Why some people use functional languages

Philip Wadler
University of Edinburgh / IOHK
Loft, São Paulo
27 November 2019

Haskell

Use Case 1:
Haxl @ Facebook

<Haxl>

```haskell
blog :: Fetch Html
blog = renderPage <$> leftPane <*> mainPane


leftPane :: Fetch Html
leftPane = renderSidePane <$> popularPosts <*> topics


mainPane :: Fetch Html
mainPane = do
  posts <- getAllPostsInfo
  let ordered =
        take 5 $
        sortBy (flip (comparing postDate)) posts
  content <- mapM (getPostContent . postId) ordered
  return $ renderPosts (zip ordered content)


popularPosts :: Fetch Html
popularPosts = do
  pids <- getPostIds
  views <- mapM getPostViews pids
  let ordered =
        take 5 $ map fst $
        sortBy (flip (comparing snd))
               (zip pids views)
  content <- mapM getPostDetails ordered
  return $ renderPostList content
```

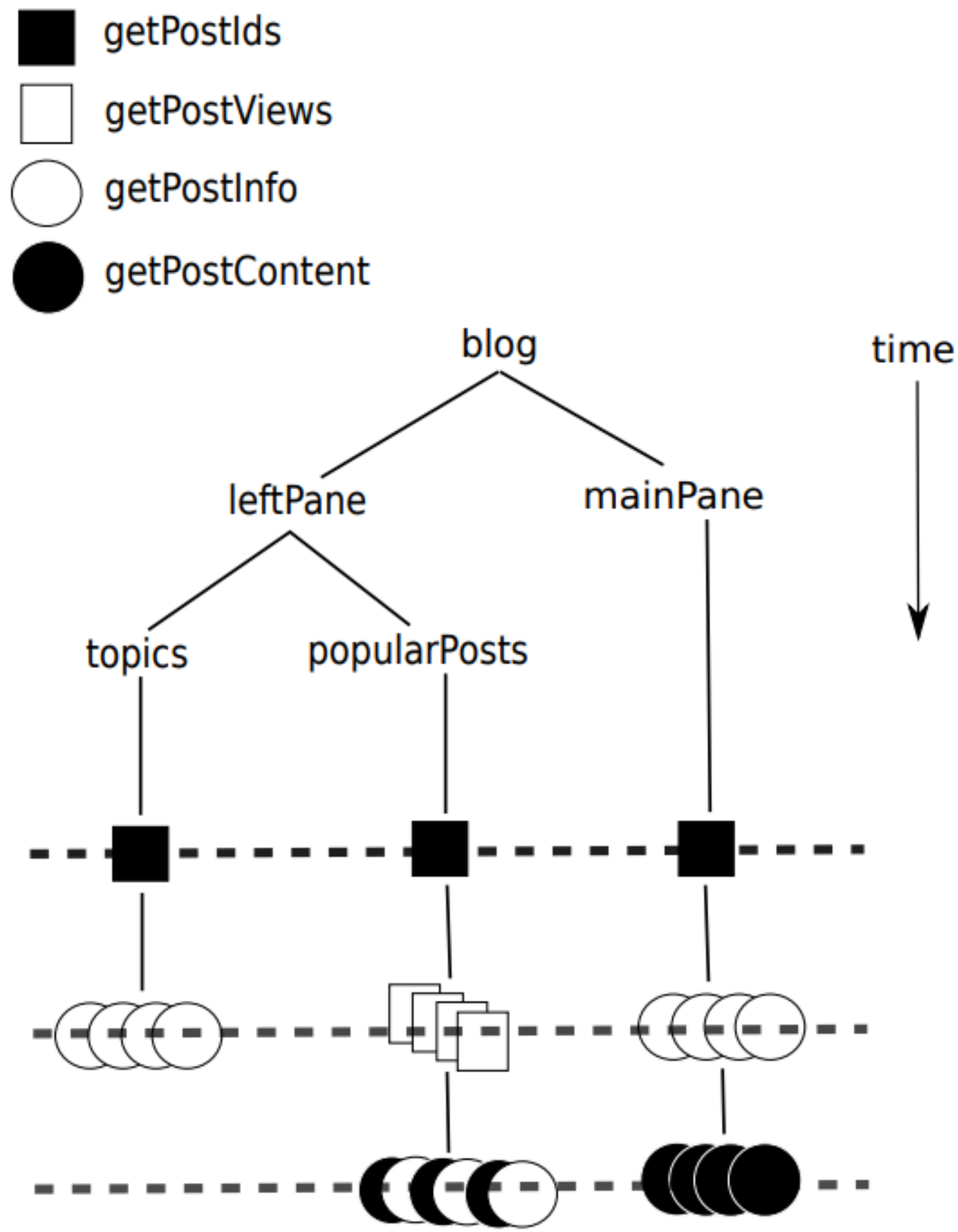


Figure 1. Data fetching in the blog example

Use Case 2:
SeL4

Security. Performance. Proof.

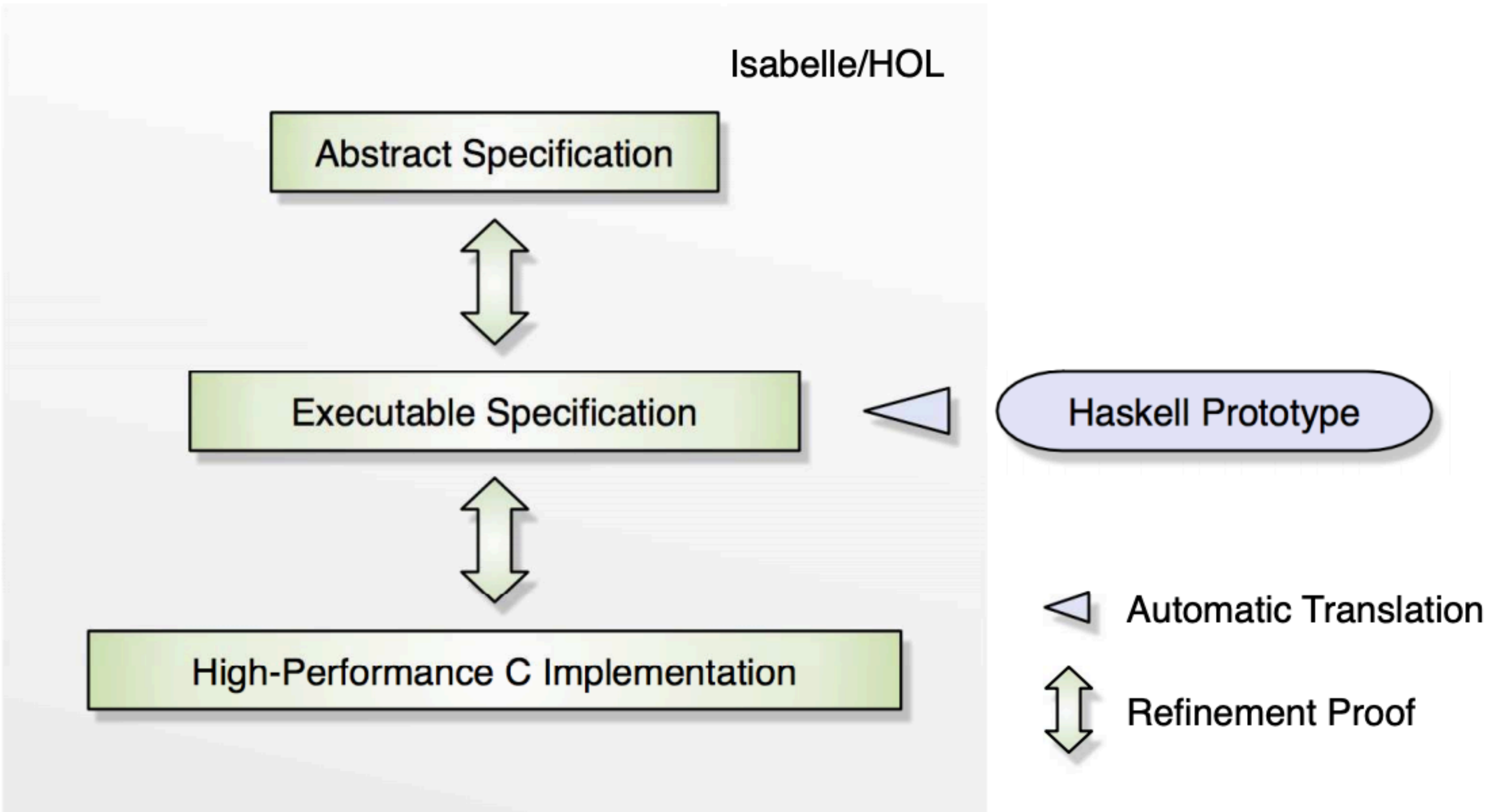


Figure 1. Specification layers in the L4.verified project.

```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

Figure 3: Isabelle/HOL code for scheduler at abstract level.

```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread
chooseThread' prio = do
     q <- getQueue prio
     liftM isJust $ findM chooseThread'' q
chooseThread'' thread = do
     runnable <- isRunnable thread
     if not runnable then do
             tcbSchedDequeue thread
             return False
     else do
             switchToThread thread
             return True
```

Figure 4: Haskell code for `schedule`.

```
void setPriority(tcb_t *tptr, prio_t prio) {
  prio_t oldprio;
  if(thread_state_get_tcbQueued(tptr->tcbState)) {
    oldprio = tptr->tcbPriority;
    ksReadyQueues[oldprio] =
      tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
    if(isRunnable(tptr)) {
      ksReadyQueues[prio] =
        tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
    }
    else {
      thread_state_ptr_set_tcbQueued(&tptr->tcbState,
                                               false);
    }
  }
  tptr->tcbPriority = prio;
}
```

Figure 5: C code for part of the scheduler.

Use Case 3:
Finance

ABN·AMRO

Tsuru Capital

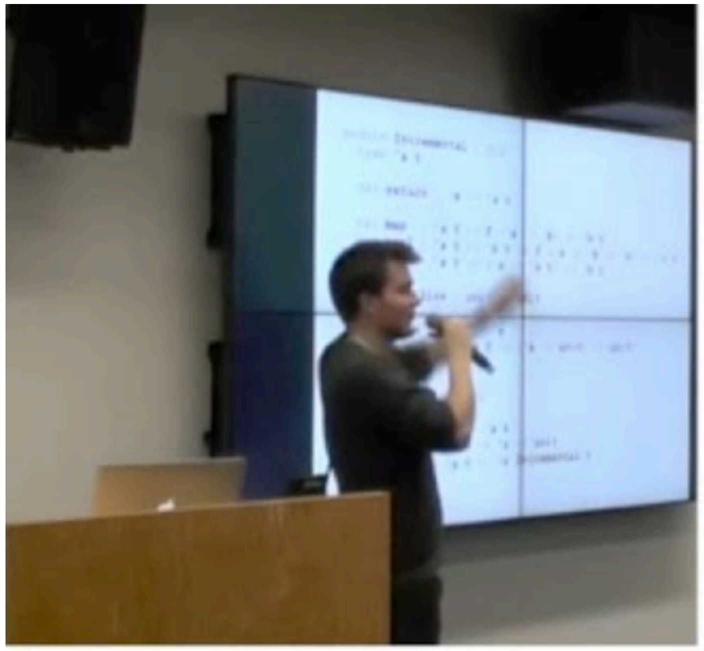
Bank of America

Standard Chartered

Credit Suisse

Barclays

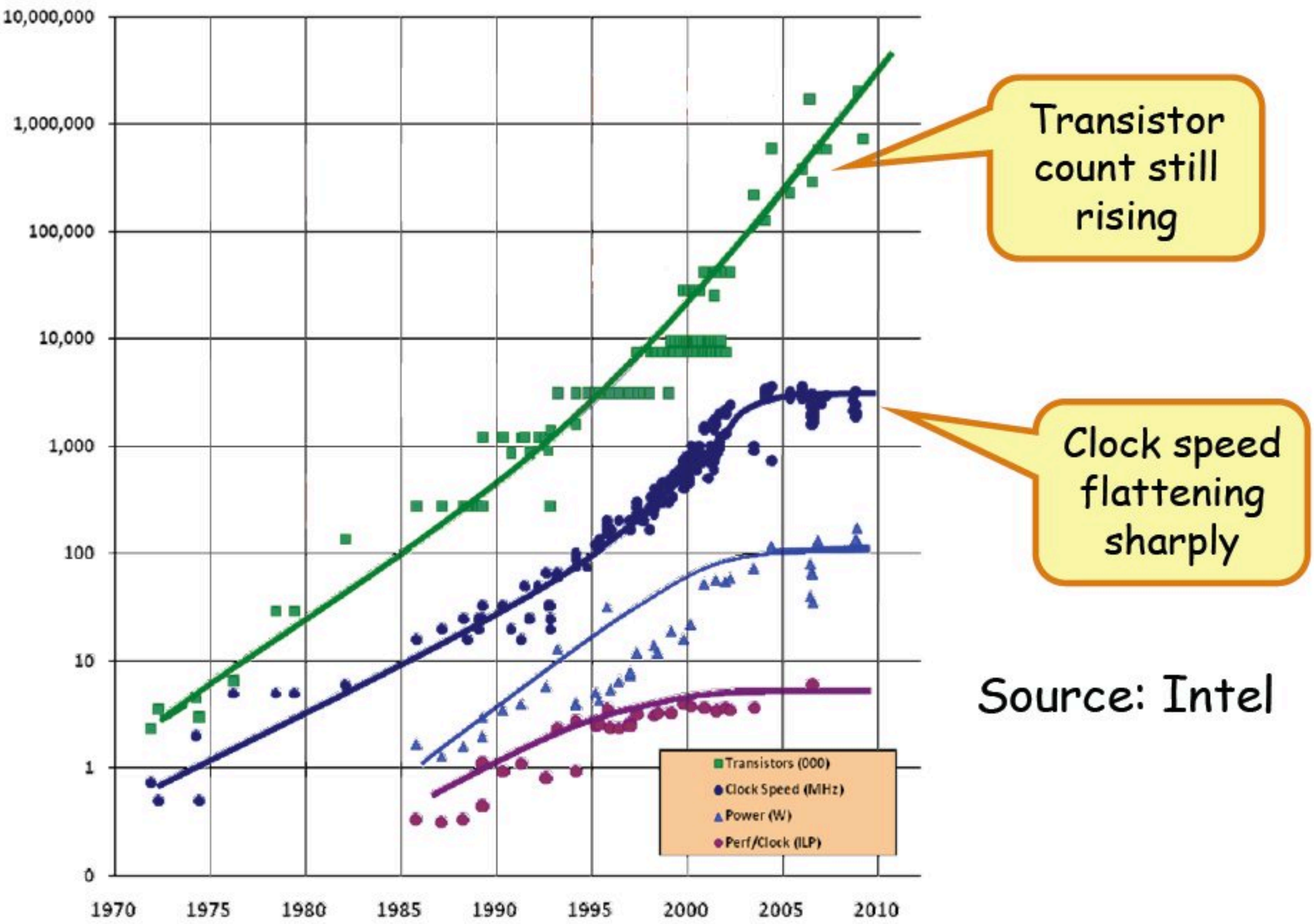
Deutsche Bank



```
module Incremental : sig
  type 'a t

  val return : 'a -> 'a t

  val map  : 'a t -> f:('a -> 'b) -> 'b t
  val map2 : 'a t -> 'b t -> f:('a -> 'b -> 'c) -> 'c t
  val bind : 'a t -> ('a -> 'b t) -> 'b t

  val stabilize : unit -> unit

  val value : 'a t -> 'a
  val on_update : 'a t -> ('a -> unit) -> unit
end

module Variable : sig
  type 'a t
  val create : 'a -> 'a t
  val set    : 'a t -> 'a -> unit
  val read   : 'a t -> 'a Incremental.t
```

Jane Street

Concurrency



Source: Intel

Agda for Fun and Profit: System F


Philip Wadler


Manuel Chakravarty


Vanessa McHale


Michael Peyton Jones


Kris Jenkins


Simon Thompson


Roman Kireev


Kenneth MacKenzie


Rebecca Valentine (Former Member)


James Chapman


Jann Müller


David Smith


Pablo Lamela Seijas


Alexander Nemish

Plutus Core

Kinds

```
J,K ::=
    *
    J → K
```

Types

```
A,B ::=
    X
    A → B
    ∀X.B
    μX.B
    ρ
```

Terms

```
L,M,N ::=
    x
    λx:A.N
    L M
    ΛX:K.N
    L A
    wrap M
    unwrap M
    ρ
```

Plutus Core in Agda

```
data Kind : Set where
  *   : Kind
  _⇒_ : Kind → Kind → Kind

data _⊢⋆_ : Ctx⋆ → Kind → Set where
  `   : Φ ∋⋆ J
        ------
      → Φ ⊢⋆ J

  ƛ   : Φ ,⋆ K ⊢⋆ J
        -----------
      → Φ ⊢⋆ K ⇒ J

  _·_ : Φ ⊢⋆ K ⇒ J
      → Φ ⊢⋆ K
        ----------
      → Φ ⊢⋆ J

  Π   : Φ ,⋆ K ⊢⋆ *
        -----------
      → Φ ⊢⋆ *

  _⇒_ : Φ ⊢⋆ *
      → Φ ⊢⋆ *
        ------
      → Φ ⊢⋆ *
```

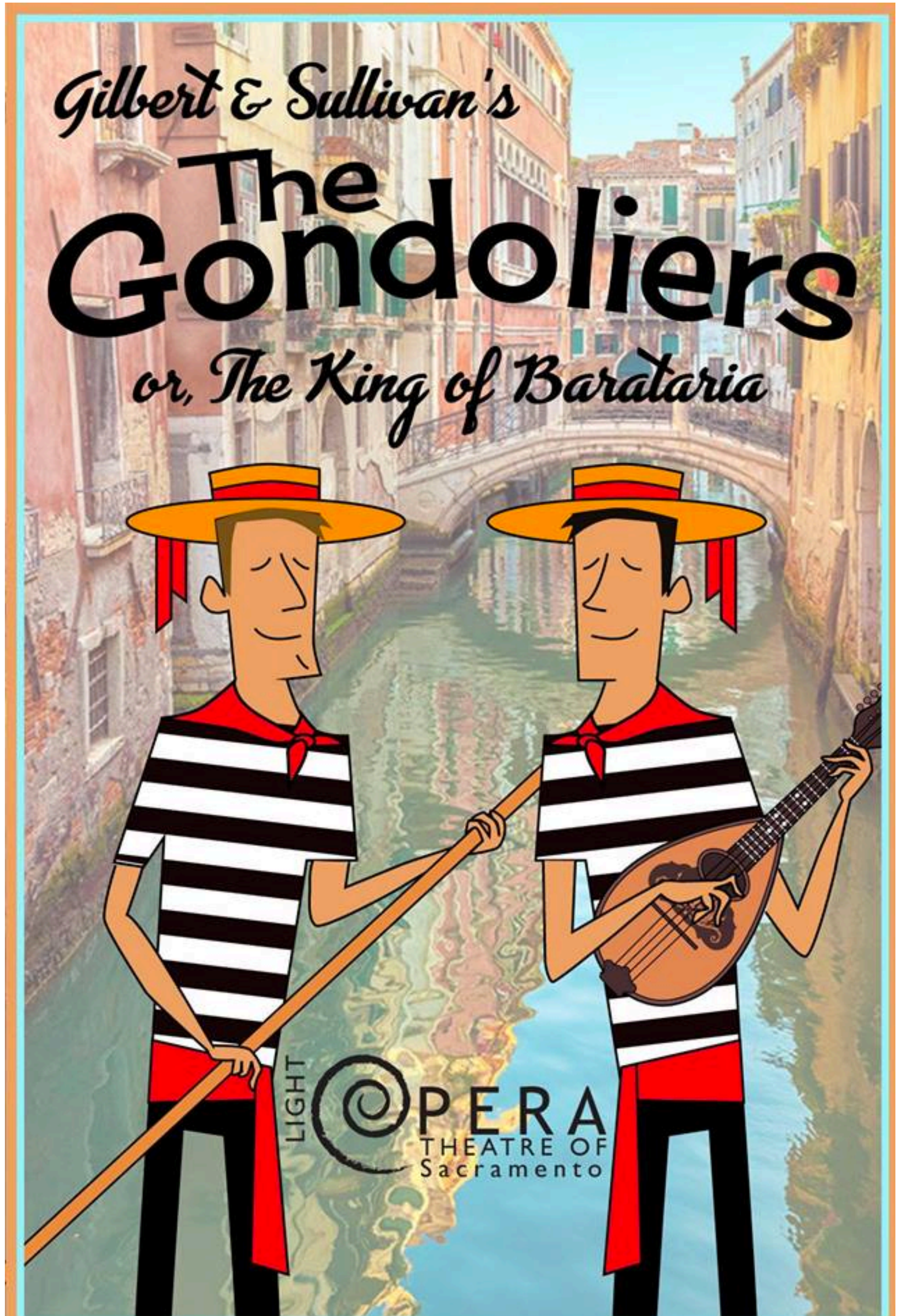
```
data _⊢_ : ∀ Γ → ∥ Γ ∥ ⊢⋆ J → Set where
  `     : Γ ∋ A
          -----
        → Γ ⊢ A

  ƛ     : Γ , A ⊢ B
          ---------
        → Γ ⊢ A ⇒ B

  _·_   : Γ ⊢ A ⇒ B
        → Γ ⊢ A
          ---------
        → Γ ⊢ B

  Λ     : Γ ,⋆ K ⊢ B
          ----------
        → Γ ⊢ Π B

  _·⋆_  : Γ ⊢ Π B
        → (A : ∥ Γ ∥ ⊢⋆ K)
          ---------------
        → Γ ⊢ B [ A ]

  conv  : A ≡β B
        → Γ ⊢ A
          -----
        → Γ ⊢ B
```

Roman Kireev 3 months ago

I haven't talked with James except for a couple of messages, but I read what he wrote in Agda and I'm very surprised that you can formalize System F in a non-disgusting way. Or at least I do not see those huge clunky theorems which I see everywhere including my own attempts

A Profound Pun

$$\text{yoghurt} \vee \neg\text{yoghurt}$$


Gilbert & Sullivan's
The Gondoliers
or, The King of Barataria
LIGHT OPERA
THEATRE OF
Sacramento

A & B

$$A \rightarrow B$$

$$A \lor B$$

$$A \mathbin{\&} B$$

$$A \vee B$$

$$A \rightarrow B$$

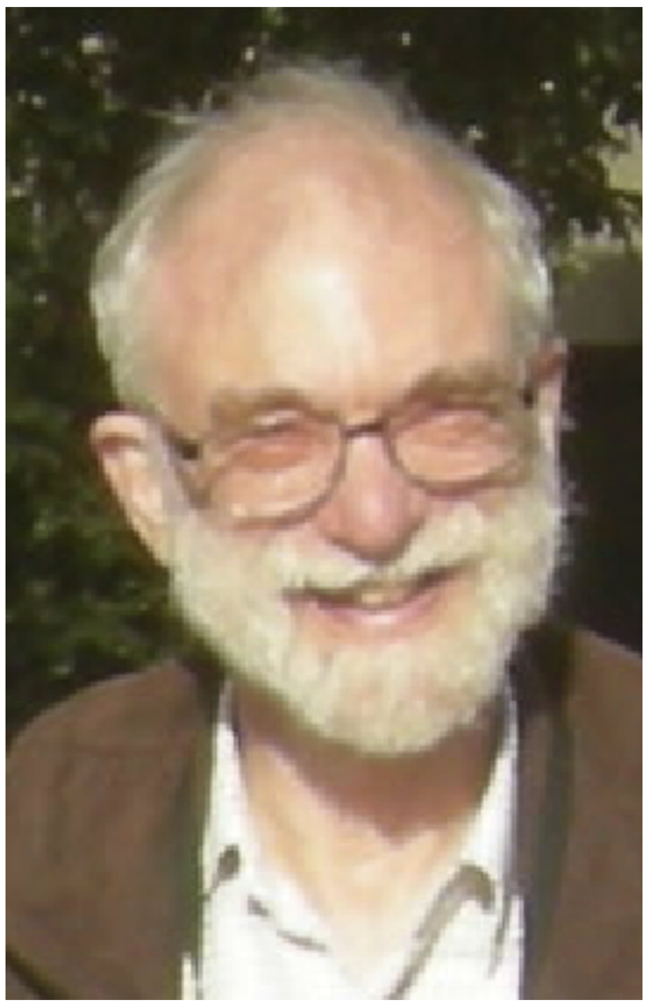
Lambda Calculus

Alonzo Church, 1932-40

Natural Deduction

Gerhard Gentzen, 1935

Principle Type Scheme



Roger Hindley, 1969

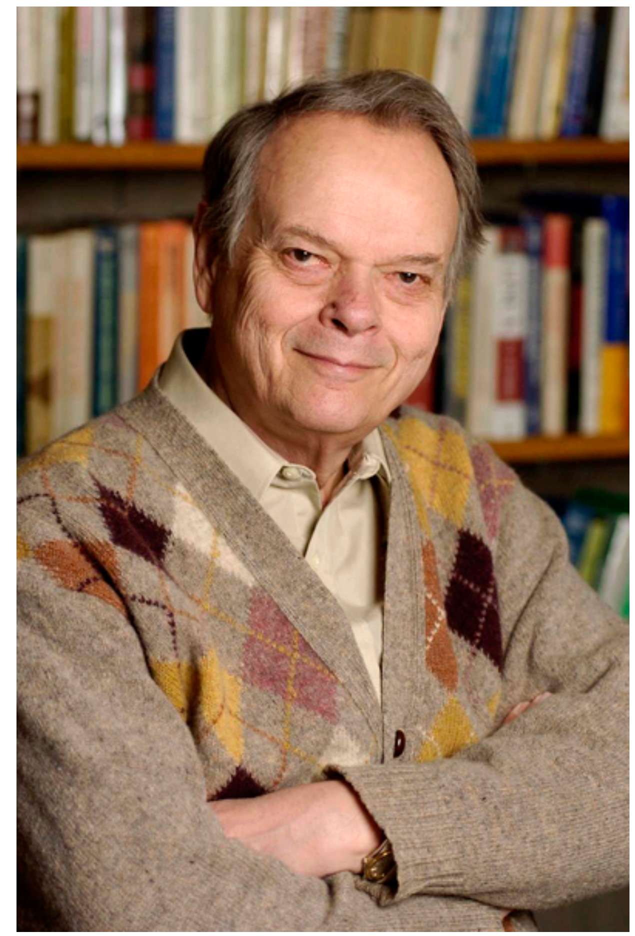
Type Polymorphsim

Robin Milner, 1978

System F

Jean-Yves Girard, 1972

Polymorphic
Lambda Calculululus



John Reynolds, 1974

JS

php

R

Object-Oriented Programming

Alternative

- F-bounded polymorphism
  - » max : ∀ t <: □□greater : t→bool□□. t ´ t → t
- Higher-order bounded polymorphism
  - » max : ∀F <: λt. □□greater : t→bool□□. μF ´ μF → μF
- Similar in spirit but technical differences
  - » Transitive relation
  - » “Standard” bounded quantificaion

Translation of Types:

$$|\mathtt{C}| = \mathtt{C} \quad |\mathtt{C.E}| = \mathtt{C\$EFix}$$
$$|\mathtt{X}| = \mathtt{X} \quad |\mathtt{X.E}| = \mathtt{X\$E} \quad |\mathtt{.E}| = \mathtt{E}$$

Definition of *nestedclasses*

$$\mathit{nestedclasses}(\mathtt{Object}) = \bullet$$

$$\frac{\mathtt{class\ C}\triangleleft\mathtt{D\{\ ..\ \overline{NL}\ \}} \quad \mathit{nestedclasses}(\mathtt{D}) = \overline{\mathtt{E}}'}{\mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}}', \{\mathtt{E} \mid \mathtt{E} \notin \overline{\mathtt{E}}', \mathtt{class\ E\{..\}} \in \overline{\mathtt{NL}}\}}$$

Translation of Type Arguments:

$$\frac{\mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}}}{|\mathtt{F}|_{\mathtt{C}} = |\mathtt{F}.\overline{\mathtt{E}}|, |\mathtt{F}|}$$

Translation of Expressions:

$$|\mathtt{x}|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{x}$$

$$|\mathtt{e}_0.\mathtt{f}_i|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{f}_i$$

$$\frac{\Delta;\Gamma;\mathtt{A} \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathit{mtype}_{.\mathrm{FJ}}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0@\mathtt{A})) = \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0}{|\mathtt{e}_0.\mathtt{<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{e}})|_{\Delta,\Gamma,\mathtt{A}} = |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}}.\mathtt{<}|\overline{\mathtt{F}}|_{\overline{\mathtt{C}}}\mathtt{>m}(|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}})}$$

$$|\mathtt{new\ A}_0(\overline{\mathtt{e}})|_{\Delta,\Gamma,\mathtt{A}} = \mathtt{new}\ |\mathtt{A}_0|(|\overline{\mathtt{e}}|_{\Delta,\Gamma,\mathtt{A}})$$

Definition of Ceiling:

$$\lceil\mathtt{C.E}\rceil = \begin{cases} \mathtt{C\$E} & \text{if } \mathtt{class\ E\{..\}} \in \overline{\mathtt{NL}} \\ \lceil\mathtt{D.E}\rceil & \text{otherwise} \end{cases}$$

where $\mathtt{class\ C}\triangleleft\mathtt{D\{..\overline{NL}\}}$

Translation of Methods:

$$\frac{\mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}}}{|\mathtt{X}\triangleleft\mathtt{C}| = |\mathtt{X.E}_1|\triangleleft\lceil\mathtt{C.E}_1\rceil\mathtt{<}|\mathtt{X}.\overline{\mathtt{E}}|\mathtt{>}, \ldots, |\mathtt{X.E}_n|\triangleleft\lceil\mathtt{C.E}_n\rceil\mathtt{<}|\mathtt{X}.\overline{\mathtt{E}}|\mathtt{>}, \mathtt{X}\triangleleft\mathtt{C}}$$

$$\frac{\Gamma = \overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{this}:\mathit{thistype}(\mathtt{A}) \quad \Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{C}}}{|\mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}\mathtt{>T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\uparrow \mathtt{e}_0;\ \}|_{\mathtt{A}} = \mathtt{<}|\overline{\mathtt{X}}\triangleleft\overline{\mathtt{C}}|\mathtt{>}|\mathtt{T}_0|\ \mathtt{m}(|\overline{\mathtt{T}}|\ \overline{\mathtt{x}})\{\uparrow |\mathtt{e}_0|_{\Delta,\Gamma,\mathtt{A}};\}}$$

Translation of Classes:

$$\frac{\mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}}}{|\triangleleft\mathtt{C}| = |\mathtt{.E}_1|\triangleleft\lceil\mathtt{C.E}_1\rceil\mathtt{<}|.\overline{\mathtt{E}}|\mathtt{>}, \ldots, |\mathtt{.E}_n|\triangleleft\lceil\mathtt{C.E}_n\rceil\mathtt{<}|.\overline{\mathtt{E}}|\mathtt{>}}$$

$$\frac{\mathtt{class\ C}\triangleleft\mathtt{D\{..\}} \quad \mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}} \quad \mathit{nestedclasses}(\mathtt{D}) = \overline{\mathtt{E}}'}{|\mathtt{class\ E}_i\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}|_{\mathtt{C}} = \mathtt{class}\ i\ \mathtt{C\$E}_i\mathtt{<}|\triangleleft\mathtt{C}|\mathtt{>}\triangleleft\lceil\mathtt{D.E}_i\rceil\mathtt{<}|.\overline{\mathtt{E}}'|\mathtt{>\{}\ |\overline{\mathtt{T}}|\ \overline{\mathtt{f}};|\overline{\mathtt{M}}|_{\mathtt{C.E}}\mathtt{\}}}$$

$$\frac{\mathtt{class\ C}\triangleleft\mathtt{D\{..\}} \quad \mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}} \quad \mathit{nestedclasses}(\mathtt{D}) = \overline{\mathtt{E}}' \quad \mathtt{E} \notin \overline{\mathtt{E}}'}{|\mathtt{class\ E}_i\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}|_{\mathtt{C}} = \mathtt{class}\ i\ \mathtt{C\$E}_i\mathtt{<}|\triangleleft\mathtt{C}|\mathtt{>}\triangleleft\mathtt{Object\{}\ |\overline{\mathtt{T}}|\ \overline{\mathtt{f}};|\overline{\mathtt{M}}|_{\mathtt{C.E}}\mathtt{\}}}$$

$$\frac{\mathit{nestedclasses}(\mathtt{C}) = \overline{\mathtt{E}}}{\mathit{fix}(\mathtt{C},\mathtt{E}) = \mathtt{class}\ |\mathtt{C.E}|\triangleleft\lceil\mathtt{C.E}\rceil\mathtt{<}|\mathtt{C}.\overline{\mathtt{E}}|\mathtt{>\{\}}}$$

$$|\mathtt{class\ C}\triangleleft\mathtt{D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \overline{\mathtt{NL}}\mathtt{\}}| = \mathtt{class\ C}\triangleleft\mathtt{D}\ \{|\overline{\mathtt{T}}|\ \overline{\mathtt{f}};|\overline{\mathtt{M}}|_{\mathtt{C}}\}\ |\overline{\mathtt{NL}}|_{\mathtt{C}}\ \mathit{fix}(\mathtt{C}, \mathit{nestedclasses}(\mathtt{C}))$$

Class Types	**Method Types**
$C \triangleq \{c \mid c \text{ is declared in } CT\}$	$M \triangleq \{\boldsymbol{m} \mid \boldsymbol{m} \text{ is declared in } CT\} \cup \{\mathsf{get}[f] \mid f \in F\} \cup \{\mathsf{cast}[c] \mid c \in C\}$
$\tau^{c}(t_{\mathsf{obj}}) \triangleq \prod_{c' f \in F^{c}} t_{\mathsf{obj}}$	$\tau_{m}(t_{\mathsf{obj}}) \triangleq \tau'_{m}(t_{\mathsf{obj}})\ \mathtt{cmd}$
	where $\tau'_{\boldsymbol{m}}(t_{\mathsf{obj}}) \triangleq \tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}) \rightharpoonup (t_{\mathsf{obj}}\ \mathtt{cmd})$
	$\tau'_{\mathsf{get}[f]}(t_{\mathsf{obj}}) \triangleq t_{\mathsf{obj}}$
	$\tau'_{\mathsf{cast}[c']}(t_{\mathsf{obj}}) \triangleq t_{\mathsf{obj}}$
$\tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \triangleq \prod_{c \in C} (\tau^{c}(t_{\mathsf{obj}}) \rightharpoonup t_{\mathsf{obj}})$	$\tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \triangleq \prod_{m \in M} (t_{\mathsf{obj}} \rightharpoonup \tau_{m}(t_{\mathsf{obj}}))$

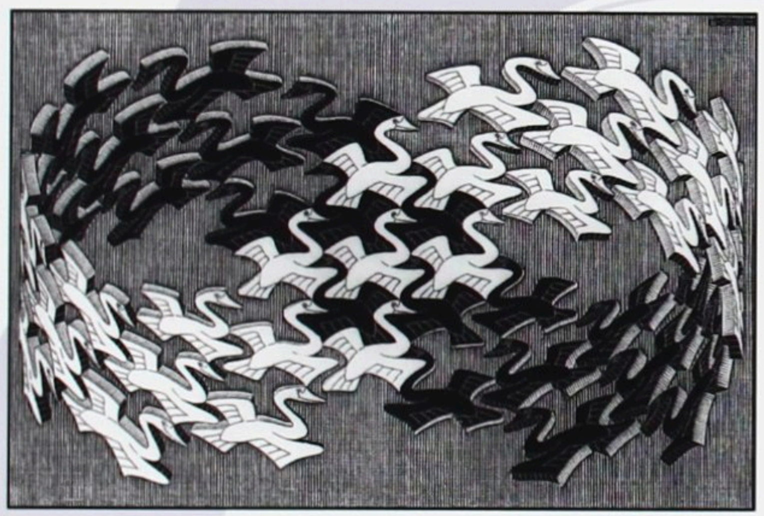
$$\boxed{\tau^{c}_{m} \triangleq \forall\, t_{\mathsf{obj}}\ .\ \tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \rightharpoonup \tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \rightharpoonup \tau^{c}(t_{\mathsf{obj}}) \rightharpoonup \tau_{m}(t_{\mathsf{obj}})}$$

Dahl and Nygaard at the time of Simula's development

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

SOFTWARE Development PRODUCTIVITY AWARD 1994

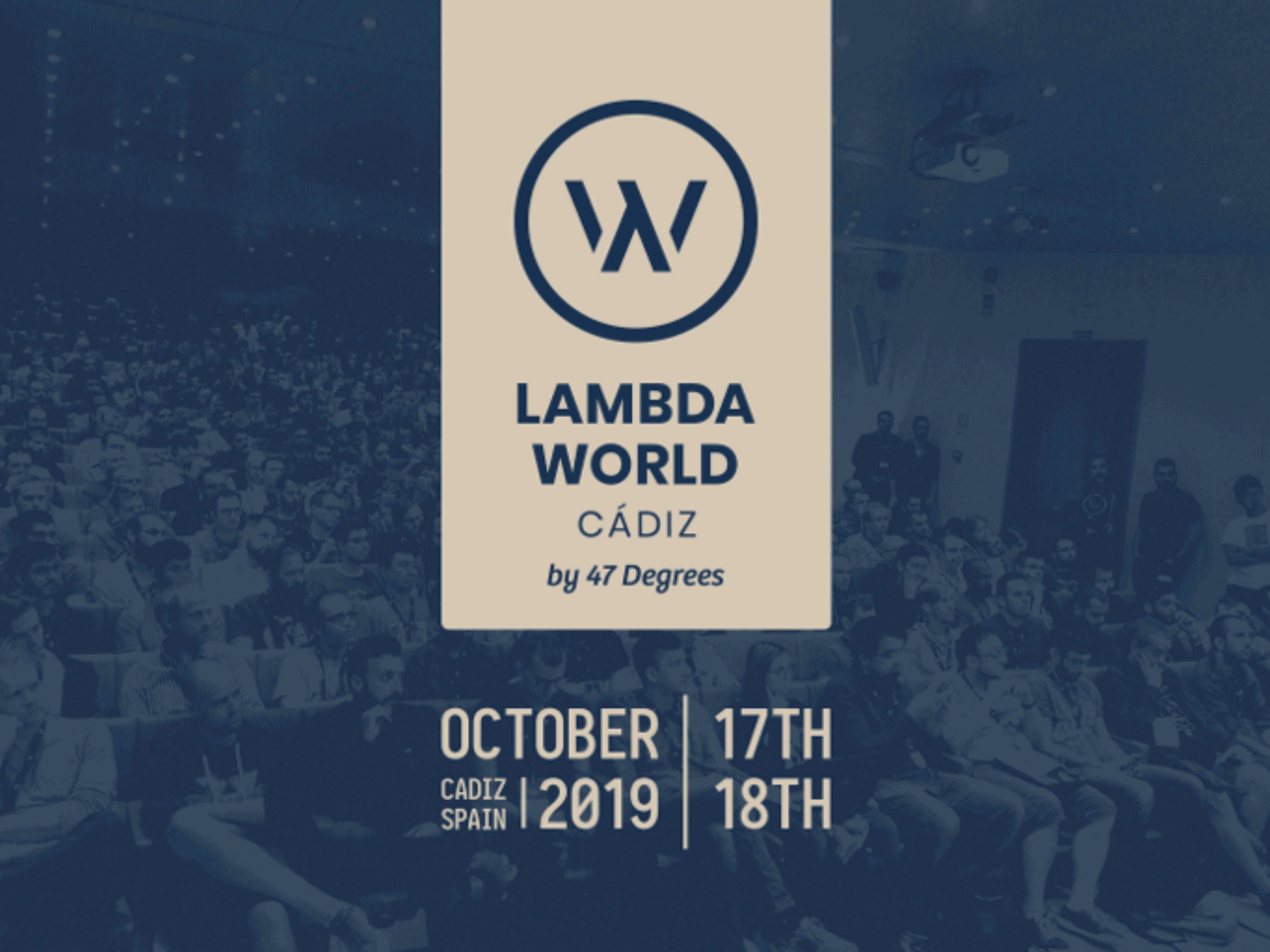
That leads us to our second principle of object-oriented design:

> *Favor object composition over class inheritance.*

Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition. But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.

Nevertheless, our experience is that designers overuse inheritance as a reuse technique, and designs are often made more reusable (and simpler) by depending more on object composition. You'll see object composition applied again and again in the design patterns.

User community


LAMBDA
WORLD
CÁDIZ
by 47 Degrees
OCTOBER
CADIZ
SPAIN
2019
17TH
18TH


lambda
DAλS
13–14 FEBRUARY 2020
KRAKÓW | POLAND

YOW!
Lambda Jam

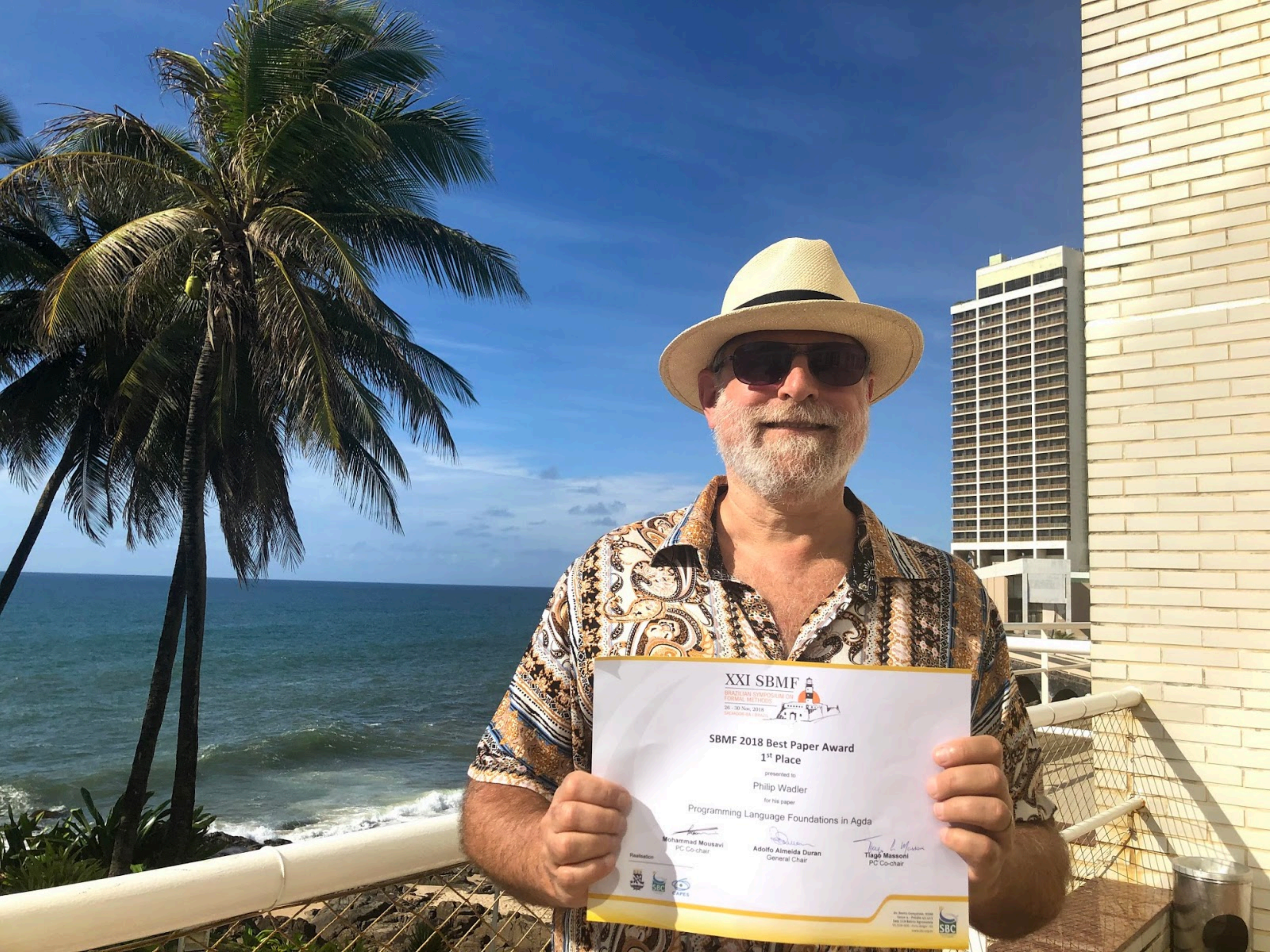

YOW!
LAMBDA JAM
YOW!

BUZZ

CONF

FUNCTIONAL PROGRAMMING
DISTRIBUTED SYSTEMS
MACHINE LEARNING
BIG DATA

Conclusions


XXI SBMF
BRAZILIAN SYMPOSIUM ON FORMAL METHODS
26 - 30 Nov, 2018
SBMF 2018 Best Paper Award
1st Place
presented to
Philip Wadler
for his paper
Programming Language Foundations in Agda
Mohammad Mousavi
PC Co-chair
Adolfo Almeida Duran
General Chair
Tiago Massoni
PC Co-chair
Realisation
SBC
CAPES
SBC

CONTRIBUTED ARTICLES

Propositions as Types

By Philip Wadler

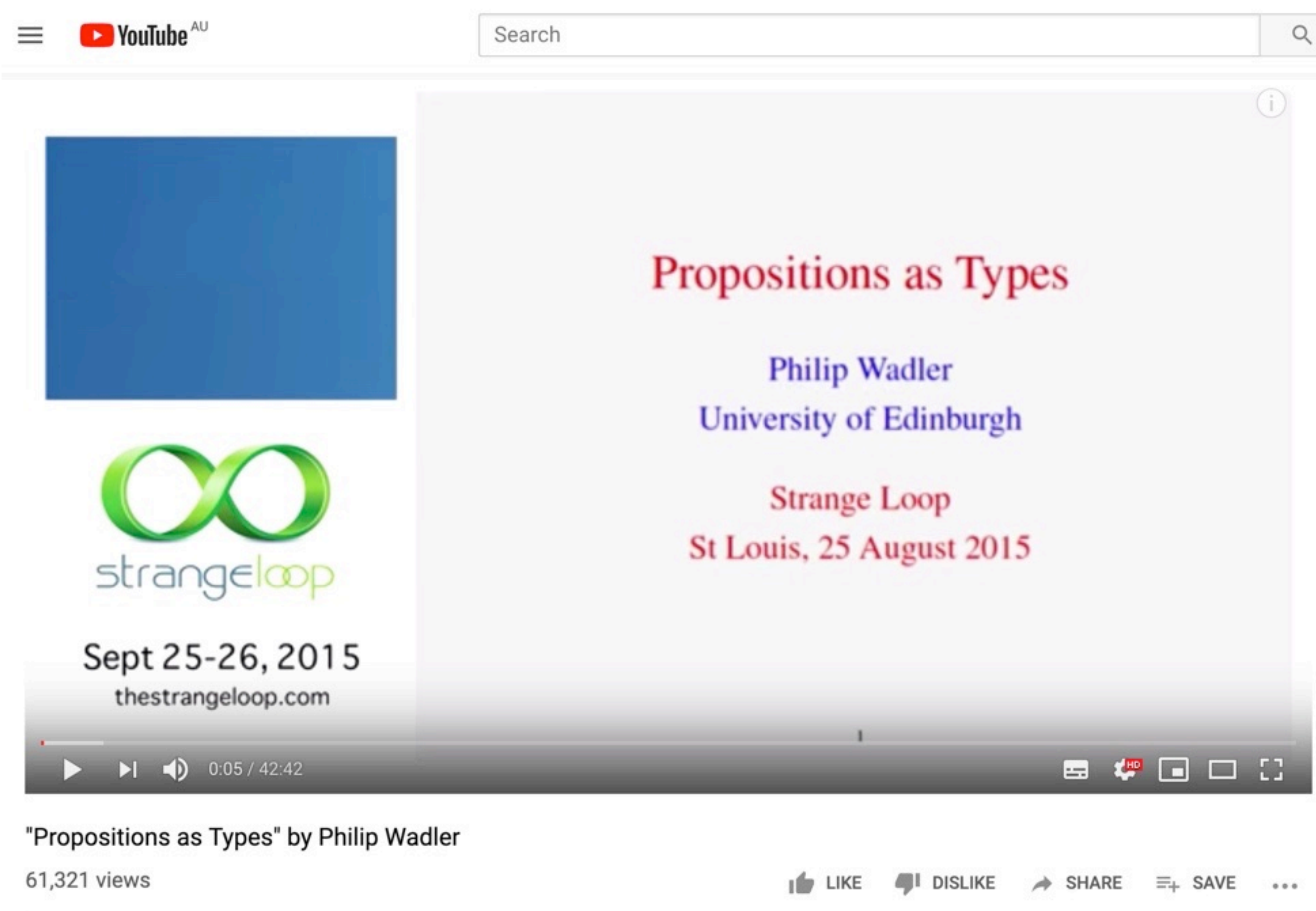
Communications of the ACM, December 2015, Vol. 58 No. 12, Pages 75-84
10.1145/2699407

Comments (1)

VIEW AS: | SHARE:

Powerful insights arise from linking two fields of study previously thought separate. Examples include Descartes's coordinates, which links geometry to algebra, Planck's Quantum Theory, which links particles to waves, and Shannon's Information Theory, which links thermodynamics to communication. Such a synthesis is offered by the principle of Propositions as Types, which links logic to computation. At first sight it appears to be a simple coincidence—almost a pun—but it turns out to be remarkably robust, inspiring the design of automated proof assistants and programming languages, and continuing to influence the forefronts of computing.

Back to Top


YouTube AU
Search
Propositions as Types
Philip Wadler
University of Edinburgh
Strange Loop
St Louis, 25 August 2015
strangeloop
Sept 25-26, 2015
thestrangeloop.com
0:05 / 42:42
"Propositions as Types" by Philip Wadler
61,321 views
LIKE
DISLIKE
SHARE
SAVE

vilem
@buggymcbugfix

Follow

I just proved commutativity of multiplication in Agda and got way too much serotonin out of it. 🤪

Programming Language Foundations in Agda is AMAZING. Check it out at plfa.github.io.

Thank you, Phil Wadler and @wenkokke.

(PS: If you have a better proof, let me know!)

```
*-comm : (m n : ℕ) → m * n ≡ n * m
*-comm zero n
  rewrite *-absorption n = refl
*-comm m zero
  rewrite *-absorption m = refl
*-comm (suc m') (suc n')                -- suc m' * suc n' ≡ suc n' * suc m'
  rewrite *-comm m' (suc n')            -- n' + (m' + n' * m') ≡ m' + n' * suc m'
        | sym (+-assoc n' m' (n' * m')) -- n' + m' + n' * m' ≡ m' + n' * suc m
        | *-comm n' m'                  -- n' + m' + m' * n' ≡ m' + n' * suc m'
        | +-comm n' m'                  -- m' + n' + m' * n' ≡ m' + n' * suc m'
        | *-comm n' (suc m')            -- m' + n' + m' * n' ≡ m' + (n' + m' * n')
        | +-assoc m' n' (m' * n')
        = refl
```

10:35 AM - 16 Oct 2018

14 Likes

http://plfa.inf.ed.ac.uk

https://github.com/plfa

Or search for “Kokke Wadler”

Please send your comments and pull requests!