

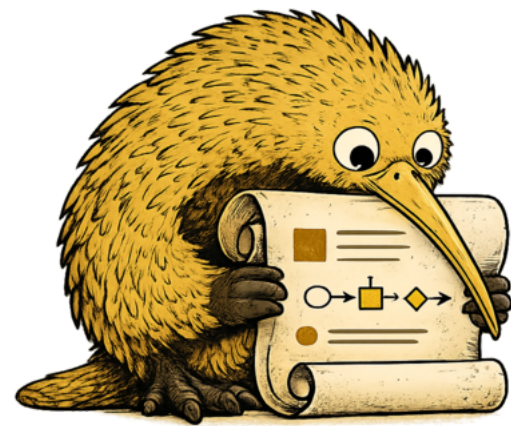
ss: A Slide Description Language with Dependence-Scheduled Execution

Yuchi Yamaguchi
(Institute of Science Tokyo)



What is ss?

- ss is a domain-specific language for **slide authoring**
- Our goal is to create a **practical slide authoring tool** for programmers who love abstraction ability



Of course, this slide is
created by ss!

Example of ss

```
1 import std:themes/default as *
2 import tablelib as tl
3
4 fn/! section_title(title: String) -> Object
5   return head("Fibonacci: " ++ title)
6 end
7
8 const sample_program: String =
9   "fib(n) = n < 2 ? n : fib(n-1) + fib(n-2)"
10
11 page toc
12   text!(join(doc_objs(docctx()), "title",
13             " / ", (title: Object) |-> content(title)))
14 end
15
16
17 page algo
18   section_title!("Implementations")
19   text!(sample_program)
20 end
21
22 page result
23   section_title!("Benchmarks")
24   text!(sample_program)
25   tl.show_table!("bench_result.csv")
26 end
```

Annotations:

- Module System (points to line 1)
- Function (points to line 4)
- Query Component (points to line 12)
- Reuse Component (points to line 18)
- Reuse Shared Value (points to line 23)
- Data-Driven Content (points to line 25)

👍 Abstraction enables

- **Reusability** and
- **Modularity**

in slide authoring.

Key Features of S S

- 1. Dependence-Scheduled Execution Model**
- 2. WYSIWYG Integration**

1. Dependence-Scheduled Execution Model

Dependencies Across Slides

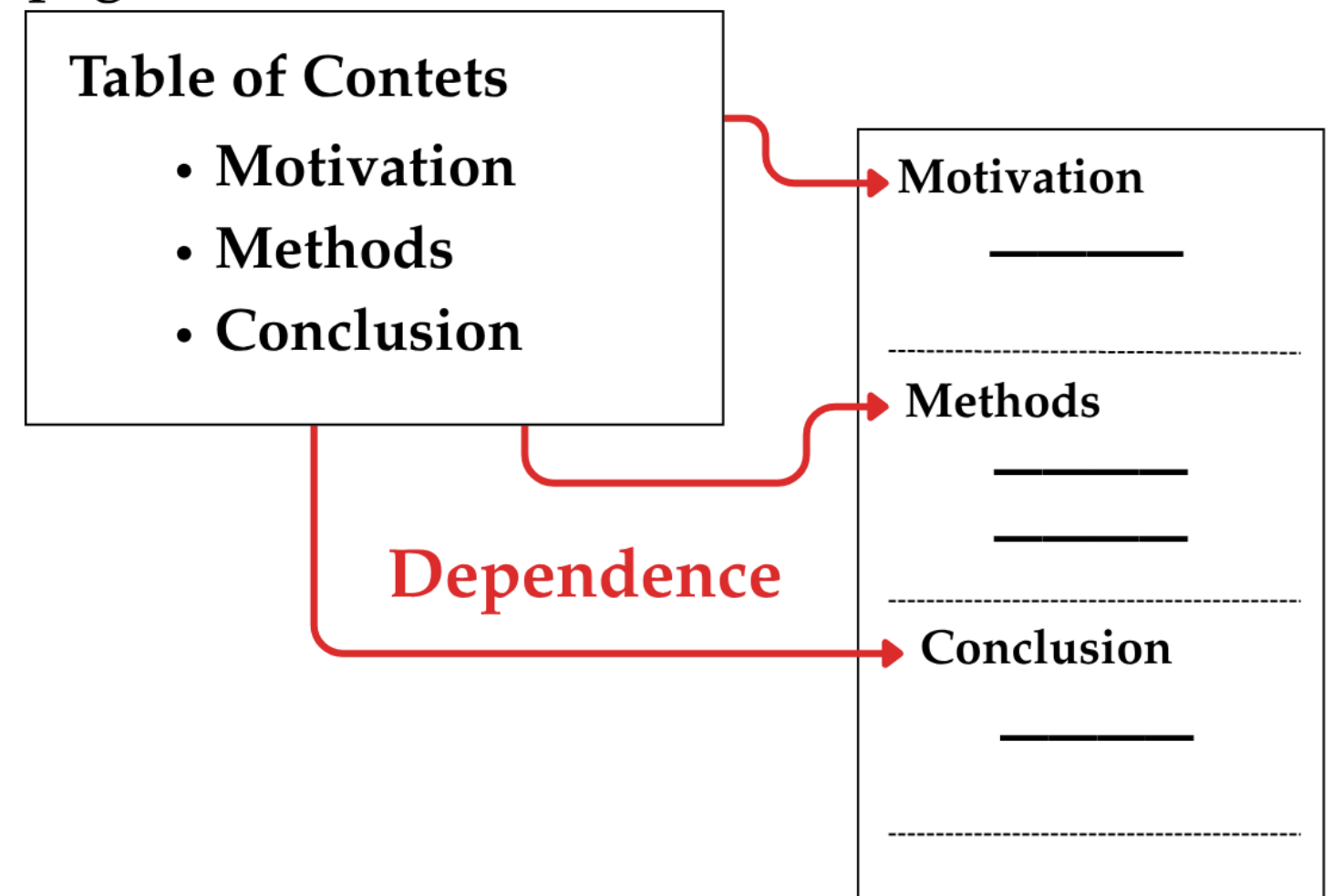
In slides,

Some components depend on objects across the slides

e.g.,

- Table of Contents
 - depends on all title-like objects
- References
 - depend on all citation and reference objects ...

page 1



Q. How can we create such a component correctly?

A. Manage dependencies carefully yourself.

```
1 #lang slideshow
2 (require racket/string)
3
4 (define titles '())
5 (define (section title)
6   (set! titles (append titles (list title))))
7   (slide #:title title (t title)))
8
9 (slide #:title "Contents"
10   (t (string-join titles " / ")))
11 (section "Algorithm")
12 (section "Results")
```

 **Missing
Dependence!**

Our Approach: Dependence-Scheduled Execution Model

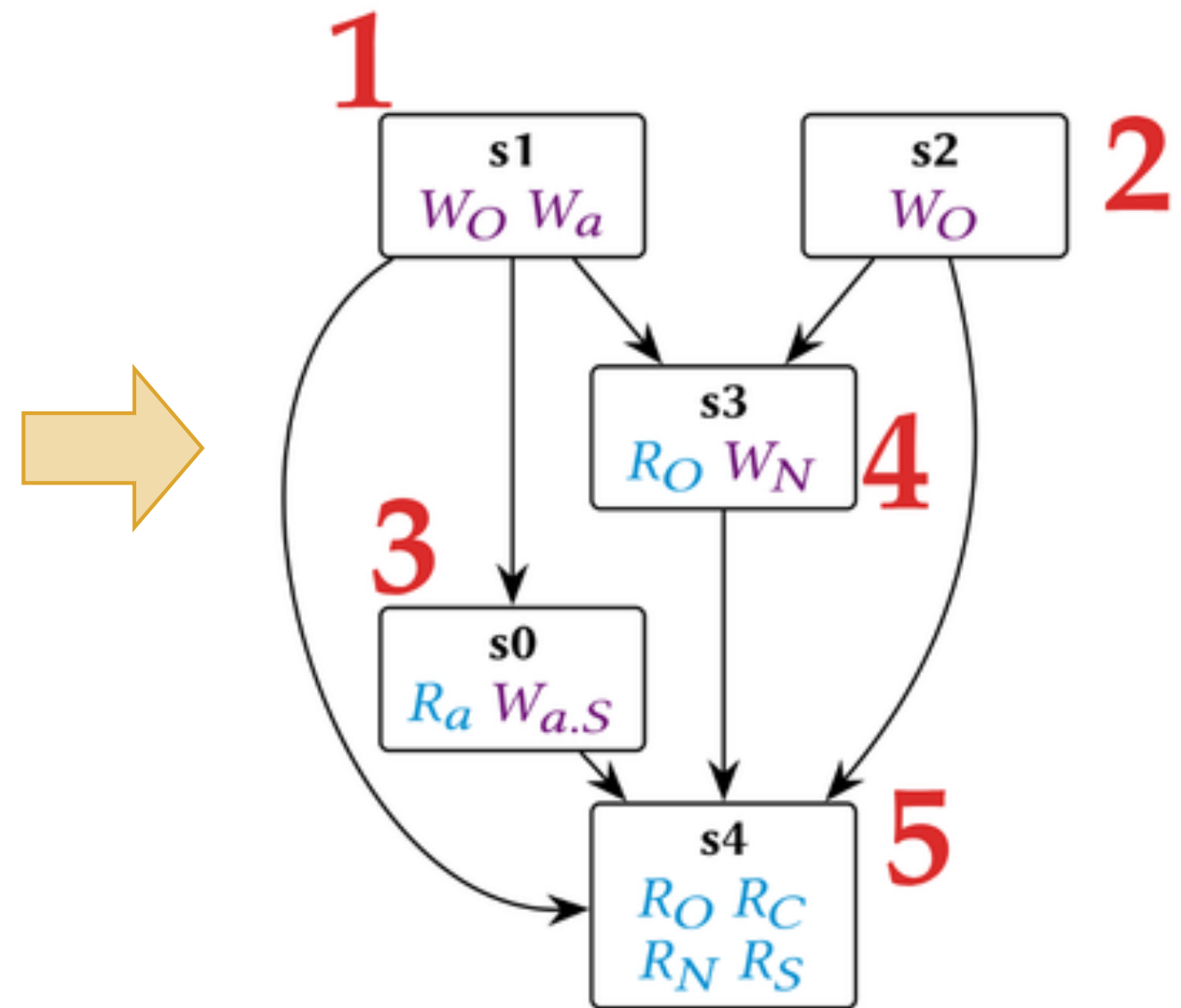
1. The compiler infers the objects read and written by each statement.

```
1 let titles = doc_objs(docctx(), "title")
2 ;; ^dep?
3 ;; DependencyQuery: DependencyQuery:
4 ;;   read Objects(role=title)
5 ;;   write Variable(scope=page:title, name=titles)
6
7 head!( "Introduction" )
8 ;; ^dep?
9 ;; DependencyQuery: DependencyQuery:
10 ;;   write Objects(role=title)
11 ;;   flag places_objects
12 ;;   ...
```

Our Approach: Dependence-Scheduled Execution Model

2. Construct the dependence graph and schedule statements based on it.

```
1 fn num_fig(item:Object, i:Number) -> Void
2   item.number_prefix = str(i) ++ ":"
3 end
4
5 document
6   foreach_enumerate(
7     doc_objs(docctx(), "fig"),
8     num_fig)          ;; s3
9 end
10
11 page summary
12   text!(join(doc_objs(docctx(), "fig"),
13             ", ", fig_repr)) ;; s4
14 end
15
16 page one
17   let a = fig!("A")      ;; s1
18   a.suffix = "important!" ;; s0
19 end
20
21 page two
22   fig!("B")              ;; s2
23 end
```



Example: Revisited

Order:

3

```
1 page toc
2   text!(join(doc_objs(docctx()), "title"),
3             " / ", (title: Object) |-> content(title)))
4   ;; ^dep?
5   ;;   DependencyQuery: DependencyQuery:
6   ;;   read Objects(role=title)
7   ;;   read Property(owner=object:Title.*, key=content)
8   ;;   ...
9 end
```

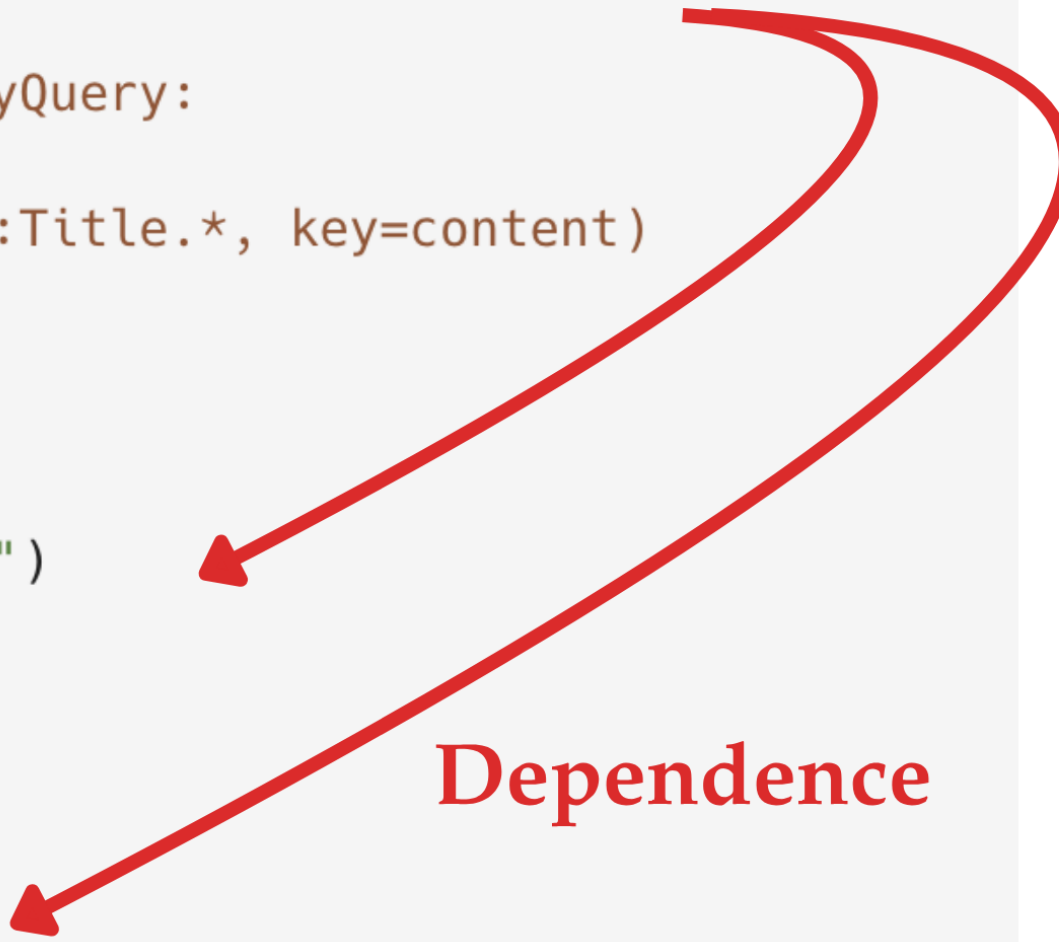
1

```
10
11 page algo
12   section_title!("Implementations")
13   text!(sample_program)
14 end
```

2

```
15
16 page result
17   section_title!("Benchmarks")
18   text!(sample_program)
19   tl.show_table!("bench_result.csv")
20 end
```

Dependence



User-Defined, Declarative, No Repeated Passes document-wide components through Dependence-Scheduled Execution.

Key Features of S S

- 1. Dependence-Scheduled Execution Model**
- 2. WYSIWYG Integration**

2. WYSIWYG Integration

Positioning Slide Objects using Constraints

- In `ss`, users position objects using constraints between objects
- This way, most layouts can be expressed easily.

```
let img = image!("dog.png")  
let caption = text!("My Cute Dog!")
```

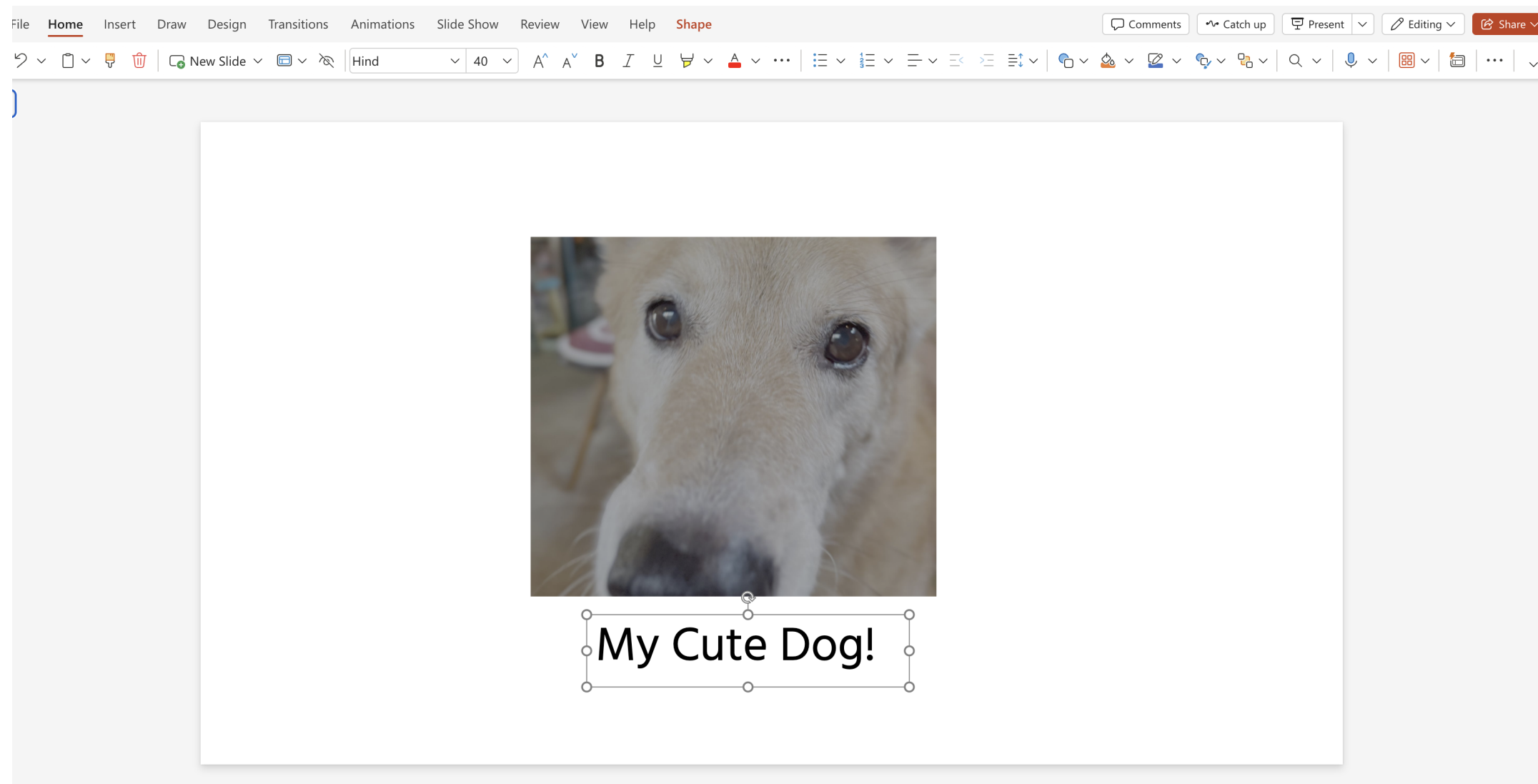
```
~ caption.top == img.bottom - 10  
~ caption.center_x == img.center_x
```



My Cute Dog!

WYSIWYG Editor for Layout Adjustment

- WYSIWYG tools like **PowerPoint** and **Keynote** allow users to make adjustments easily.



Our Approach: WYSIWYG Integration

ss provides **WYSIWYG Integration!**

⇒ **Live Demo**

(This page is for a demo)



My Cute Dog!

- Users can create most layouts easily
- Users also can adjust layouts precisely
- And source code is kept as the **single source of truth (SSoT)**.
 - ⇒ It's **version control friendly!**

Implementation & Performance

- We implemented a compiler for ss in Zig. (github.com/abap34/ss)
- We evaluated its performance on a 57-page slide deck.
- **Its performance is sufficient for practical use.**

Phase (ms)	Initial (1 job)	Initial (8 jobs)	Incremental (Text Edit)	Incremental (Position Edit)
Analyze	40	40	33	—
Evaluate document	44	43	26	—
Prepare pages	5	5	5	—
Solve layouts	633	550	24	2
Build Render IR	153	166	10	—
Generate / Update HTML	153	154	5	8
Total	1,049	978	148	15

Why does s s Achieve Practical Performance?

- **Parallelism & Incremental Processing:** After solving layouts, pages are independent and can be processed in parallel and incrementally.
- **No Repeated Passes:** Document-wide components are resolved without repeated passes, unlike **LaTeX or Typst**.

Future Work

- `ss` is a **new standalone** language!
 - This means we are free to redesign the language itself, unlike embedded DSLs (such as **slideshow**).
 - I plan to introduce new syntax so that most slides can be written easily.
- We also plan to build a central package registry for greater reusability.



abap34/ss

Backup

Performance for PDF

Phase (ms)	Initial (1 job)	Initial (8 jobs)
Analyze	41	40
Evaluate document	44	43
Prepare pages	5	5
Solve layouts	634	519
Build Render IR	154	162
Render PDF	1,440	470
Total	2,349	1,278

Construction of Dependency Graph

Create edges between statements s_i and s_j iff

$$\exists w_i \in \hat{W}_i, r_j \in \hat{R}_j, w_i \rightsquigarrow r_j$$

($\hat{W}_i$ is the set of written resources of statement s_i , and $\hat{R}_j$ is the set of read resources of statement s_j , $\rightsquigarrow$ is pre-defined relation between resources that indicates a dependency.)