A Delta for Hybrid Type Checking

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

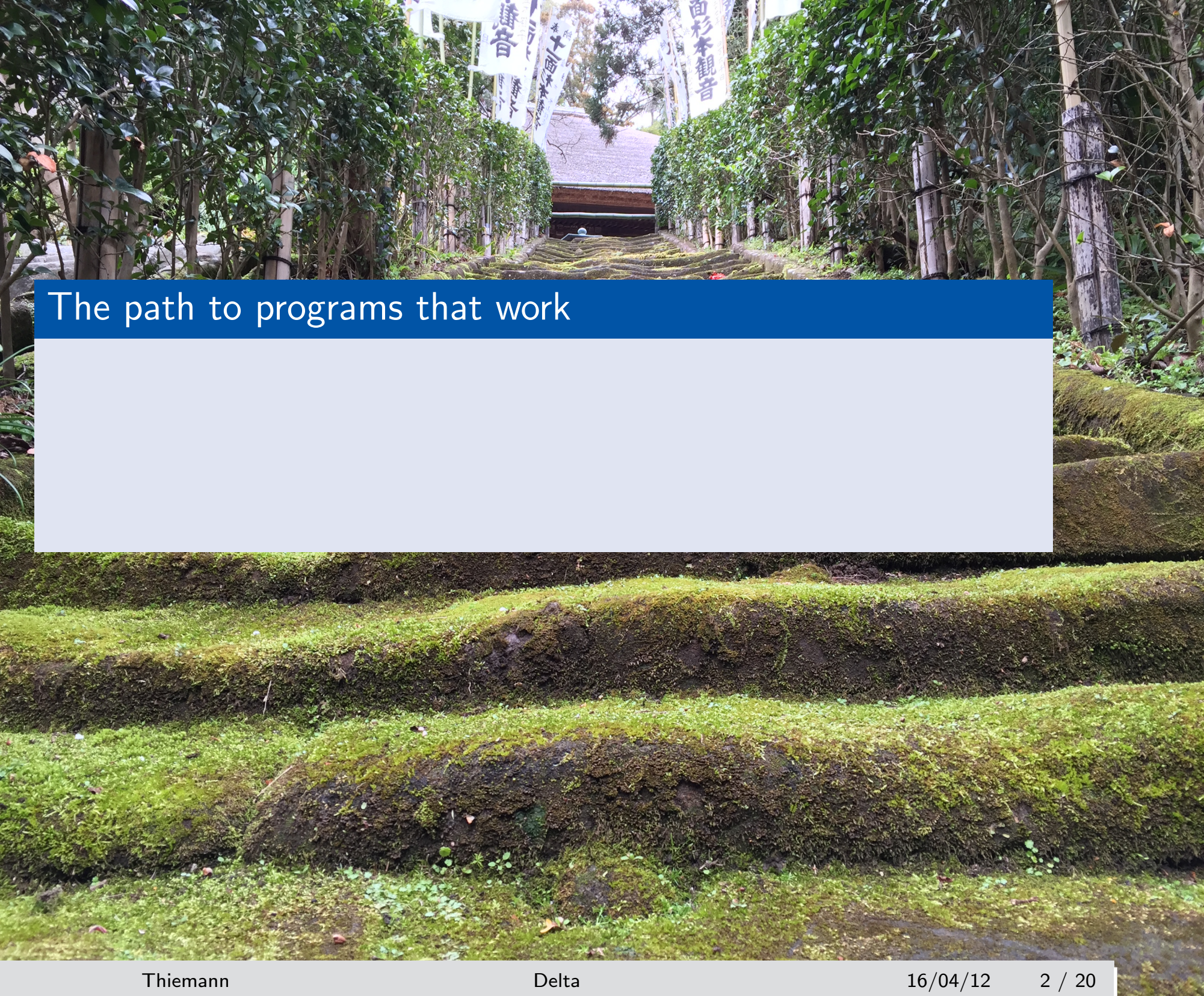
Peter Thiemann

University of Freiburg, Germany

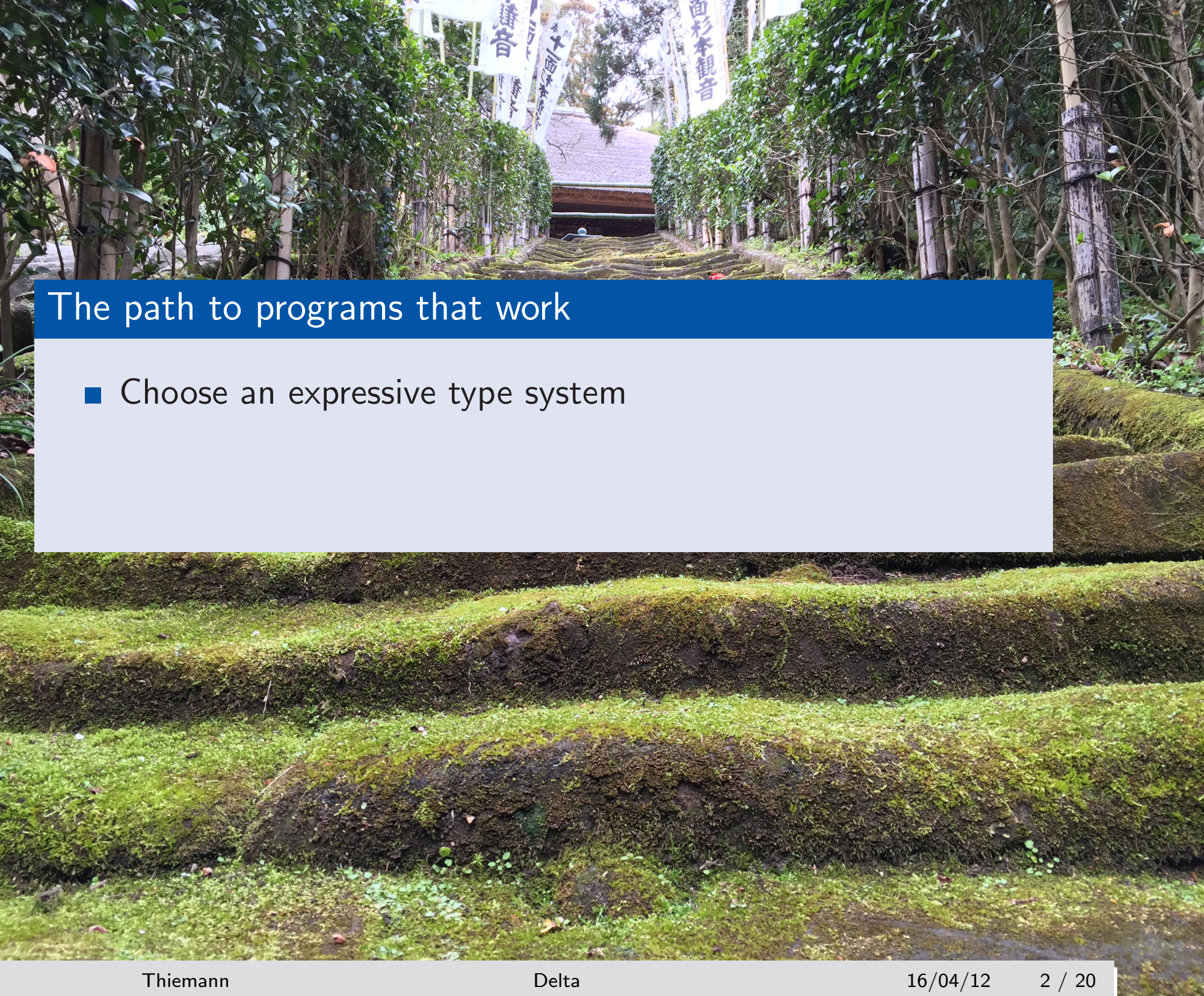
thiemann@informatik.uni-freiburg.de

12 April 2016


十一面観音
三十三番札所
十一面杉本観音

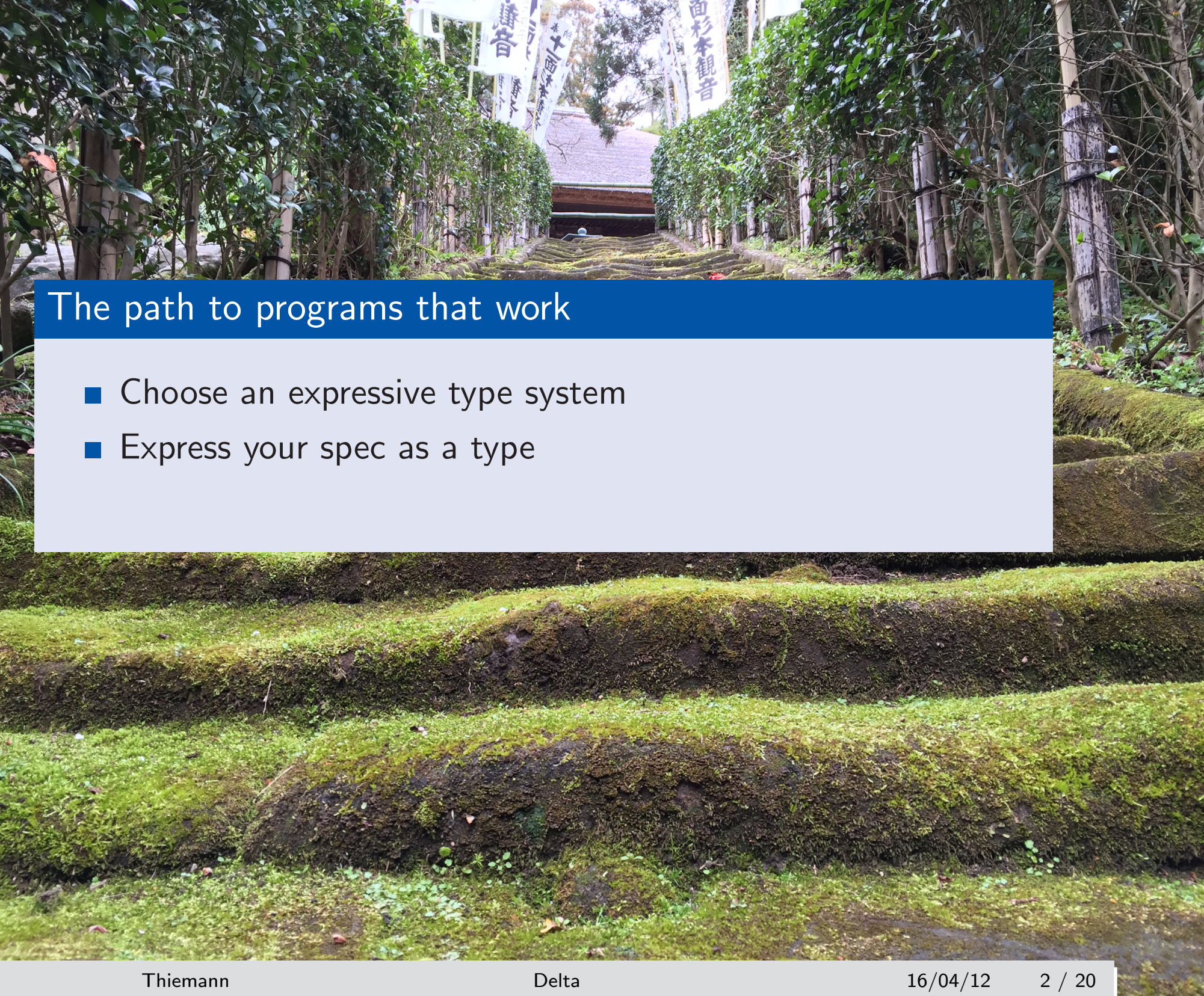


The path to programs that work



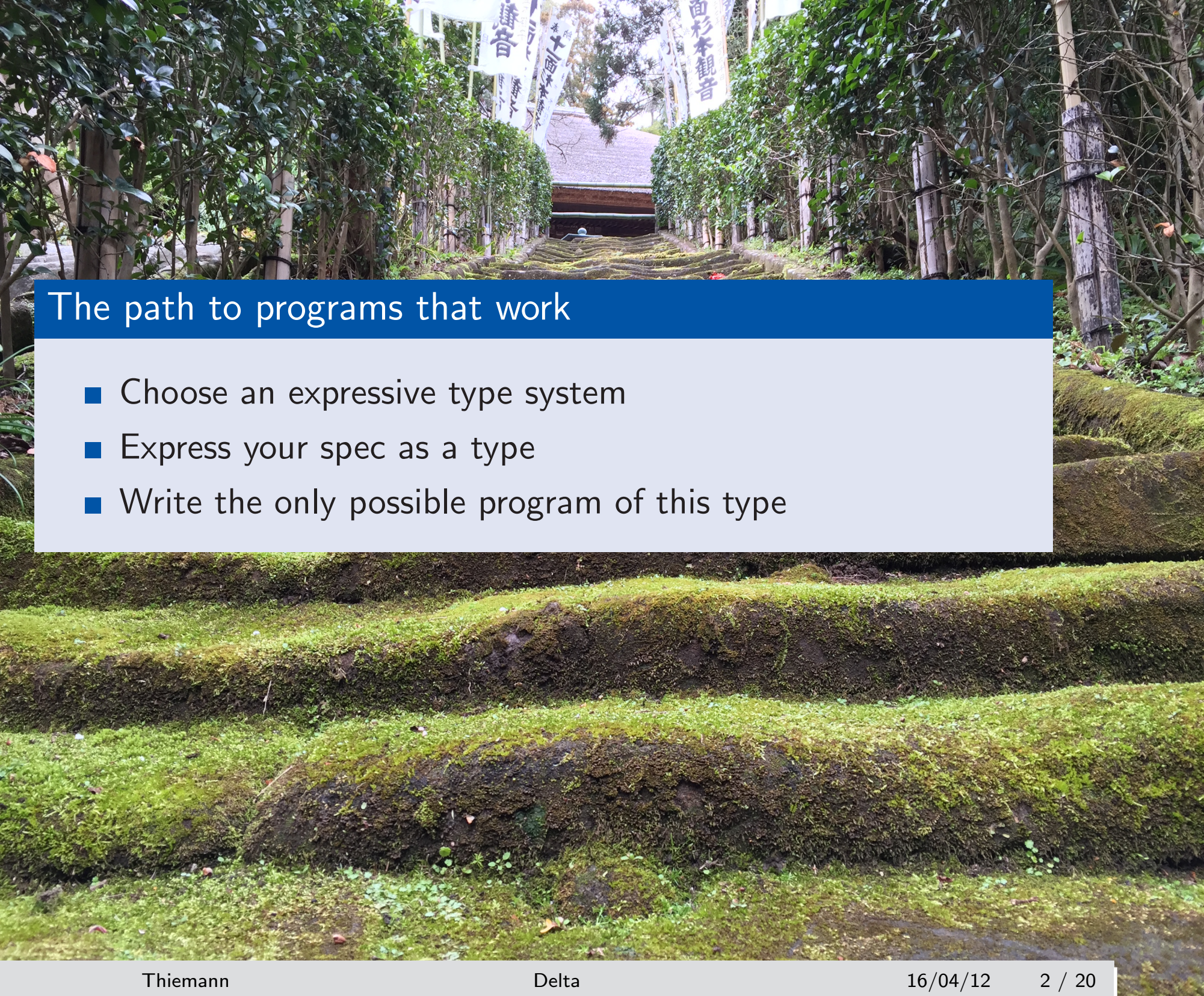
The path to programs that work

- Choose an expressive type system



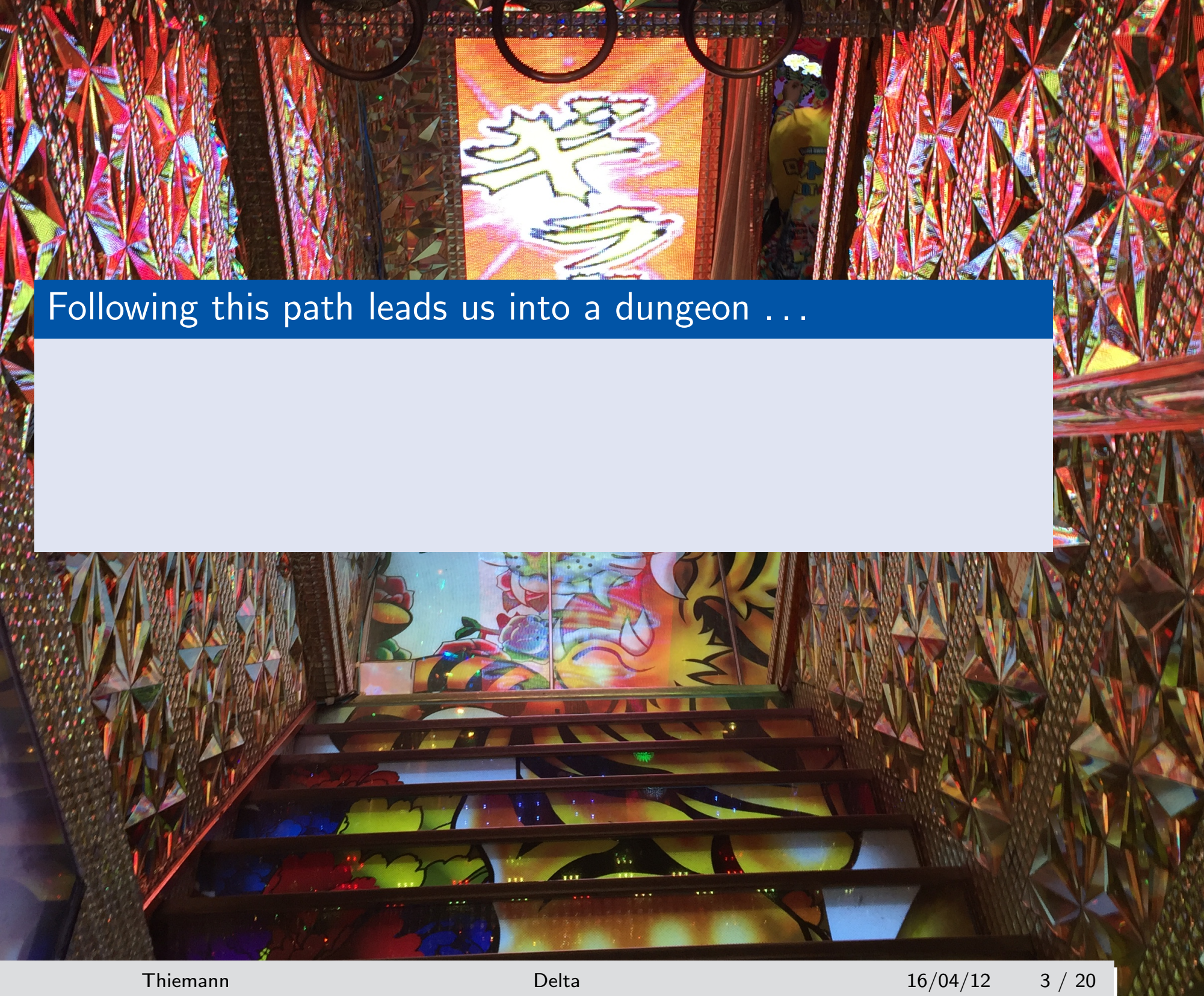
The path to programs that work

- Choose an expressive type system
- Express your spec as a type



The path to programs that work

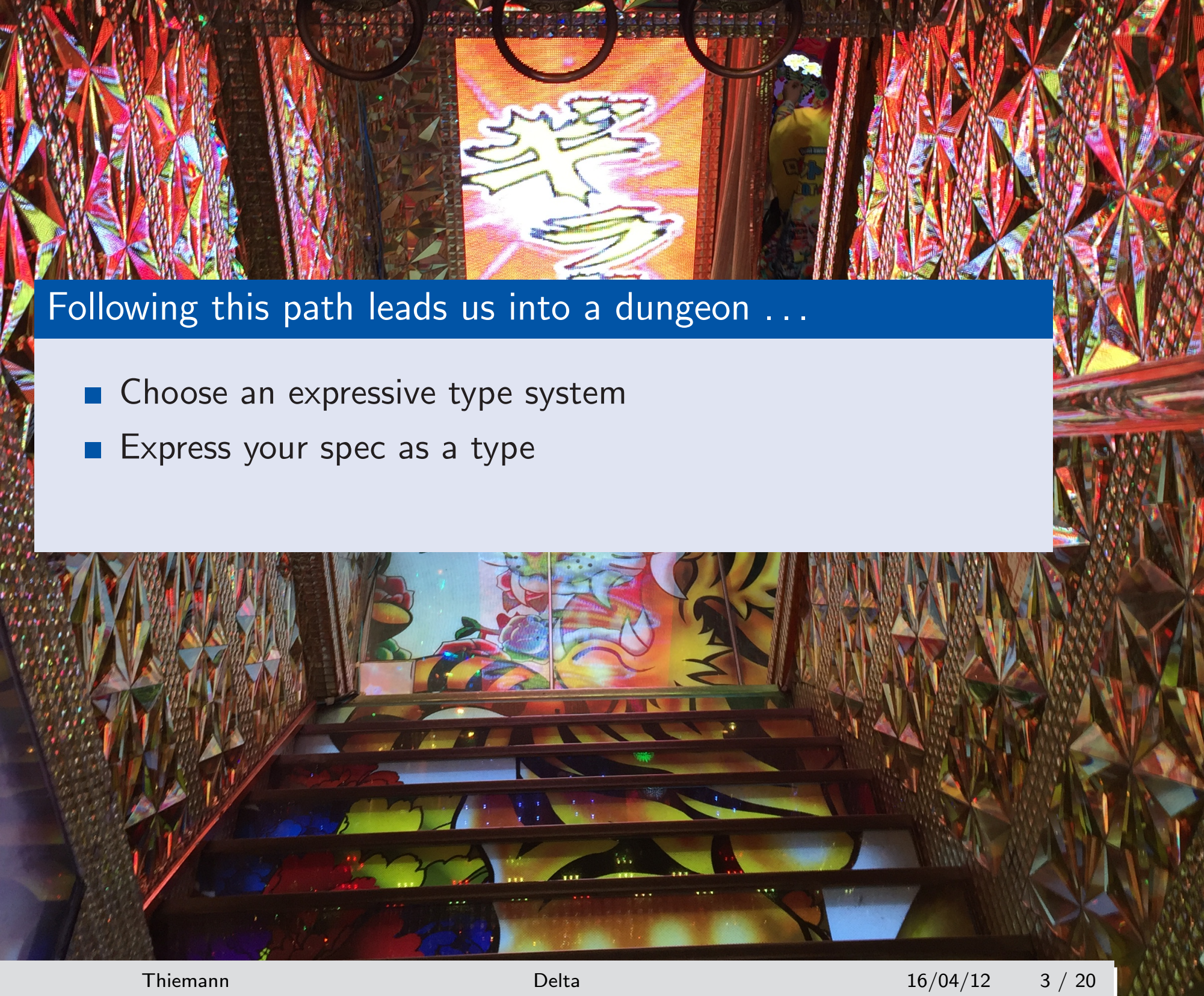
- Choose an expressive type system
- Express your spec as a type
- Write the only possible program of this type



Following this path leads us into a dungeon . . .

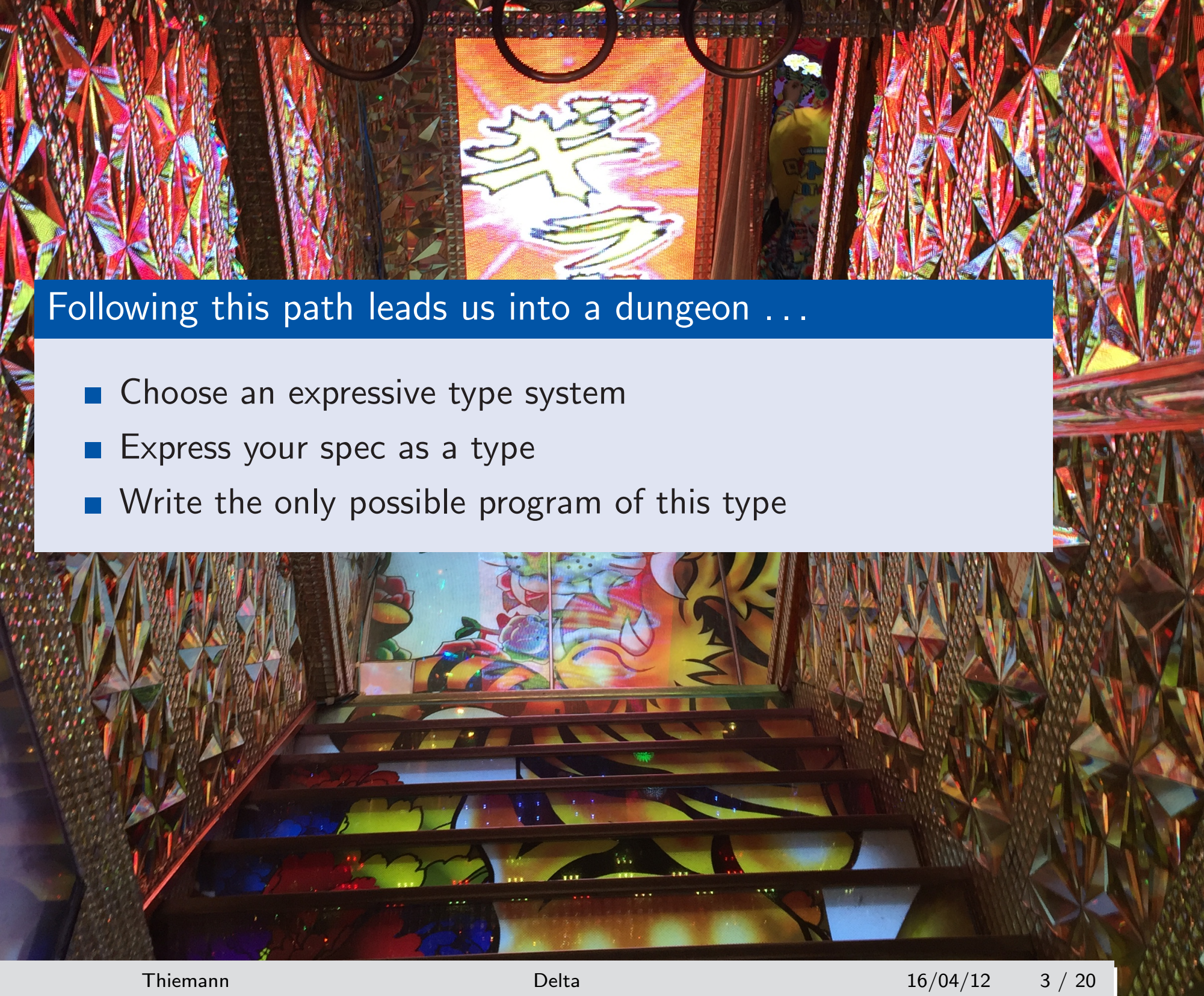
Following this path leads us into a dungeon . . .

- Choose an expressive type system



Following this path leads us into a dungeon . . .

- Choose an expressive type system
- Express your spec as a type



Following this path leads us into a dungeon . . .

- Choose an expressive type system
- Express your spec as a type
- Write the only possible program of this type

The type checker fails

Hybrid typing to the rescue!
[Knowles, Flanagan 2010]

Hybrid type checking
Back to some (thing) concrete

UNI FREIBURG

Refinement types

- $x : \mathsf{int}\{x > 0\}$
- $x : \mathsf{int}\{x\%2 = 0\}$

Hybrid type checking
Back to some (thing) concrete

UNI FREIBURG

Refinement types

- $x : \mathsf{int}\{x > 0\}$
- $x : \mathsf{int}\{x\%2 = 0\}$

Dependent function types

$x : \mathsf{int}\{x > 0\} \rightarrow y : \mathsf{int}\{y > x\}$

Hybrid type checking
Back to some (thing) concrete

UNI FREIBURG

Refinement types

- $x : \mathsf{int}\{x > 0\}$
- $x : \mathsf{int}\{x\%2 = 0\}$

Dependent function types

$x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}$

Subtyping dependent types

Aspinall Compagnoni 2001

Hybrid type checking
Back to some (thing) concrete

UNI FREIBURG

Refinement types

- $x : \mathsf{int}\{x > 0\}$
- $x : \mathsf{int}\{x\%2 = 0\}$

Dependent function types

$x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}$

Subtyping dependent types

Aspinall Compagnoni 2001

Cast — a blame calculus!

$M : S \Rightarrow T$

A concrete example

UNI FREIBURG

Assume

$F \quad : \quad x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}$

A concrete example

Assume

$$F \quad : \quad x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}$$

Apply a cast

$$\begin{aligned} G = (F \ : \quad & (x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x\}) \\ \Rightarrow \quad & (x : \mathsf{int}\{x > 0\} \to y : \mathsf{int}\{y > x \wedge y < 2 * x\})) \end{aligned}$$

A concrete example

Assume

$$F \quad : \quad x : \mathrm{int}\{x > 0\} \to y : \mathrm{int}\{y > x\}$$

Apply a cast

$$\begin{aligned} G = (F \ : \quad & (x : \mathrm{int}\{x > 0\} \to y : \mathrm{int}\{y > x\}) \\ \Rightarrow \quad & (x : \mathrm{int}\{x > 0\} \to y : \mathrm{int}\{y > x \wedge y < 2 * x\})) \end{aligned}$$

Run

$$(F\,42) \ : \ y : \mathrm{int}\{y > 42\} \Rightarrow y : \mathrm{int}\{y > 42 \wedge y < 2 * 42\}$$

Observations

Run

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

Observations

Run

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

- cast cannot be resolved by subtyping

Observations

Run

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

- cast cannot be resolved by subtyping
- requires run-time check

Observations

Run

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

- cast cannot be resolved by subtyping
- requires run-time check
- standard approach checks **entire predicate** $y > 42 \wedge y < 2 * 42$

Observations

Run

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

- cast cannot be resolved by subtyping
- requires run-time check
- standard approach checks **entire predicate** $y > 42 \wedge y < 2 * 42$
- but we already know $y > 42$!

Observations

Run

$$(F\,42)\ :\ y : \mathsf{int}\{y > 42\} \Rightarrow y : \mathsf{int}\{y > 42 \wedge y < 2 * 42\}$$

- cast cannot be resolved by subtyping
- requires run-time check
- standard approach checks **entire predicate** $y > 42 \wedge y < 2 * 42$
- but we already know $y > 42$!

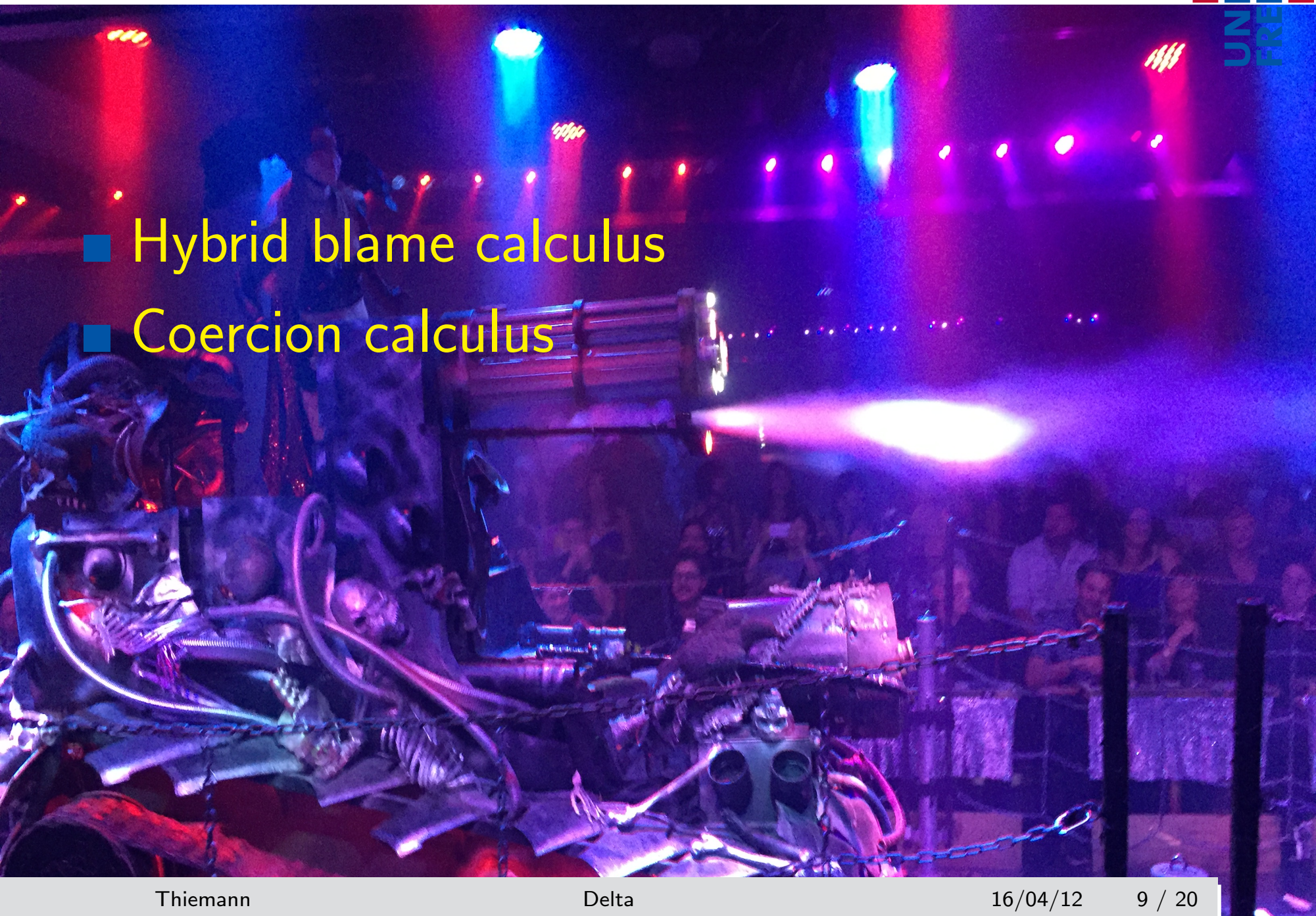
Our suggestion

- Only check the **delta**: $y < 2 * 42$

Plan of attack

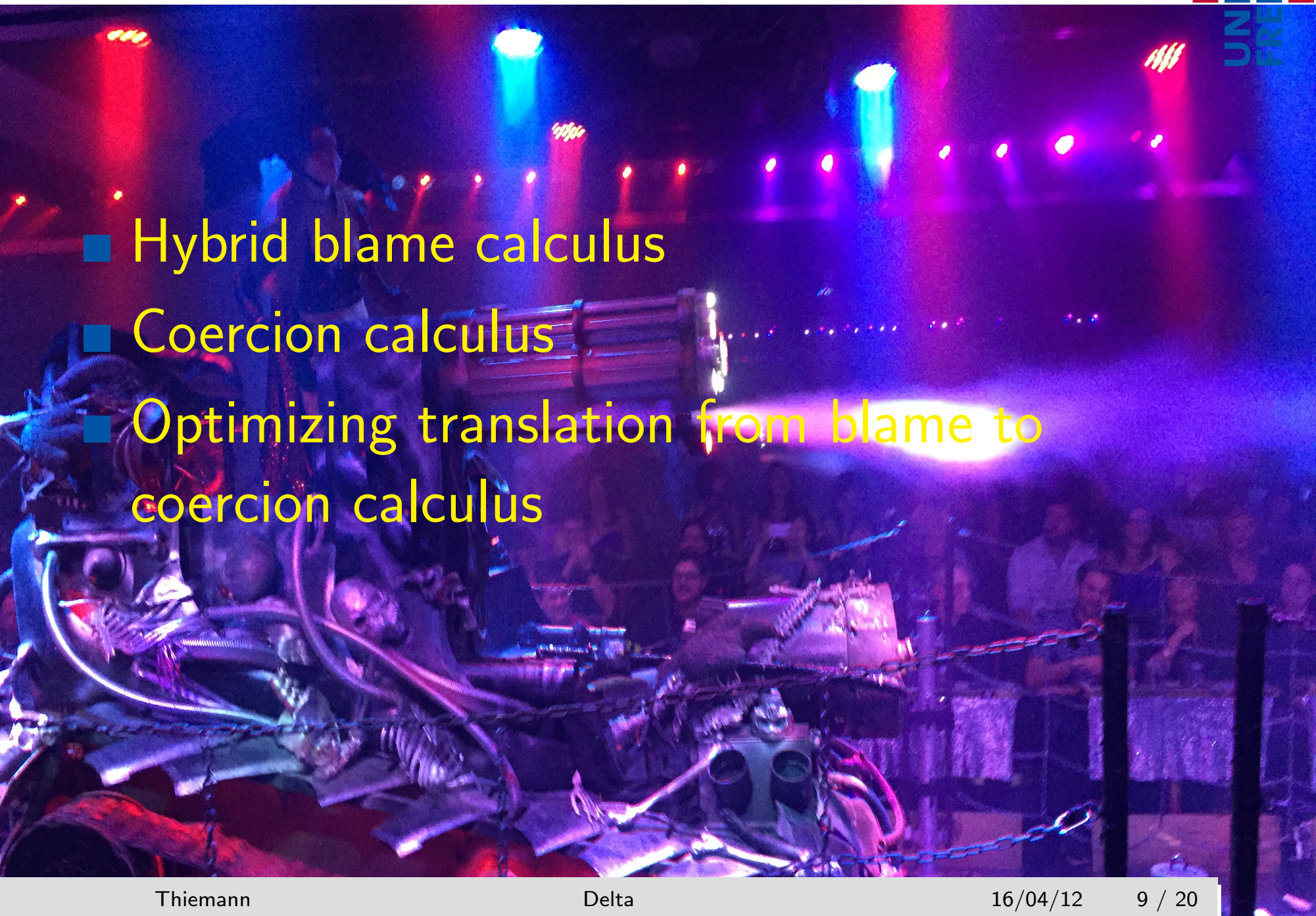
- Hybrid blame calculus

Plan of attack



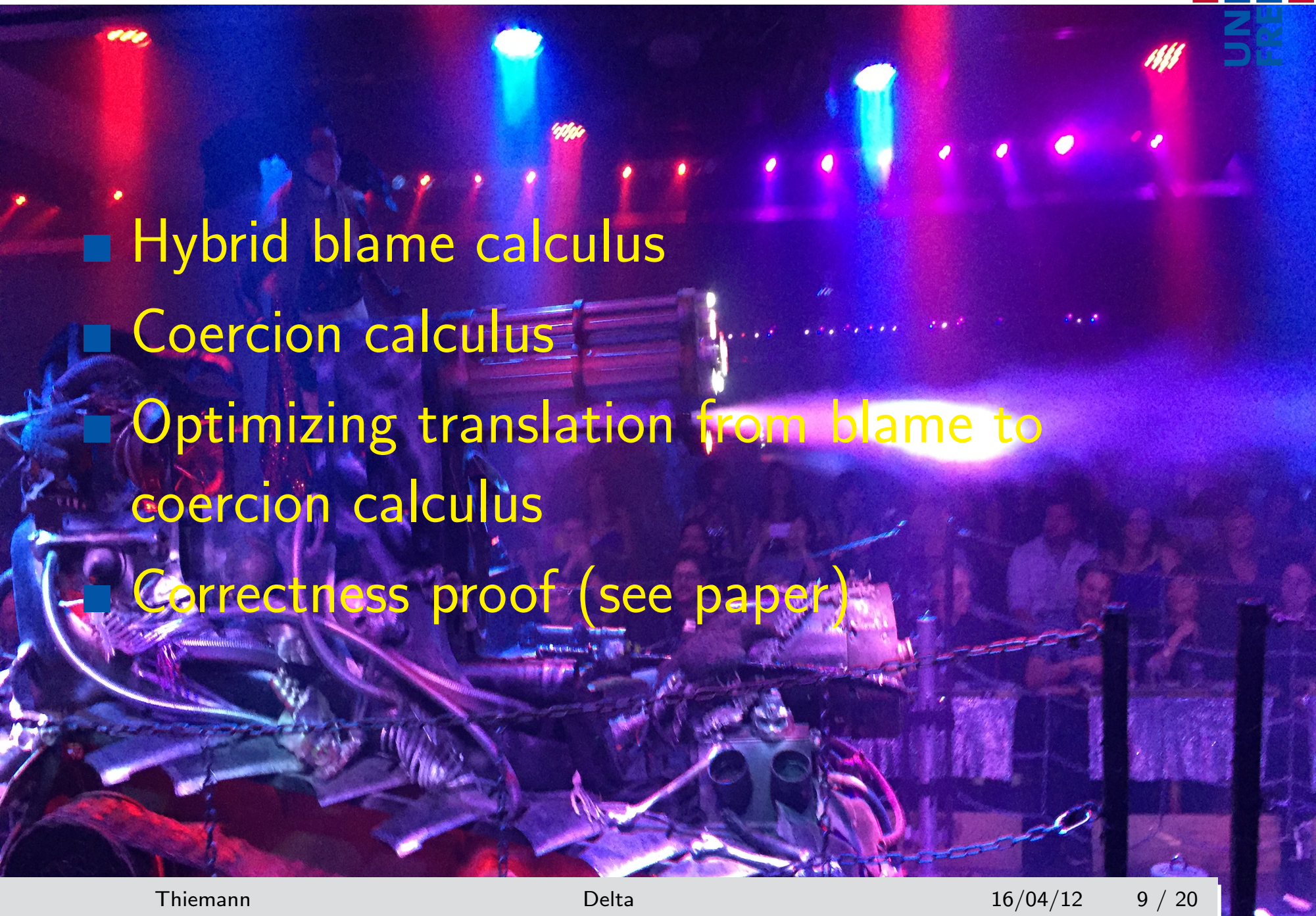
- Hybrid blame calculus
- Coercion calculus

Plan of attack



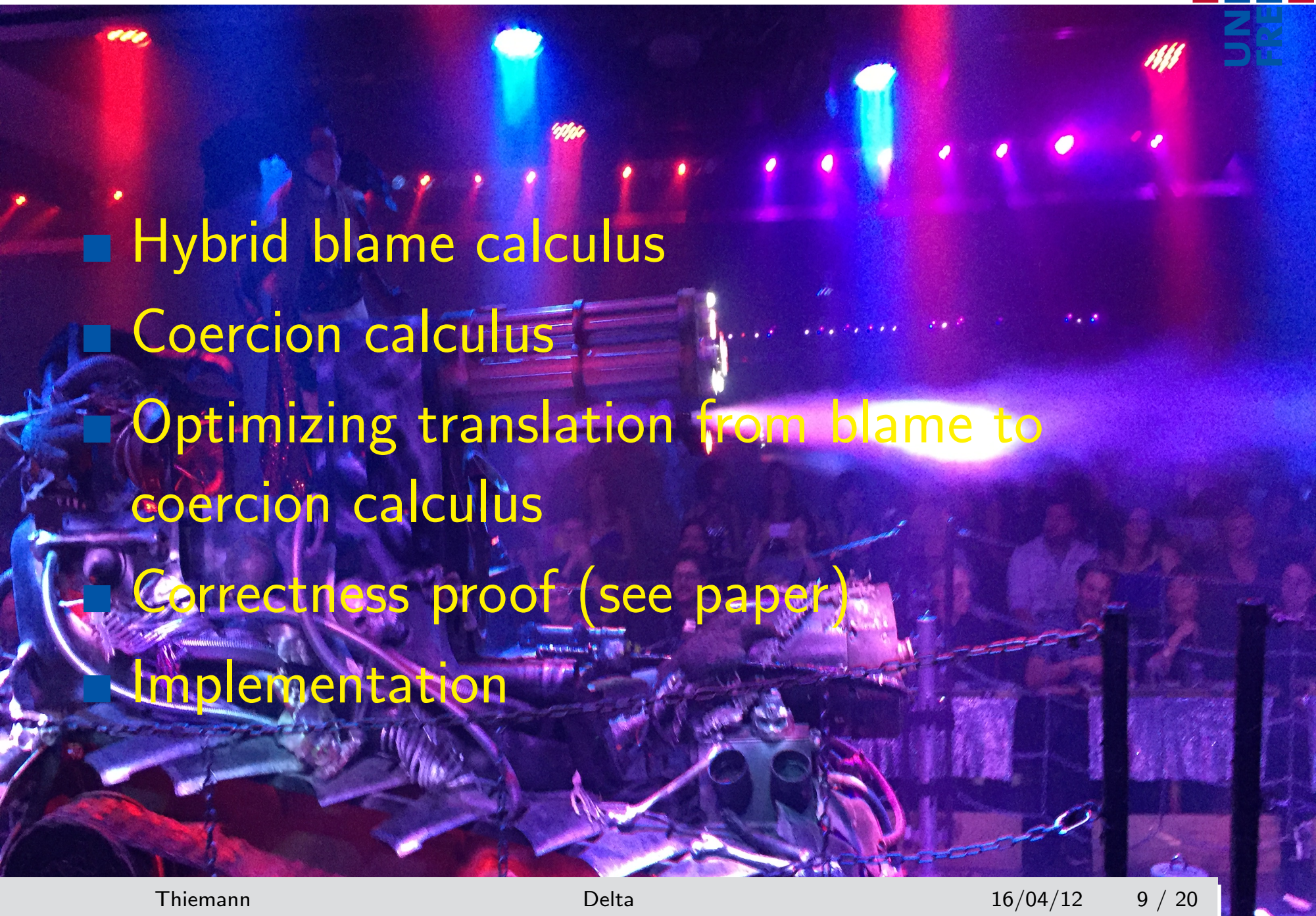
- Hybrid blame calculus
- Coercion calculus
- Optimizing translation from blame to coercion calculus

Plan of attack



- Hybrid blame calculus
- Coercion calculus
- Optimizing translation from blame to coercion calculus
- Correctness proof (see paper)

Plan of attack



- Hybrid blame calculus
- Coercion calculus
- Optimizing translation from blame to coercion calculus
- Correctness proof (see paper)
- Implementation

Blame Calculus
Selected rules

Type formation

$$\frac{\vdash \Gamma \text{ ctx} \qquad \lfloor \Gamma \rfloor, x : B \vdash P \text{ pred}}{\Gamma \vdash x : B\{P\} \text{ type}}$$

- restricted language of predicates
  (e.g., linear arithmetic formulas)
- free variables in refinements have first-order type

Blame Calculus
Selected rules

Type formation

$$\frac{\vdash \Gamma \text{ ctx} \qquad \lfloor \Gamma \rfloor, x : B \vdash P \text{ pred}}{\Gamma \vdash x : B\{P\} \text{ type}}$$

- restricted language of predicates (e.g., linear arithmetic formulas)
- free variables in refinements have first-order type

Typing — Dependent types vs call-by-value

$$\frac{\Gamma \vdash M : (x : S) \to T \qquad \Gamma \vdash N : S \qquad \Gamma \vdash T[N/x] \text{ type}}{\Gamma \vdash M\, N : T[N/x]}$$

- Application rule adapted from Sjöberg and others: avoids substitution of terms that raise blame

Findler-Felleisen Flue Woes

UNI FREIBURG

Recall the function cast

Findler-Felleisen 2002

$$(V : S \to T \Rightarrow S' \to T')\, W \quad \longrightarrow \quad (V\,(W : S' \Rightarrow S)) : T \Rightarrow T'$$

Findler-Felleisen Flue Woes

Recall the function cast

Findler-Felleisen 2002

$$(V : S \to T \Rightarrow S' \to T')\, W \quad \longrightarrow \quad (V\,(W : S' \Rightarrow S)) : T \Rightarrow T'$$

But it doesn't type check anymore!

- $V : (x : S)\ to T$ with x maybe appearing in T
- but on the rhs, substituting $(W : S' \Rightarrow S)$ for x should be well formed ...

Blame Calculus I

Function cast

Two reduction rules required for type preservation

Cast-Arg-Base

$$\frac{(W : S' \Rightarrow S) \longrightarrow W}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\, W \longrightarrow (V\, W) : (T \Rightarrow T')[W/x]}$$

Cast-Arg-Fun

$$\frac{(W : S' \Rightarrow S)\ \text{value}}{(V : (x : S) \to T \Rightarrow (x : S') \to T')\, W \longrightarrow (V\, (W : S' \Rightarrow S)) : (T \Rightarrow T')}$$

Blame Calculus II

Subtyping

Between refinement types

$$\frac{\vdash \Gamma \text{ ctx} \qquad \lfloor \Gamma \rfloor, x : B \vdash P \supset Q \text{ pred} \qquad \models \forall \left[\Gamma, x : B\{P\}\right].Q}{\Gamma \vdash x : B\{P\} \leq x : B\{Q\}}$$

Plus Aspinall/Compagnoni dependent function subtyping rule

$$\frac{\Gamma, x : S' \vdash T' \text{ type} \qquad \begin{array}{c}\Gamma \vdash S' \text{ type} \\ \Gamma \vdash S' \leq S\end{array} \qquad \Gamma, x : S' \vdash T \leq T'}{\Gamma \vdash (x : S) \to T \leq (x : S') \to T'}$$

Coercion Calculus

At base type

$$\frac{\forall \lceil \Gamma \rceil\, \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor \Gamma \rfloor, x : B \vdash P, Q, R \text{ pred}}{\Gamma \vdash x.R :: x : B\{P\} \Longrightarrow x : B\{Q\}}$$

Coercion Calculus

At base type

$$\frac{\forall \lceil \Gamma \rceil\ \forall x : B.\ P \supset (Q \Leftrightarrow R) \qquad \lfloor \Gamma \rfloor, x : B \vdash P, Q, R\ \mathsf{pred}}{\Gamma \vdash x.R :: x : B\{P\} \Longrightarrow x : B\{Q\}}$$

At function type

$$\frac{\Gamma \vdash k :: S' \Longrightarrow S \qquad \Gamma, x : S \curlywedge S' \vdash d :: T \Longrightarrow T'}{\Gamma \vdash (x : k) \to d :: (x : S) \to T \Longrightarrow (x : S') \to T'}$$

Blame calculus $\longrightarrow$ coercion calculus

UNI FREIBURG

Translation of refinement cast

$$\frac{\forall \lceil \Gamma \rceil\, \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor \Gamma \rfloor, x : B \vdash P, Q, R \text{ pred}}{\Gamma \vdash x : B\{P\} \Rightarrow x : B\{Q\} \rightsquigarrow x.R}$$

Blame calculus $\longrightarrow$ coercion calculus

Translation of refinement cast

$$\frac{\forall \lceil \Gamma \rceil\, \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor \Gamma \rfloor, x : B \vdash P, Q, R \text{ pred}}{\Gamma \vdash x : B\{P\} \Rightarrow x : B\{Q\} \rightsquigarrow x.R}$$

Translation of function cast

$$\frac{\Gamma \vdash S' \Rightarrow S \rightsquigarrow k \qquad \Gamma, x : S \curlywedge S' \vdash T \Rightarrow T' \rightsquigarrow d}{\Gamma \vdash (x : S) \to T \Rightarrow (x : S') \to T' \rightsquigarrow (x : k) \to d}$$

- Problem: what's the relation between S, S', and T'?

Blame calculus $\longrightarrow$ coercion calculus

Translation of refinement cast

$$\frac{\forall \lceil \Gamma \rceil\, \forall x : B.\, P \supset (Q \Leftrightarrow R) \qquad \lfloor \Gamma \rfloor, x : B \vdash P, Q, R \text{ pred}}{\Gamma \vdash x : B\{P\} \Rightarrow x : B\{Q\} \rightsquigarrow x.R}$$

Translation of function cast

$$\frac{\Gamma \vdash S' \Rightarrow S \rightsquigarrow k \qquad \Gamma, x : S \curlywedge S' \vdash T \Rightarrow T' \rightsquigarrow d}{\Gamma \vdash (x : S) \to T \Rightarrow (x : S') \to T' \rightsquigarrow (x : k) \to d}$$

- Problem: what's the relation between S, S', and T'?
- Solution: first-order conjunction $\curlywedge$

Question

How do we obtain R?

- Given Γ, P, and Q
- Needed R such that $\forall \left[\Gamma\right] \forall x : B.\, P \supset (Q \Leftrightarrow R)$

Question

How do we obtain R?

- Given Γ, P, and Q
- Needed R such that $\forall \left[\Gamma\right] \forall x : B.\, P \supset (Q \Leftrightarrow R)$

Program synthesis!

- Using the Rosette system [Torlak & Bodik]
- Embedded in Racket

Synthesis for linear arithmetic

Grammar of predicates

$$r ::= \text{true} \mid \text{false} \mid p$$

$$p, q ::= z \leq c \mid c \leq z \mid z \leq y \mid \neg p \mid p \wedge q \mid p \vee q$$

Synthesis for linear arithmetic

Grammar of predicates

$$r ::= \text{true} \mid \text{false} \mid p$$

$$p, q ::= z \leq c \mid c \leq z \mid z \leq y \mid \neg p \mid p \wedge q \mid p \vee q$$

Synthesis code

```
(evaluate
 (grammar vars depth)
 (synthesize
  #:forall vars
  #:guarantee
  (assert
   (=> (interpret p)
       (<=> (interpret q) (interpret r))))))
```

Delta predicates synthesized by Rosette

P given	Q required	R tested
$1 \leq x \wedge x \leq 10$	$5 \leq x \wedge x \leq 15$	$5 \leq x$
$5 \leq x \wedge x \leq 15$	$1 \leq x \wedge x \leq 10$	$x \leq 10$
$1 \leq x \leq 10 \wedge 0 \leq y \leq 9$	$5 \leq x \leq 15 \wedge 3 \leq y \leq 6$	$3 \leq y \leq 6 \wedge 5 \leq x$
$1 \leq x \leq 9 \wedge 1 \leq y \leq 5$	$1 \leq x \leq 5 \wedge x \leq y$	$x \leq y$
$0 \leq x \leq 5 \vee 10 \leq x \leq 15$	$6 \leq x \leq 9$	false
$6 \leq x \leq 9$	$5 \leq x \leq 10$	true

Delta predicates synthesized by Rosette

P given	Q required	R tested
$1 \leq x \wedge x \leq 10$	$5 \leq x \wedge x \leq 15$	$5 \leq x$
$5 \leq x \wedge x \leq 15$	$1 \leq x \wedge x \leq 10$	$x \leq 10$
$1 \leq x \leq 10 \wedge 0 \leq y \leq 9$	$5 \leq x \leq 15 \wedge 3 \leq y \leq 6$	$3 \leq y \leq 6 \wedge 5 \leq x$
$1 \leq x \leq 9 \wedge 1 \leq y \leq 5$	$1 \leq x \leq 5 \wedge x \leq y$	$x \leq y$
$0 \leq x \leq 5 \vee 10 \leq x \leq 15$	$6 \leq x \leq 9$	false
$6 \leq x \leq 9$	$5 \leq x \leq 10$	true

Minimal cost

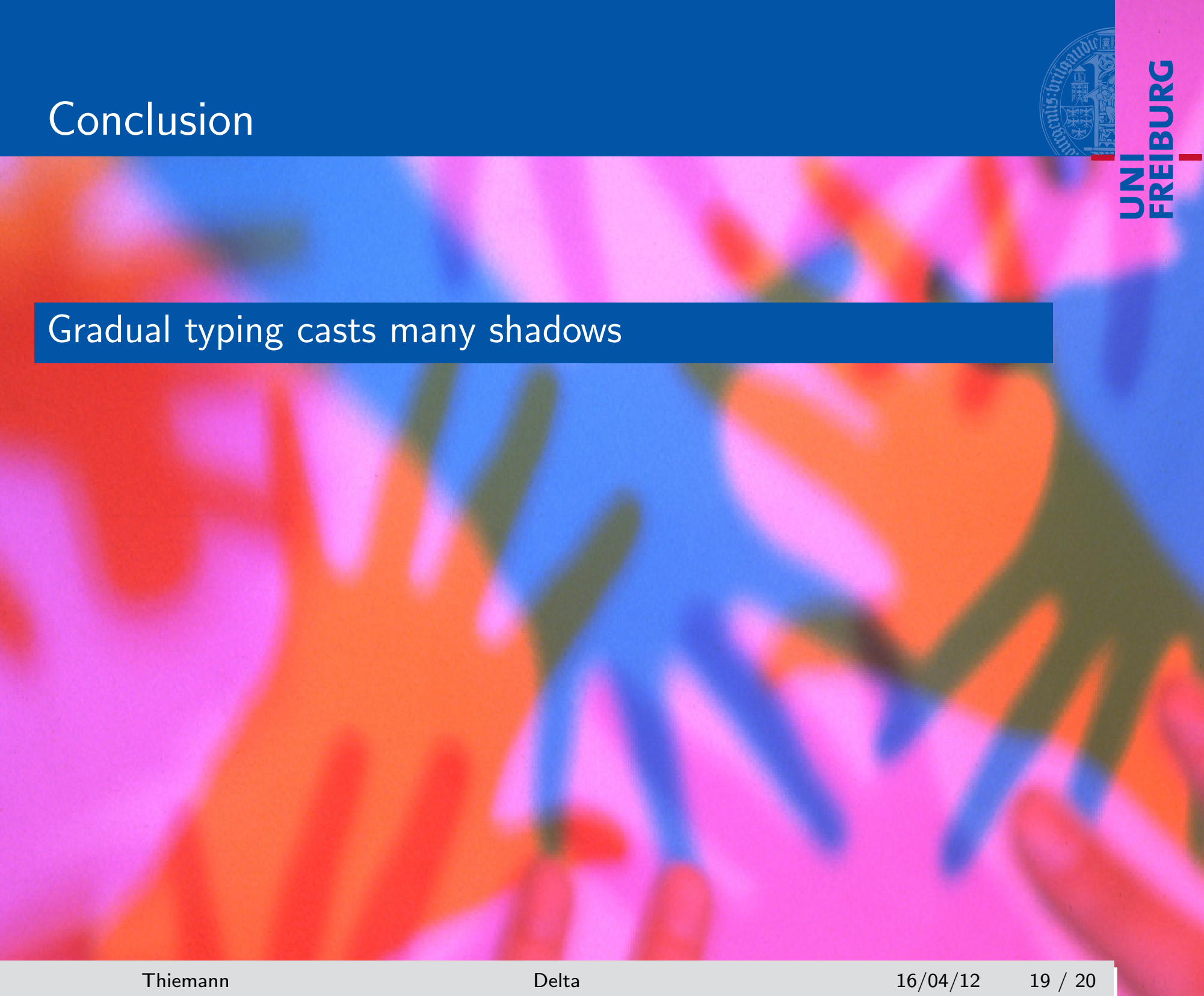
$$cost(\text{true}) = cost(\text{false}) = 0$$

$$cost(z \leq c) = cost(c \leq z) = cost(y \leq z) = 1$$

$$cost(\neg p) = 1 + cost(p)$$

$$cost(p \wedge q) = cost(p \vee q) = 1 + cost(p) + cost(q)$$

Conclusion



Gradual typing casts many shadows

Conclusion

Gradual typing casts many shadows

Naive implementation of casts ignores static knowledge

- from source type of cast
- from precondition of dependent function

Conclusion

UNI FREIBURG

Gradual typing casts many shadows

Naive implementation of casts ignores static knowledge

- from source type of cast
- from precondition of dependent function

Translation "blame $\rightarrow$ coercion" strikes again

Optimization is "obvious" in coercion form

Conclusion

Gradual typing casts many shadows

Naive implementation of casts ignores static knowledge

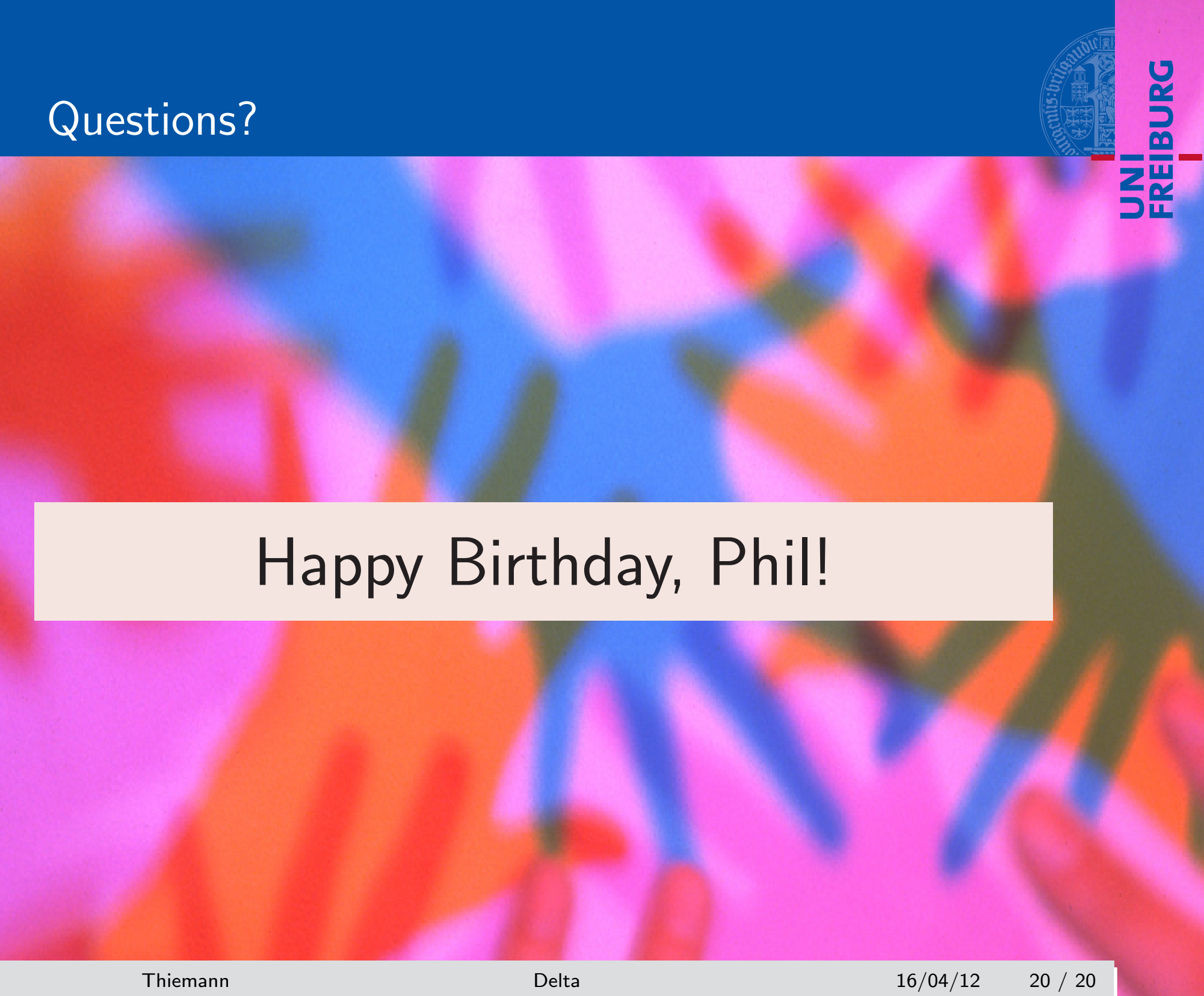
- from source type of cast
- from precondition of dependent function

Translation "blame $\rightarrow$ coercion" strikes again

Optimization is "obvious" in coercion form

Delta predicate determined by program synthesis

Questions?



Happy Birthday, Phil!