features/tool.ts`. A client should start and then stop profiling to collect performance events inside the time window.

5. Contributing

See CONTRIBUTING.md.

Language Queries

6. Base Analyses

There are seven basic analyzers:

- *lexical hierarchy* matches crucial lexical structures in the source file.
- *expression info* is computed incrementally on source files.
- *type check info* is computed with *expression info*.
- *definition finder* finds the definition based on *expression info* and *type check info*.
- *references finder* finds the references based on *definitions* and *expression info*.
- *signature resolver* summarizes signature based on *definitions* and *type check info*.
- *call resolver* check calls based on *signatures* and *type check info*.

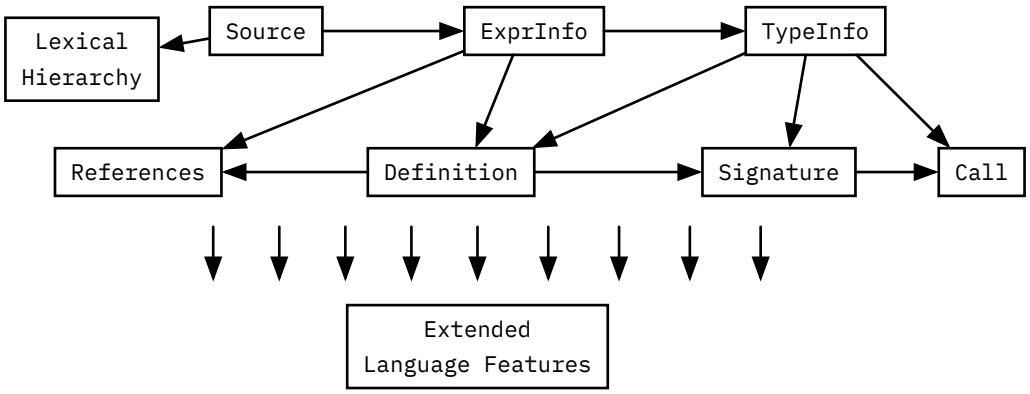

Figure 6: The relationship of analyzers.

7. Extending Language Features

Typicial language features are implemented based on basic analyzers:

- The `textDocument/documentSymbol` returns a tree of nodes converted from the *lexical hierarchy*.
- The `textDocument/foldingRange` also returns a tree of nodes converted from the *lexical hierarchy* but with a different approach.
- The `workspace/symbol` returns an array of nodes converted from all *lexical hierarchy*s in workspace.
- The `textDocument/definition` returns the result of *find definition*.
- The `textDocument/completion` returns a list of types of *related* nodes according to *type check info*, matched by *AST matchers*.
- The `textDocument/hover` *finds definition* and prints the definition with a checked type by *type check info*. Or, specific to typst, prints a set of inspected values during execution of the document.
- The `textDocument/signatureHelp` also *finds definition* and prints the signature with union of inferred signatures by *type check info*.
- The `textDocument/prepareRename` *finds definition* and determines whether it can be renamed.
- The `textDocument/rename` *finds definition and references* and renamed them all.

8. Contributing

See CONTRIBUTING.md.

Preview

Todo: rewrite this guide Developer Guide: Typst-Preview Architecture.

9. Contributing

See CONTRIBUTING.md.