Tail: A Typed and Structured Document Editor

ALPEREN KELES, University of Maryland, College Park, USA

1 INTRODUCTION

Today, the vast literature of programming language theory is complemented by a large body of work on text editors for programming languages such as pretty printing, linting, or structured editing. Yet, both document languages and document editors are understudied in the literature, meaning that the rich landscape of document languages and editors lack a theoretical framework to construct and evaluate them, as well as understand their semantics. Recent work on a core document calculus by Crichton and Krishnamurthi [1] motivates a new line of research on document languages by pointing out their ubiquity, demonstrating the connections between document languages and programming languages, and providing a formalization of the document language semantics. In this research, inspired by works such as Hazel [3] on program editors, we propose Tail, a typed and structured document editor and language for constructing and manipulating document templates. This extended abstract introduces a model of templates as partial documents, a refinement relation on partial documents, and builds up the Tail editor on top of the provided refinement relation. The paper is concluded with the current state of the project followed by a brief discussion of future directions.

2 LANDSCAPE

The age of modern computing is rich in document editors and languages. On one hand, we have document languages that host Turing-complete programming languages within them such as TEX or Typst; on the other hand, we have WYSIWYG editors like Microsoft Word, Google Docs, Notion, or Obsidian that provide entirely different experiences to their users. Additionally, we also have special-purpose document editors such as Resume Builders or Contract Generators that hide the complexity of their document-building process by presenting the user with a set of holes in a document via forms and switches, instead of the full control general-purpose systems provide.

With the existence of such different options, one question naturally arises. What do these different options enable for their users? The coarse categorization in the previous paragraph does not allow us to fully construct a taxonomy of document editors, but we will provide a set of features document editors tend to facilitate. Below is a non-exhaustive list of features of document systems.

(1) **Text:** Font, alignment, color, style, spacing

(2) **Figures:** Charts, tables, images

(3) **Interactive Elements:** Check-boxes, toggles

(4) **Layout:** Margin, column layout

(5) **References:** External (people, database element) or internal (sections, figures)

(6) **Re-usability/Programmability:** Commands, variables

Author's address: Alperen Keles, Computer Science Department, University of Maryland, College Park, College Park, Maryland, USA, akeles@umd.edu.

One feature that the current document systems lack is the ability to facilitate templating. Here, templating refers to the capability that a user may create a template that will act as a blueprint for different users to instantiate with their data. The programmability features are useful for templating, yet their implementations are either ad hoc (as in the case of Google Docs, where one can create variables but cannot compose several variables into creating a template), or require too much manual effort such as in the case TEXor Typst.

3 TAIL

The design of Tail is focused on accommodating templating and therefore filling in the aforementioned gap in the document system space. A Tail document is a partial document with holes, and Tail editor allows for refining partial documents by progressively filling holes. In this section, we express the semantics for the refinement relation and describe Tail editor as a structured editor for applying semantically valid refinements on partial documents.

Below is the definition of *Template*. Type constructors are **written in bold**.

Template t ::= Literal | Stack | Row | Reference | Hole | Plural | Fill | **Empty**
Literal l ::= String
Reference r ::= Id
Hole h ::= **hole** Vertical | **hole** Horizontal
Stack s ::= **stack** [Template]
Row r ::= **row** [Template]
Plural p ::= **plural** Template Template
Fill f ::= **fill** [Template] Template

For two given Templates, T_1 and T_2, we say T_1 **is a refinement of** T_2 when there exists a set of steps for refining T_2 into T_1. Below are the single step relation rules for refinement. $\mapsto$ denotes a refinement relation, *hasHole* is a proposition denoting if the Template t_1 has any holes to be filled, and *fill* is a function that apply a refinement rule on the hole denoted by *hasHole*.

$$\frac{}{t \mapsto t}\ \text{(Reflexivity)}$$

$$\frac{t_1 \mapsto t_2 \qquad t_2 \mapsto t_3}{t_1 \mapsto t_3}\ \text{(Transitivity)}$$

$$\frac{}{\text{Reference } v \mapsto \textbf{Empty}}\ \text{(Missing Variable)}$$

$$\frac{}{\text{Reference } r \mapsto \text{Literal } l}\ \text{(Variable Substitution)}$$

$$\frac{}{\text{Hole } h \mapsto \textbf{Empty}}\ \text{(Unfilled Hole)}$$

$$\frac{}{\textbf{hole Vertical} \mapsto \textbf{stack [Template } t\textbf{, hole Vertical]}}\ \text{(Vertical Template Fill)}$$

$$\frac{}{\textbf{hole Horizontal} \mapsto \textbf{row [Template } t\textbf{, hole Horizontal]}}\ \text{(Horizontal Template Fill)}$$

$$\frac{t_1 \mapsto t_3}{\textbf{plural } t_1\ t_2 \mapsto \textbf{plural } t_3\ t_2}\ \text{(Plural Source Refinement)}$$

$$\frac{t_2 \mapsto t_3}{\textbf{plural } t_1\ t_2 \mapsto \textbf{plural } t_1\ t_3}\ \text{(Plural Destination Refinement)}$$

$$\frac{\bigwedge_{i=1}^{n}(t \mapsto t_i)}{\textbf{plural}\ t\ t' \mapsto \textbf{fill}\ [t_1, t_2 ... t_n]\ t'}\ \text{(Plural Fill)}$$

$$\frac{hasHole(t_i) \qquad t' = fill(t_i, t)}{\textbf{stack}\ [t_1 ... t_i ... t_n] \mapsto \textbf{stack}\ [t_1 ... t' ... t_n]}\ \text{(Stack Hole Fill)}$$

$$\frac{hasHole(t_i) \qquad t' = fill(t_i, t)}{\textbf{row}\ [t_1 ... t_i ... t_n] \mapsto \textbf{row}\ [t_1 ... t' ... t_n]}\ \text{(Row Hole Fill)}$$

The refinement relation defined with respect to the rules above is still under active development. The relation acts as a basis for the potential user actions in our editor. For a more formal definition, readers are encouraged to read the short inductive definition of the refinement relation in the Coq theorem prover in Appendix A.

3.1 Rendering Documents

So far, we have talked about partial documents, and how Tail is a useful tool for creating and manipulating partial documents. Yet, to be useful, Tail also needs to be able to render these documents into readable forms. Although we do not currently have formal semantics for the rendering process, we have two implementations, one in Rust, one in TypeScript, that can render documents with realistic document features such as custom fonts, alignments, and styles.

3.2 Future Work: Document Editing with Live GUI's

In PLDI'21, Omar et al. [2] has introduced Livelit's into Hazel [3]. A livelit allows clients to manipulate programs using a live GUI instead of by code. We propose a similar mechanism with respect to our variable declarations. A *Reference* in Tail is a variable defined within a template to be filled by the user. References can have types, where examples of such types can include a string, date, number, location, list. In the future, we plan to add (1) a type system for being able to define custom types, (2) a mechanism for implementing customized editors for user-defined types. Livelits will be useful for modelling features such as charts or tables that many editors implement using ad hoc mechanisms, and their extensibility will give template designers to simulate creating their editors.

4 CONCLUSION

Tail is a project born of a practical need for document templating, built on top of a simple non-overlapping document model and relies on an abstract refinement relation for partial documents for correctness and generality. Whereas programming language community has worked on understanding the trade-offs of different designs of programming languages and editors, developed tools for analyzing and optimizing programs, our understanding of the document systems of today are archaic with respect to the level of understanding we have for programming systems. This study, following the path of others working on understanding document systems, is a step in following through this long needed path.

REFERENCES

[1] Will Crichton and Shriram Krishnamurthi. 2023. A Core Calculus for Documents. arXiv:cs.PL/2310.04368

[2] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021).* Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

[3] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. arXiv:cs.PL/1703.08694

APPENDIX A

```
Require Import String List.
Inductive Orientation := Horizontal | Vertical.
Inductive Template :=
| Literal: string → Template
| Stack: list Template → Template
| Row: list Template → Template
| Reference: string → Template
| Hole: Orientation → Template
| Plural: Template → Template → Template
| Fill: list Template → Template → Template
| Empty: Template.

Axiom fill : Template → Template → Template.

Inductive hasHole : Template → Prop :=
| hasHole_stack: forall ts, AllHasHole ts → hasHole (Stack ts)
| hasHole_row: forall ts, AllHasHole ts → hasHole (Row ts)
| hasHole_hole: forall o, hasHole (Hole o)
with AllHasHole : list Template → Prop :=
| AllHasHole_nil: AllHasHole nil
| AllHasHole_cons: forall t ts, hasHole t → AllHasHole ts → AllHasHole (t :: ts).

Inductive Refinement: Template → Template → Prop :=
| refine_id: forall t, Refinement t t
| refine_transitive: forall t1 t2 t3, Refinement t1 t2 → Refinement t2 t3 → Refinement t1 t3
| refine_missing_var : forall s, Refinement (Reference s) Empty
| refine_variable_subst: forall v s, Refinement (Reference v) (Literal s)
| refine_unfilled_hole : forall o, Refinement (Hole o) Empty
| refine_vertical_template_fill: forall t, Refinement (Hole Vertical)
    (Stack (t :: (Hole Vertical) :: nil))
| refine_horizontal_template_fill: forall t, Refinement (Hole Horizontal)
    (Row (t :: (Hole Horizontal) :: nil))
| refine_plural_source: forall t1 t2 t3, Refinement t1 t3 →
    Refinement (Plural t1 t2) (Plural t3 t2)
| refine_plural_destination: forall t1 t2 t3, Refinement t2 t3 →
    Refinement (Plural t1 t2) (Plural t1 t3)
| refine_plural_fill: forall t1 t2 ts, AllRefines ts t1 → Refinement (Plural t1 t2) (Fill ts t2)
| refine_stack_fill: forall t1 t2 ts ts', hasHole t1 →
    Refinement (Stack (ts ++(t1 :: nil) ++ts'))
    (Stack (ts ++(fill t1 t2 :: nil) ++ts'))
| refine_row_fill: forall t1 t2, hasHole t1 → Refinement (Row (t1 :: nil))
    (Row (fill t1 t2 :: nil))
with AllRefines : list Template → Template → Prop :=
| refine_nil : forall t, AllRefines nil t
| refine_cons : forall t1 ts t2, AllRefines ts t2 → Refinement t1 t2 → AllRefines (t1 :: ts) t2.
```