Introduction
Farhad Mehta

Prof. Dr. Farhad Mehta
Department of Computer Science

OST
Eastern Switzerland
University of Applied Sciences

Who am I?

- First name: Farhad
- Last name: Mehta
- Born: Mumbai, India
- Grew up in Dubai
- Lived in Zurich, Delhi, Munich, Paris
- In Switzerland since 2004
- Married, 2 children

Background

Education	Dr. Sc. Computer Science (ETH Zurich 2008) M.Sc. Informatik (TU Munich 2004) B.Tech Computer Science & Engineering (IIT Delhi 2001)	
Experience	1997	Allied Enterprises, Dubai (IT Support)
	2000	DRDO, Bangalore (Research: IT Security)
	2001	INRIA, Paris (Research: Linguistics & Compilers)
	2002 - 2004	TU Munich (Teaching & Research: Logic & Software Engg.)
	2004 - 2008	ETH Zurich (Teaching & Research: Formal Methods & Software Engg.)
	2008 - 2014	Systransis AG (Development, Management, Marketing, ...)
	2013 - 2014	Part-time: Teacher at the Bildungszentrum Zürichsee (BZZ)
	Since Feb 2015	Professor for Computer Science at the OST
Interests	Software Engineering, Programming Languages, Functional Programming, Algorithms, Safety-critical Systems, Formal Methods, Logic. But also: Electronics, Usability, Didactics.	

I currently teach the following OST courses:
- SE Practices 1
- SE Project
- Functional Programming
- MSE EVA “Programming Languages”
- MSE Module “Advanced Prog. Paradigms”
- MAS SE Module “Functional Programming”
- CAS SW Testing Module “Unit Testing”

I have taught the following courses in the past:
- Software Engineering 1
- Software Engineering 2
- Engineering Project
- Programming Languages & Formal Methods
- Distributed Systems
- Compiler design
- Formal Methods and Functional Prog. (ETHZ)
- Informatik für nicht-Informatiker (ETHZ)
- Logik (ETHZ)
- …

I supervise:
- Semester Projects
- Bachelor Thesis Projects
- Master Semester Projects
- Master Thesis Projects

OST
Eastern Switzerland
University of Applied Sciences

FUNCTIONAL PROGRAMMING AND PROOF

Prof. Dr. Farhad Mehta
Department of Computer Science

Reasoning about Programs
Motivation

Till now, we have been able to:

- Formulate some interesting properties
- Test them using property-based testing (Quickcheck)

But testing can only show the presence of faults, never their absence.

How we can prove that our programs always satisfy some given properties?

We will now see how this is possible using techniques that you are already familiar with from high-school math!

Reasoning about Programs
Why is this important?

- Our luck at producing programs that work will run out ☺
- Formal Proof gives us the security to program in a way that is
  - Scalably Reliable
  - Scalably Efficient
- It forces us to keep our programs simple and elegant
- It makes it even possible to ‘derive’ correct programs from their properties

- There is a deep connection between proofs and programs:
  "PAT" Interpretation: Propositions As Types, Proofs As Terms

Reasoning about Programs
Why is this relevant to Functional Programming?

- Functional programs are particularly amenable to sound and simple reasoning
- This is their superpower, and is what makes them easier to work with for humans and machines

- Functional Programming and Formal Proof share a very rich legacy
- Remember: ML was originally the “Meta Language” for a theorem prover
- Functional Programming is often the gateway drug to other formal methods

Reasoning about Programs
Lesson Goals

All participants are able to

- State relevant correctness properties for functional programs (done)
- Provide counter-examples of program properties that do not hold (done)
- Perform formal proofs of program properties that hold using equational rewriting and induction

Proof Techniques
Equational Reasoning


We have already seen this many times in math, and since the start of this course.

```
 sum [1..5]
== { applying [..] }
 sum [1,2,3,4,5]
== { applying sum }
 1+2+3+4+5
== { applying + }
 15
```

```
∀x.qsort [x] == [x]

 qsort [x]
== {++applying qsort }
 qsort [] ++ [x] ++ qsort []
== { applying qsort }
 [] ++ [x] ++ []
== { applying ++ }
 [x]
```

```
totalWordCount :: [String] -> Int
totalWordCount =
  \strs -> foldr (+) 0 ( map length (map words strs))
== {Definition of (.), η conversion, map f . map g == map (f . g) }
  foldr (+) 0 . map (length . words)
== {applying "foldr f v . map g = foldr (f.g) v" }
  foldr ((+) . length . words) 0
```

This is and will remain the main workhorse for our proofs.

Note: Mathematitians are often cavalier and inconsiderate, and often overestimate their readers' patience. In this course we will be extra careful and explicit. Each step of a proof may only use a definition, or a property that we have already proven. The justification for each step needs to reflect this.

Proof Techniques

Equational Reasoning – A note on the form of proofs used in the text book

Deviation from Textbook (Programming in Haskell 2ed)

The textbook uses the '$=$' symbol in between lines of a derivation:

```
add :: Nat -> Nat -> Nat
add Zero     m = m
add (Succ n) m = Succ (add n m)
```

```
   add Zero (add y z)
=      { applying the outer add }
   add y z
=      { unapplying add }
   add (add Zero y) z
```

We will use the '==' symbol instead to be more consistent with the notation of equality used in Haskell, since '=' in Haskell is used for definitions. We will also be more explicit on which properties we use in each step of the proof:

add Zero m == m $(\mathtt{add}_{\mathtt{Zero}})$

```
 add Zero (add y z)
== { applying addZero }
 add y z
== { unapplying addZero }
 add (add Zero y) z
```

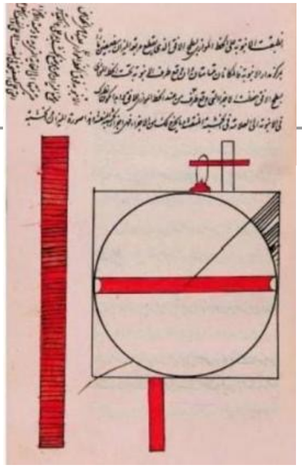
It is often easier to state and simplify both sides of the equality that we are trying to prove in each step, thereby avoiding awkward 'unapplying' steps. We will also underline the sub terms that get rewritten at each step for more clarity:

```
 add Zero (add y z) == add (add Zero y) z
== { applying addZero }
 add y z == add (add Zero y) z
== { applying addZero }
 add y z == add y z
== { (==)refl }
  True
```

Note: All undefined variables that occur in properties to be assumed or proven are by convention assumed to be universally qualified. For instance the statement `add Zero m == m` of $(\mathtt{add}_{\mathtt{Zero}})$ is actually `∀m.(add Zero m == m)`.

The example used on this slide is the proof of `add Zero (add y z) == add (add Zero y) z` using the definiton of **add** on page 233 of "Programming in Haskell 2ed" by Graham Hutton

Proof Techniques
Mathematical Induction



We additionally need induction to prove properties about recursively defined data structures.

This is the same principle of induction that you have learnt in high school and has been used since about 1000 AD.

Revision Exercise: Proove that the sum of the first n natural numbers is n(n+1)/2 using the technique of mathematical induction and equational reasoning as you have learnt in high school.

Notice: The idea behind induction is the same as the one behind recursion. One could even think that a proof by induction is nothing more than a recursive function that returns a proof! For any finite input, one could always "unroll" the induction to construct a proof without it, just like we can do for computation using recursion!

Proof Techniques
Mathematical Induction

Revision Exercise: Proove that the sum of the first n natural numbers is n(n+1)/2 using the technique of mathematical induction and equational reasoning as you have learnt in high school.

Proposition. For every $n \in \mathbb{N}$, $0+1+2+\cdots+n = \frac{n(n+1)}{2}$.

Proof. Let $P(n)$ be the statement $0+1+2+\cdots+n = \frac{n(n+1)}{2}$. We give a proof by induction on n.

Base case: Show that the statement holds for the smallest natural number $n = 0$.

$P(0)$ is clearly true: $0 = \frac{0(0+1)}{2}$.

Induction step: Show that for every $k \geq 0$, if $P(k)$ holds, then $P(k+1)$ also holds.

Assume the induction hypothesis that for a particular k, the single case $n = k$ holds, meaning $P(k)$ is true:

$$0+1+\cdots+k = \frac{k(k+1)}{2}.$$

It follows that:

$$(0+1+2+\cdots+k)+(k+1) = \frac{k(k+1)}{2}+(k+1).$$

Algebraically, the right hand side simplifies as:

$$\begin{aligned}\frac{k(k+1)}{2}+(k+1) &= \frac{k(k+1)+2(k+1)}{2}\\ &= \frac{(k+1)(k+2)}{2}\\ &= \frac{(k+1)((k+1)+1)}{2}.\end{aligned}$$

Equating the extreme left hand and right hand sides, we deduce that:

$$0+1+2+\cdots+k+(k+1) = \frac{(k+1)((k+1)+1)}{2}.$$

That is, the statement $P(k+1)$ also holds true, establishing the induction step.

Conclusion: Since both the base case and the induction step have been proved as true, by mathematical induction the statement $P(n)$ holds for every natural number n. Q.E.D.

https://en.wikipedia.org/wiki/Mathematical_induction

Proof Techniques
Mathematical Induction

The induction principle (a.k.a. induction rule) for natural numbers is typically expressed in term of the following logical inference rule (a.k.a. proof rule), where $P :: Nat \rightarrow Bool$ is any property that we want to prove.

$$\frac{\overbrace{P\ 0}^{\text{Base Case}} \qquad \overbrace{\forall n.\ (\ \overbrace{P\ n}^{\text{Induction Hypothesis}} \Rightarrow \overbrace{P\ (n+1)}^{\text{Induction Step}}\ )}^{\text{Inductive Case}}}{\forall n.\ P\ n}$$

Subgoals that need to be proven in order to prove the main goal

Main goal to be proven

In the case of the revision exercise:

$$P\ n = (\ (0+1+2+\ldots+n == n(n+1)/2\ )$$

Proof Techniques
Structural Induction - Lists

Natural numbers are not the only recursive structures that allow proof by induction. Every recusively defined structure admits an induction principle. This more general form of induction is sometimes known as structural induction.

For instance, here is the induction principle for lists in Haskell:

```
data [a] = [] | a:[a]
```

$$\frac{\overbrace{P\ []}^{\text{Base Case}} \qquad \overbrace{\forall x\ xs.(\overbrace{P\ xs}^{\text{Induction Hypothesis}} \Rightarrow \overbrace{P\ (x:xs)}^{\text{Induction Step}})}^{\text{Inductive Case}}}{\forall xs.\ P\ xs}$$

Subgoals that need to be proven in order to prove the main goal

Main goal to be proven

where `P xs :: [a] -> Bool` is any property on lists that we want to prove.

Notice: I have changed the font used on this slide to reflect that we are now no more in the realm of mathematics, but proving properties about Haskell programs.

Notice: I am mixing Haskell and mathematical syntax here (there is no ⇒ or ∀ in Haskell). This is definitely not kosher, but since I have no way to reason about Haskell programs within Haskell, I have no other choice. There are extensions to functional programming (for instance, Higher-Order Logic (HOL) & dependently typed languages such as Agda, Coq and Idris) that combine programming and proving. But since we are currently only interested in proofs on paper, we will let this slide and appeal to our (often imprecise) notion of proof from standard mathematics.

Note: "`∀x y. P`" is a short form for "`∀x.(∀y.P)`"

Proof Techniques
Structural Induction - Lists

$$\frac{P\ []\qquad \forall x\ xs.(P\ xs \Rightarrow P\ (x:xs))}{\forall xs.\ P\ xs}$$

Exercise:

Formally state and prove that the following property holds for all lists in Haskell:

"Concatenating two lists results in a list of length equal to the sum of the concatenated lists"

You are only allowed use the following properties, as well as the fact that lists are recursively defined data types in your proof:

	`length [] == 0`	($length_{[]}$)
∀x xs.	`length (x:xs) == 1 + length xs`	($length_{(:)}$)
∀xs.	`[] ++ xs == xs`	($(++)_{[]}$)
∀x xs ys.	`(x:xs) ++ ys == x:(xs++ys)`	($(++)_{(:)}$)
∀n.	`0 + n == n`	($(+)_0$)
∀a b c.	`(a + b) + c == a + (b + c)`	($(+)_{assoc}$)
∀x.	`(x == x) == True`	($(==)_{refl}$)

Hint: Start with a proof by induction on the first argument of `(++)`, since `(++)` is defined using recursion on its first argument.

`P xs = ∀ys. ( length (xs ++ ys) == length xs + length ys )`

Proof Techniques
Structural Induction - Lists

```
            length [] == 0.                  (length[])
∀x xs.      length (x:xs) == 1 + length xs.  (length(:))
∀xs.        [] ++ xs == xs                   ((++)[])
∀x xs ys.   (x:xs) ++ ys == x:(xs++ys).      ((++)(:))
∀n.         0 + n == n                       ((+)0)
∀a b c.     (a + b) + c == a + (b + c)       ((+)assoc)
∀x.         (x == x) == True.                ((==)refl)
```

$$\frac{\texttt{P [ ]} \quad \forall \texttt{x xs.(P xs} \Rightarrow \texttt{P (x:xs))}}{\forall \texttt{xs. P xs}}$$

Exercise: Formally state and prove that the following property holds for all lists in Haskell:
"Concatenating two lists results in a list of length equal to the sum of the concatenated lists"

Solution:

Required to Prove (RTP): `∀xs ys. ( length (xs ++ ys) == length xs + length ys )`

Proof. Proceed by induction on xs: Let `P xs = ∀ys. (length (xs ++ ys) == length xs + length ys)` and apply the induction rule for lists on `P xs`.

1. **Base Case.** RTP: `P []`

```
 P []
== {applying definiton of P, choosing a fixed but arbitrary ys}
 length ([] ++ ys) == length [] + length ys
== {applying length[]}
 length ([] ++ ys) == 0 + length ys
== {applying (++)[]}
 length ys == 0 + length ys
== {applying (+)0}
 length ys == length ys
== {applying (==)refl}
 True
```

Proof Techniques
Structural Induction - Lists

```
          length [] == 0.                      (length[])
∀x xs.    length (x:xs) == 1 + length xs.      (length(:))
∀xs.      [] ++ xs == xs                       ((++)[])
∀x xs ys. (x:xs) ++ ys == x:(xs++ys).          ((++)(:))
∀n.       0 + n == n                           ((+)0)
∀a b c.   (a + b) + c == a + (b + c)           ((+)assoc)
∀x.       (x == x) == True.                    ((==)refl)
```

```
P []    ∀x xs.(P xs ⇒ P (x:xs))
─────────────────────────────
        ∀xs. P xs
```

`P xs = ∀ys. (length (xs ++ ys) == length xs + length ys)`

Solution (continued):

2. Induction Step. RTP: `∀x xs.(P xs ⇒ P (x:xs))`

Choose a fixed but arbitrary `x` and `xs`, and assume that following the induction hypothesis `P xs` holds

```
∀ys. (length (xs ++ ys) == length xs + length ys)          (Induction Hypothesis)
```

```
 P (x:xs)
== {applying definiton of P, choosing a fixed but arbitrary ys}
 length ((x:xs) ++ ys) == length (x:xs) + length ys
== {applying length(:)}
 length ((x:xs) ++ ys) == (1 + length xs) + length ys
== {applying ((++)(:))}
 length (x:(xs ++ ys)) == (1 + length xs) + length ys
== {applying length(:)}
 1 + length (xs ++ ys) == (1 + length xs) + length ys
== {applying Induction Hypothesis}
 1 + (length xs + length ys) == (1 + length xs) + length ys
== {applying (+)assoc}
 1 + (length xs + length ys) == 1 + (length xs + length ys)
== {applying (==)refl}
 True
```

Note: This sample proof demonstrates the format and the formal rigour I expect to see in your exercise solutions and in the exam.

Proof Techniques
Structural Induction - Trees

To illustrate that this can be done systematically for any algebraic data structure, here is the induction principle for binary trees in Haskell:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

$$\frac{\overbrace{P\ \mathtt{Leaf}}^{\text{Base Case}} \qquad \overbrace{\forall x\ tl\ tr.(\overbrace{P\ tl \land P\ tr}^{\text{Induction Hypothesis}} \Rightarrow \overbrace{P\ (\mathtt{Node}\ tl\ x\ tr)}^{\text{Induction Step}})}^{\text{Inductive Case}}}{\forall t.\ P\ t}$$

Subgoals that need to be proven in order to prove the main goal

Main goal to be proven

where `P t :: Tree a -> Bool` is any property on trees that we want to prove.

Note: The symbol ∧ denotes logical "and" (a.k.a. conjunction), and binds tighter than ⇒, which denotes logical implication. The universal quantifier ∀ binds the weakest.

Excursion

Proof Rules & Proof Trees

Induction is not the only concept that can be precisely expressed in terms of proof rules.

Proof rules can also be used to:

- Precisely specify the meaning and use of logical connectives (e.g., $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$)
- Thereby perform other forms of proof such as proof by contradiction, case distinction, …
- Perform proofs involving equational reasoning
- Construct entire proofs (i.e., proof trees) by combining individual proof rules

In standard mathematics, proofs are normally communicated informally as prosa text.

In formal mathematics, proof rules are used to formally specify the structure of a proof upto its finest details.

This makes it possible for a computer to help construct and check the correctness of a proof!

Excursion
Proof Rules & Proof Trees

- Here is an example of what a complete set of proof rules for first-order logic with equality looks like.
- Proofs are just trees constructed using these proof rules.
- Using such rules, one could implement a data type that can only contain valid theorems.
- This is exactly the approach that automated proof assistants use, and is the original motivation behind parametric polymorphism (a.k.a. generics).
- One almost never constructs proof trees by hand.
- There are several automated proof assistants to choose from: Isabelle/HOL, Coq, Adga, Idris, Lean, F*, ACL2, PVS, HOL4, …

$$\dfrac{\dfrac{}{R \vdash R}\,hyp \qquad \dfrac{}{R, C \vdash C}\,hyp}{R \Rightarrow C, R \vdash C}\,{\Rightarrow}hyp$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.H(x) \Rightarrow M(x), H(s) \vdash H(s)}\,hyp \qquad \dfrac{}{\forall x.H(x) \Rightarrow M(x), M(s), H(s) \vdash M(s)}\,hyp}{\forall x.H(x) \Rightarrow M(x), H(s) \Rightarrow M(s), H(s) \vdash M(s)}\,{\Rightarrow}hyp}{\forall x.H(x) \Rightarrow M(x), [x := s]H(x) \Rightarrow M(x), H(s) \vdash M(s)}\,(\widehat{=}_{[:=]})*}{\forall x.H(x) \Rightarrow M(x), H(s) \vdash M(s)}\,\forall hyp$$

Theory $FoPCe$

$$\frac{}{\mathsf{H}, P \vdash P}\,hyp \qquad \frac{\mathsf{H} \vdash Q}{\mathsf{H}, P \vdash Q}\,mon \qquad \frac{\mathsf{H} \vdash P \quad \mathsf{H}, P \vdash Q}{\mathsf{H} \vdash Q}\,cut$$

$$\frac{}{\mathsf{H}, \bot \vdash P}\,\bot hyp \qquad \frac{}{\mathsf{H} \vdash \top}\,\top goal \qquad \frac{\mathsf{H}, \neg P \vdash \bot}{\mathsf{H} \vdash P}\,contr$$

$$\frac{\mathsf{H}, P \vdash \bot}{\mathsf{H} \vdash \neg P}\,\neg goal \qquad \frac{\mathsf{H} \vdash P}{\mathsf{H}, \neg P \vdash Q}\,\neg hyp$$

$$\frac{\mathsf{H} \vdash P \quad \mathsf{H} \vdash Q}{\mathsf{H} \vdash P \wedge Q}\,\wedge goal \qquad \frac{\mathsf{H}, P, Q \vdash R}{\mathsf{H}, P \wedge Q \vdash R}\,\wedge hyp$$

$$\frac{\mathsf{H} \vdash P}{\mathsf{H} \vdash P \vee Q}\,\vee goal1 \qquad \frac{\mathsf{H} \vdash Q}{\mathsf{H} \vdash P \vee Q}\,\vee goal2 \qquad \frac{\mathsf{H}, P \vdash R \quad \mathsf{H}, Q \vdash R}{\mathsf{H}, P \vee Q \vdash R}\,\vee hyp$$

$$\frac{\mathsf{H}, P \vdash Q}{\mathsf{H} \vdash P \Rightarrow Q}\,{\Rightarrow}goal \qquad \frac{\mathsf{H} \vdash P \quad \mathsf{H}, Q \vdash R}{\mathsf{H}, P \Rightarrow Q \vdash R}\,{\Rightarrow}hyp$$

$$\frac{\mathsf{H} \vdash P \Rightarrow Q \quad \mathsf{H} \vdash Q \Rightarrow P}{\mathsf{H} \vdash P \Leftrightarrow Q}\,{\Leftrightarrow}goal \qquad \frac{\mathsf{H}, P \Rightarrow Q, Q \Rightarrow P \vdash R}{\mathsf{H}, P \Leftrightarrow Q \vdash R}\,{\Leftrightarrow}hyp$$

$$\frac{\mathsf{H} \vdash P}{\mathsf{H} \vdash \forall x.P}\,\forall goal\ (x\ \widehat{\mathsf{nfin}}\ \mathsf{H}) \qquad \frac{\mathsf{H}, \forall x.P, [x := E]P \vdash Q}{\mathsf{H}, \forall x.P \vdash Q}\,\forall hyp$$

$$\frac{\mathsf{H} \vdash [x := E]P}{\mathsf{H} \vdash \exists x.P}\,\exists goal \qquad \frac{\mathsf{H}, P \vdash Q}{\mathsf{H}, \exists x.P \vdash Q}\,\exists hyp\ (x\ \widehat{\mathsf{nfin}}\ \mathsf{H} \cup \{Q\})$$

$$\frac{}{\mathsf{H} \vdash E = E}\,{=}goal \qquad \frac{\mathsf{H}, E = F \vdash [x := F]P}{\mathsf{H}, E = F \vdash [x := E]P}\,{=}hyp$$

Excursion

Proof Rules & Proof Trees

- In computer science, proof rules are used to formally specify the type systems used in programming languages.
- Here is an example of what a complete set of proof rules for the simply typed lambda calculus, and the polymorphic lambda calculus look like.
- The proof rules for type systems have a striking simmilarity to those of mathematical logic.
- This led to the discovery of a deep connection between computation and proof, known as the Curry-Howard Correspondence, a.k.a. the PAT interpretation, which is used in systems such as Agda, Coq and Idris.
- "PAT" Interpretation: Propositions As Types, Proofs As Terms

Theory $\lambda{\to}$

$$\frac{}{\Gamma, x{:}\sigma \vdash x : \sigma}\;\textit{var}$$

$$\frac{\Gamma \vdash M : \sigma{\to}\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\;\textit{app}_{\text{term}} \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma{\to}\tau}\;\textit{abs}_{\text{term}}$$

Theory $\lambda 2$

$$\frac{}{\Gamma, x{:}\sigma \vdash x : \sigma}\;\textit{var}$$

$$\frac{\Gamma \vdash M : \sigma{\to}\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\;\textit{app}_{\text{term}} \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma{\to}\tau}\;\textit{abs}_{\text{term}}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]}\;\textit{app}_{\text{type}} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma}\;\textit{abs}_{\text{type}}\;(*)$$

(*) The last rule only applies if the type variable α does not occur free in any type in Γ.

Theory *FoPCe*

$$\frac{}{\mathsf{H}, P \vdash P}\;\textit{hyp}$$

$$\frac{\mathsf{H} \vdash P \qquad \mathsf{H}, Q \vdash R}{\mathsf{H}, P \Rightarrow Q \vdash R}\;{\Rightarrow}\textit{hyp} \qquad \frac{\mathsf{H}, P \vdash Q}{\mathsf{H} \vdash P \Rightarrow Q}\;{\Rightarrow}\textit{goal}$$

$$\frac{\mathsf{H}, \forall x.P, [x := E]P \vdash Q}{\mathsf{H}, \forall x.P \vdash Q}\;\forall\textit{hyp} \qquad \frac{\mathsf{H} \vdash P}{\mathsf{H} \vdash \forall x.P}\;\forall\textit{goal}\;(x\ \widehat{\underline{\mathsf{nfin}}}\ \mathsf{H})$$

Reminder

Key Idea of FP: Denotative Language / Semantics

Central passages:

“The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack. In particular, (a) each expression has a nesting subexpression structure, (b) each subexpression denotes something (usually a number, truth value or numerical function), (c) the thing an expression denotes, i.e., its "value", depends only on the values of its sub-expressions, not on other properties of them.”

“The word "denotative" seems more appropriate than non-procedural, declarative or functional. The antithesis of denotative is "imperative”.”

“functional programming has little to do with functional notation.”

“The question arises, do the idiosyncracies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments?”

“we must think in terms, not of languages, but of families of languages. That is to say we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction.”

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

This attempt has led the author to think that many linguistic idiosyncracies are concerned with the former rather than the latter, whereas aptitude for a particular class of tasks is essentially determined by the latter rather than the former. The conclusion follows that many language characteristics are irrelevant to the alleged problem orientation.

ISWIM is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of "primitives." So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what is needed to specify such a set are discussed below.

ISWIM is not alone in being a family, even after mere syntactic variations have been discounted (see Section 4). In practice, this is true of most languages that achieve more than one implementation, and if the dialects are well disciplined, they might with luck be characterized as differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g., compare

$x(x+a)$
where $x = b + 2c$

$f(b+2c)$
where $f(x) = x(x+a)$

ISWIM is thus part programming language and part program for research. A possible first step in the research program is 1700 doctoral theses called "A Correspondence between x and Church's λ-notation."[1]

2. The where-Notation

In ordinary mathematical communication, these uses of '**where**' require no explanation. Nor do the following:

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$

$f(b+2c) + f(2b-c)$
where $f(x) = x(x+a)$
and $b = u/(u+1)$
and $c = v/(v+1)$

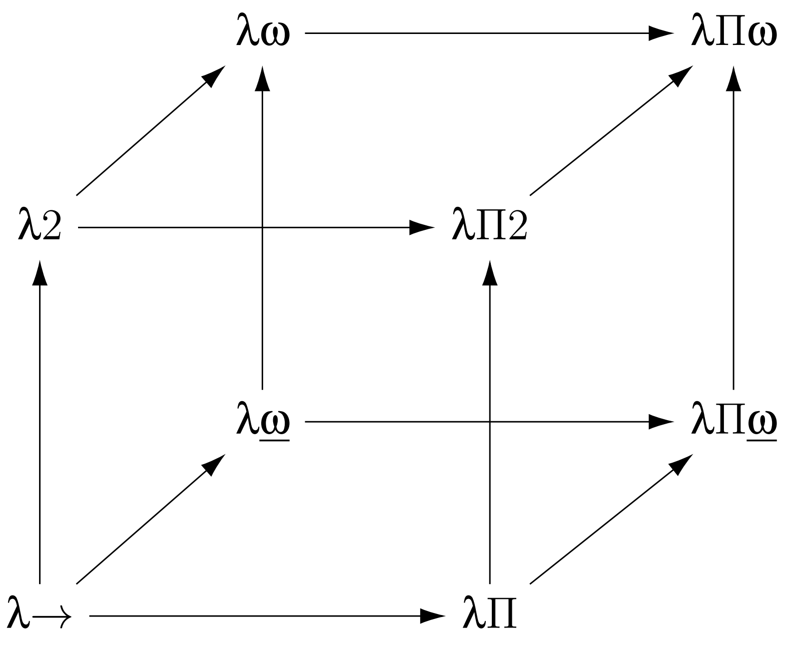
$g(f$ **where** $f(x) = ax^2 + bx + c,$
$u/(u+1),$
$v/(v+1))$
where $g(f, p, q) = f(p+2q, 2p-q)$

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

[1] There is no more use or mention of λ in this paper—cognoscenti will nevertheless sense an undercurrent. A not inappropriate title would have been "Church without lambda."

Excursion

The Lambda Cube



[Bar91] Hendrik Pieter Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

Note: These are from personal notes that I have not checked myself. Do not quote me on this.

Orign:
λ→ : Simply typed lambda calculus
Terms may only depend on Terms
Curry-Howard correspondence for λ→: Propositional calculus restricted to only use implication.

Going up (2):
λ2 : System F, second-order lambda calculus
Terms may depend on Types
(polymorphism, e.g. (Church-style) λα:*.λx:α.x : ∀α.α→α , or (Curry-style) λx.x:∀α.α→α)
Curry-Howard correspondence for λ2: fragment of second-order intuitionistic logic that uses only universal quantification.

Going inwards (ω):
Types may depend on Types
(type operators, e.g. "List α" is a type, where List is a type operator with kind * → *)
Not very interesting in isolation.
Normally combined with λ2 (System F) to give λω (System Fω) (a variant of this (System FC) is used in Haskell)
Curry-Howard correspondence for λω (System Fω): Higher-Order Logic

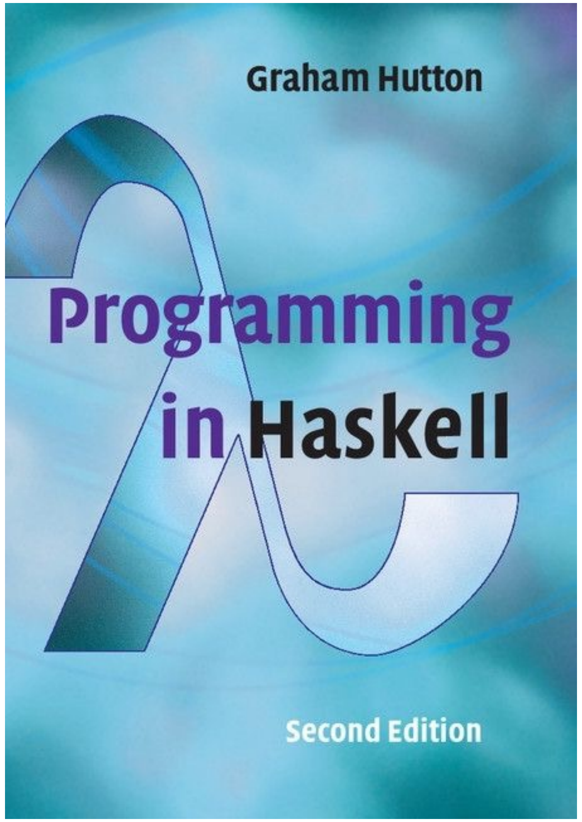
Going rightwards (Π, or P):
Types may depend on values
(dependent types, e.g. "FloatList 3" is a type denoting a list of floats with length 3, where Floatlist : Nat→*)
λΠ : also called λP, LF
Curry-Howard correspondence for λΠ: A form of predicate calculus that only uses implication and universal quantification.

Richest calculus of all 8:
λΠω : Calculus of Constructions (CC, CoC, λC)

Reasoning about Programs

Going further

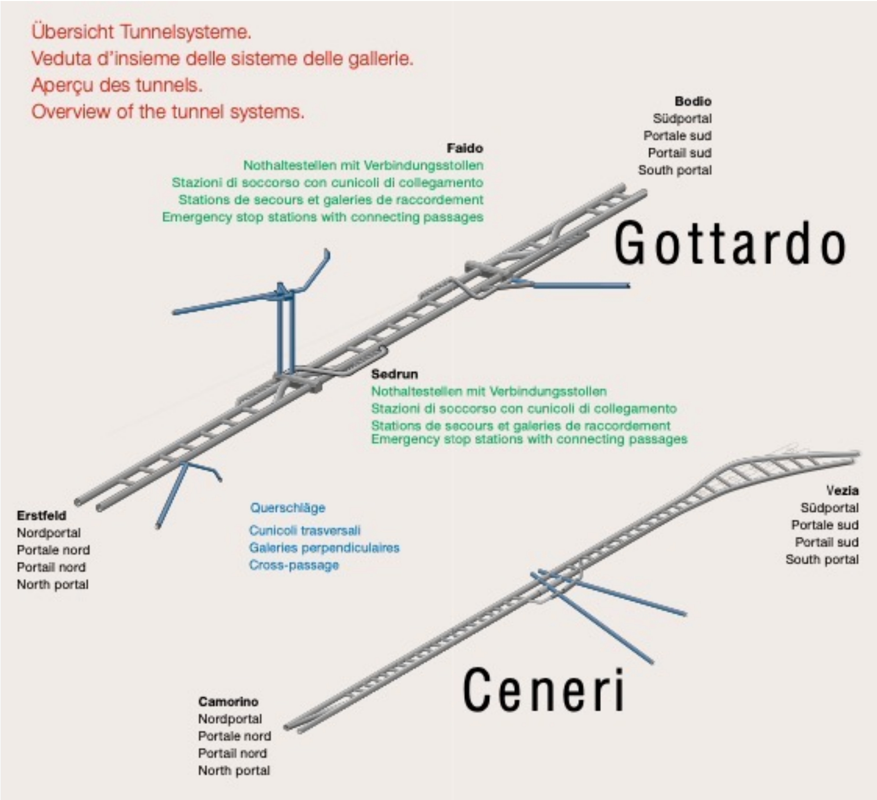
- Chapter 16 of the textbook contains some more examples of proofs, as well as a section on proving the correctness of a compiler.
- Chapter 17 of the textbook goes even further by showing how the implementation of a compiler can be calculated directly from the statement of its correctness.
- Try using an automated proof assistant such as Isabelle/HOL, or a dependently typed programming language such as Agda, Idris, or Coq.
- Try to build your own proof assistant in Haskell!



Selected Work

Gotthard Base Tunnel
Safety-relevant Functions “Freihaltung”, “Überfullverhinderung”

Gotthard Base Tunnel
Safety-relevant Functions “Freihaltung”, “Überfullverhinderung”



Facts and figures on the New Rail Link through the Alps (NEAT).

- At 57 kilometres, the GBT is the longest railway tunnel in the world. The CBT measures 15,4 km.
- It took 17 years to build the GBT, the CBT was built in 12 years.
- The GBT cost a grand total of CHF 12.2 billion, the CBT cost 3.5 billion.
- It takes just under 20 minutes to travel through the GBT on a passenger train.
- The GBT can handle up to 260 freight and 65 passenger trains per day.
- Temperatures inside the GBT can reach 35 degrees Celsius.
- Freight trains travel through the tunnel at 100 km/h and passenger trains at up to 230 km/h.
- A fire-fighting and rescue train is on hand near the north and south portals of the tunnels – ready for service round the clock for your safety.
- With the Ceneri Base Tunnel completed as well, journey times Zurich–Lugano will be cut by 45 minutes.

Inspiration from:

- Refinement calculus, Invariant preservation
- Inductively defined sets

Gotthard Base Tunnel
Safety-relevant Functions “Freihaltung”, “Überfullverhinderung”

Are we being too paranoid?
Will these functions ever be needed?

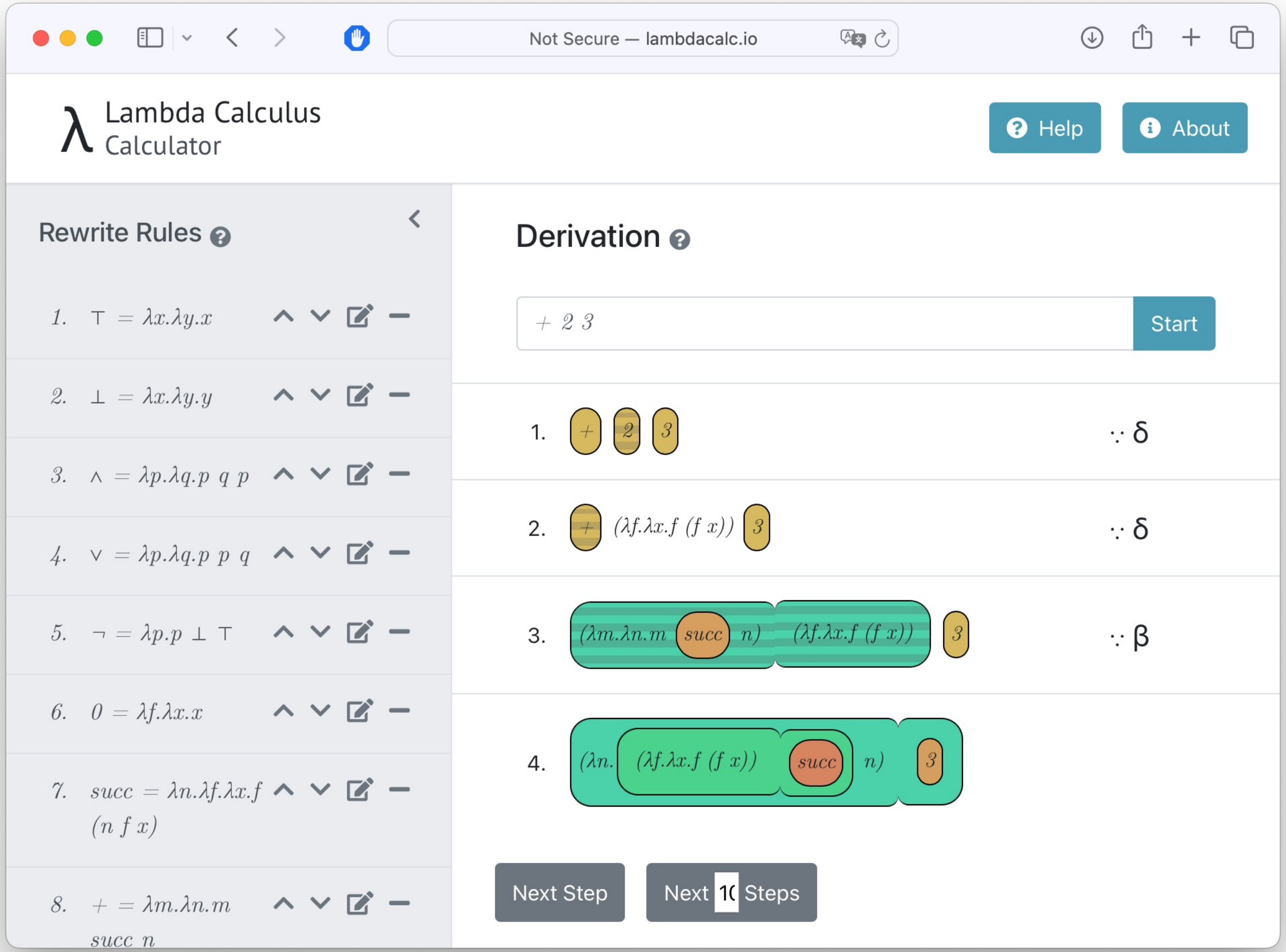

Blick | DE |
TESSIN
Brand im Gotthard-Basistunnel
Zehn A
Rauch
Die Güterzüge im G
verkehren. Am Mor
gesperrt worden.
Publiziert: 29.01.2024 um 12:03

Blick | DE | FR
SCHWEIZ
Schweiz | Gotthard-Zugentgleisung könnte 100 Millionen kosten
Experten schätzen Schadenssumme ein
Zugentgleisung im Gotthard kostet über 100 Millionen!
Die Weströhre des Gotthard-Basistunnels ist noch immer geschlossen. Seit Mitte August ein Güterzug entgleiste, ist der Zugverkehr stark eingeschränkt. Experten schätzen, dass die Schadenssumme im dreistelligen Millionenbereich liegt.
Publiziert: 26.10.2023 um 15:49 Uhr | Aktualisiert: 26.10.2023 um 16:04 Uhr

Lambda Calculus Calculator

https://lambdacalc.io

Lambda Calculus Calculator

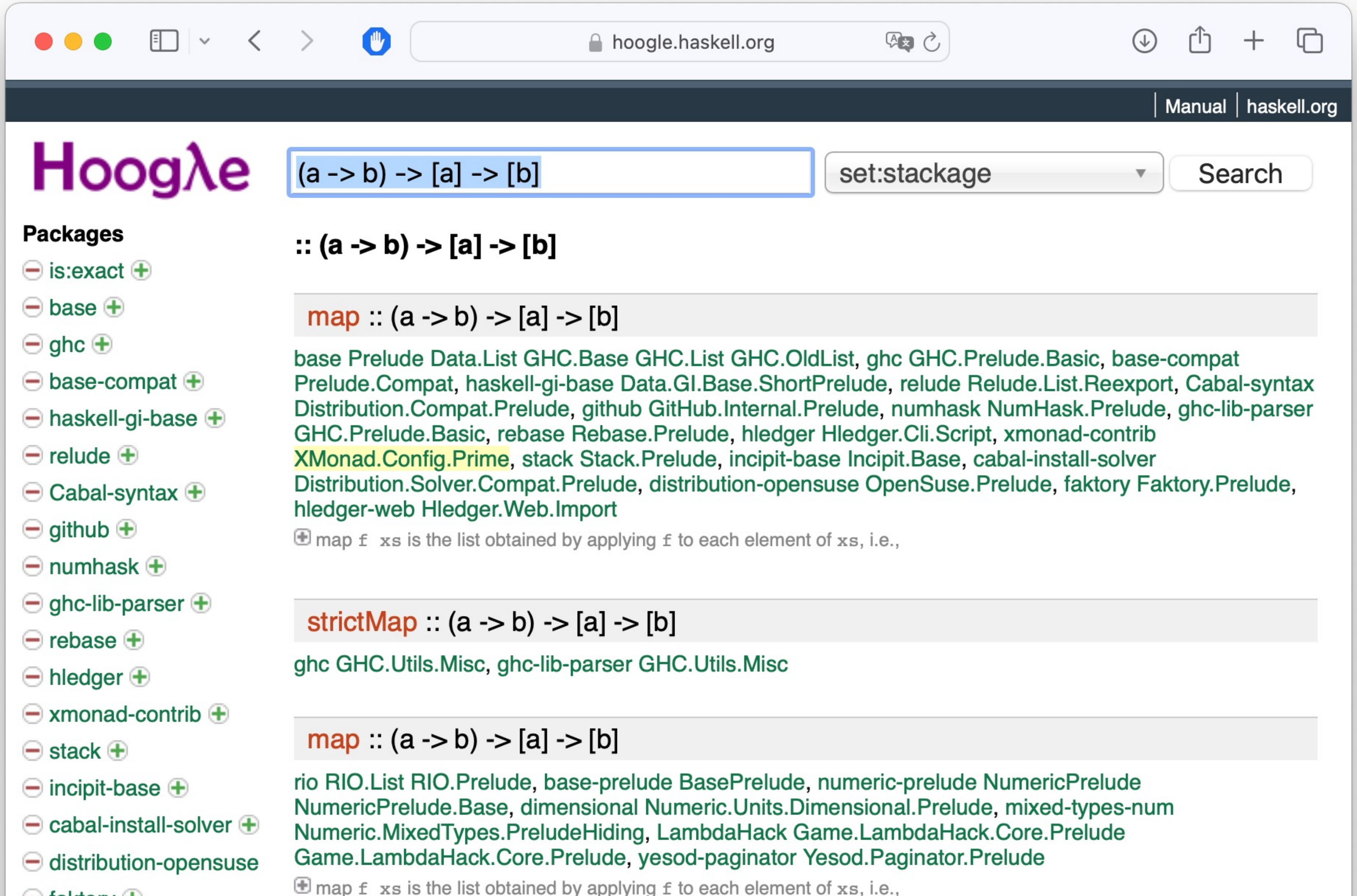


Type-Based API Search

https://typesearch.dev

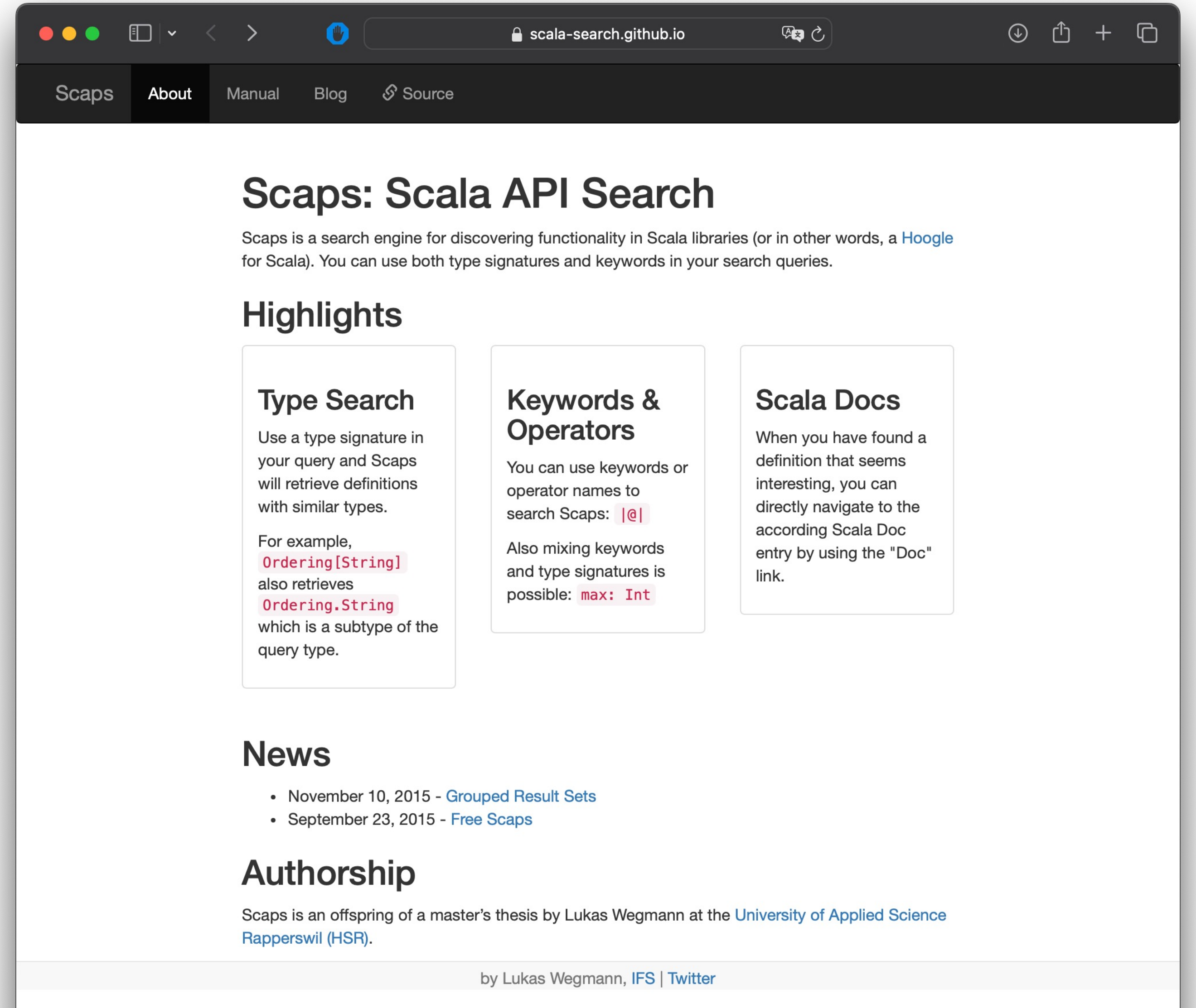
Hoogle for the hungry masses

Type-based API Search for All – typesearch.dev



Hoogle for the hungry masses

Type-based API Search for All – typesearch.dev



Scaps: Type-Directed API Search for Scala

Lukas Wegmann
1plusX AG, Switzerland
wegmaluk@gmail.com

Farhad Mehta Peter Sommerlad
Mirko Stocker
Institute for Software, University of Applied Sciences Rapperswil, Switzerland
{first}.{last}@hsr.ch

Abstract

Type-directed API search, using queries composed of both keywords and type signatures to retrieve definitions from APIs, are popular in the functional programming community. This search technique allows programmers to easily navigate complex and large APIs in order to find the definitions they are interested in. While there exist some effective approaches to address type-directed API search for functional languages, we observed that none of these have been successfully adapted for use with statically-typed, object-oriented languages. The challenge here is incorporating large and unified inheritance hierarchies and the resulting prevalence of subtyping into an API retrieval model. We describe a new approach to API retrieval and provide an implementation thereof for the Scala language. Our evaluation with queries mined from Q&A websites shows that the model retrieves definitions from the Scala standard library with 94% of the relevant results in the top 10.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques

Keywords API Search and Retrieval, Polarized Types

1. Introduction

A crucial part of creating high quality software is the reuse of existing functionality provided by in-house or third-party programming libraries. This ensures that functionality is not unnecessarily reimplemented and lowers the risk of introducing erroneous behavior. Code reused over various projects has a greater chance of being reliable since it has been well-tested in production.

Discovering existing functionality is a task that requires either deep knowledge of the relevant libraries, or appropriate tools that provide convenient access to the definitions in a library. Popular examples of such tools are code completion assistants that list the accessible members of the object in question. Although, there are several reasons why code completion is not always able to provide developers with a complete picture of suitable functionality: First, the structure of an API greatly influences the discoverability of its functionality. Indirections like factories, utility classes, extension methods and implicit conversions often hinders developers from quickly discovering library features [4, 16]. Second, programming in the functional style results in numerous abstractions of transformations over data structures. While it would be favorable to provide as much of these abstractions as possible as library functions, it becomes more difficult for users to quickly discover important features. And finally, varying naming schemes amongst programming environments further complicates API discovery. Developers used to names like `filter` and `mkString` in one environment will likely have some troubles when switching to an environment that uses `where` and `join` for the same operations. To overcome these problems, developers often resort to universal search engines like Google to find a specific implementation. While these search engines regularly provide good results, users have to scan the result pages for suitable content. Additionally, general search engines may retrieve outdated information referring to an older revision of a library.

In order to alleviate these problems search engines, like Hoogle for Haskell [8], allow searching for values based on their type signature. Hoogle retrieves definitions related to the query type ordered by their relevance to the query. This assumes that developers usually know what types they have and of what type the result should be, but do not know how to get there. The great number of questions of the form "How to create X from Y" on popular Q&A websites supports this assumption.

While the idea to use types to direct API searches is not a new one [13], there are almost no applications of this idea outside the functional programming community, even though such a tool would definitely be useful for object-oriented languages, like Scala, that leverage a high level of type safety. However, while attempting to adopt Hoogle to

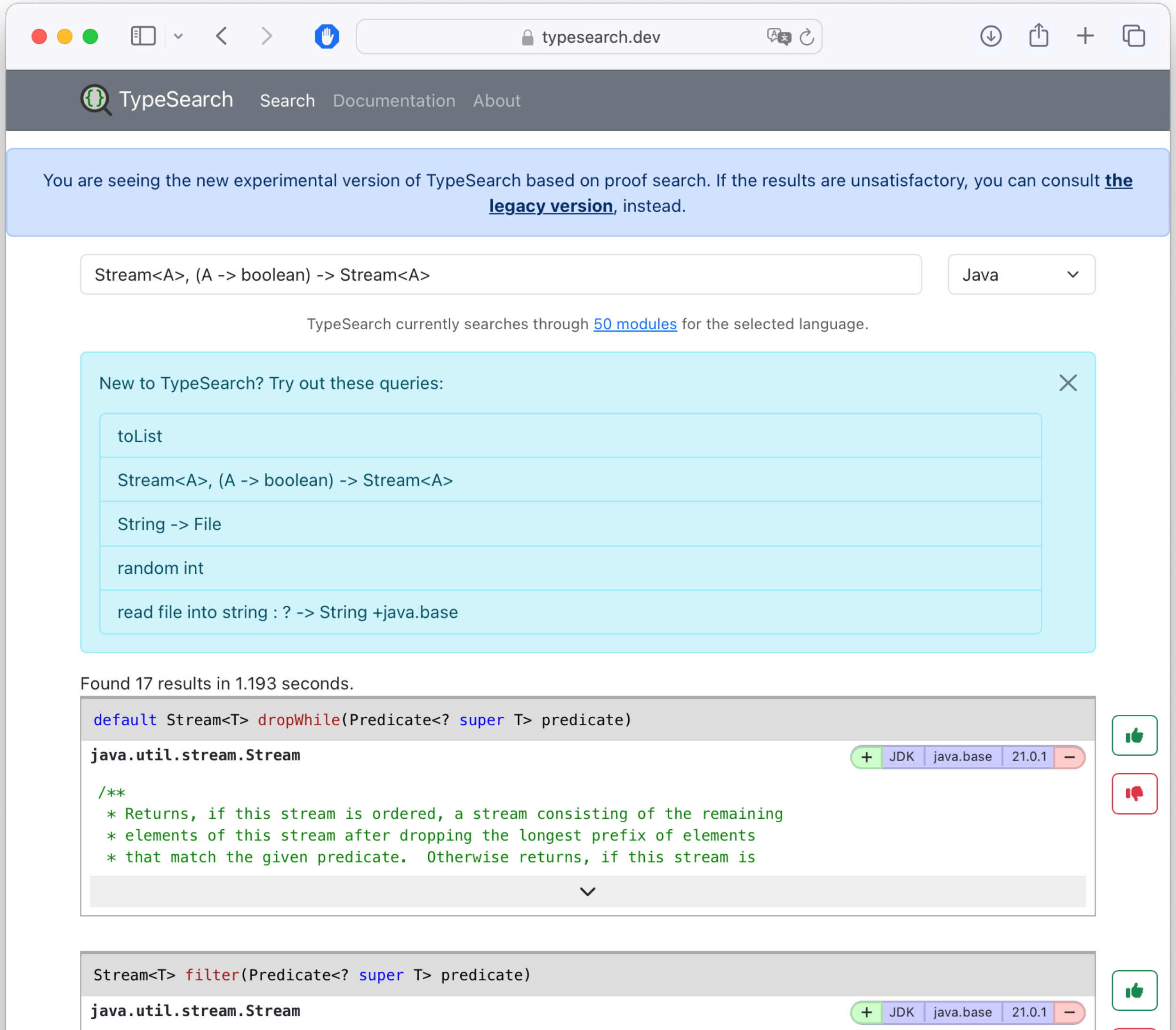
SCALA'16, October 30–31, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4648-1/16/10...$15.00
http://dx.doi.org/10.1145/2998392.2998405

95

SCALA'16, October 30–31, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4648-1/16/10...$15.00
http://dx.doi.org/10.1145/2998392.2998405

Hoogle for the hungry masses

Type-based API Search for All – typesearch.dev



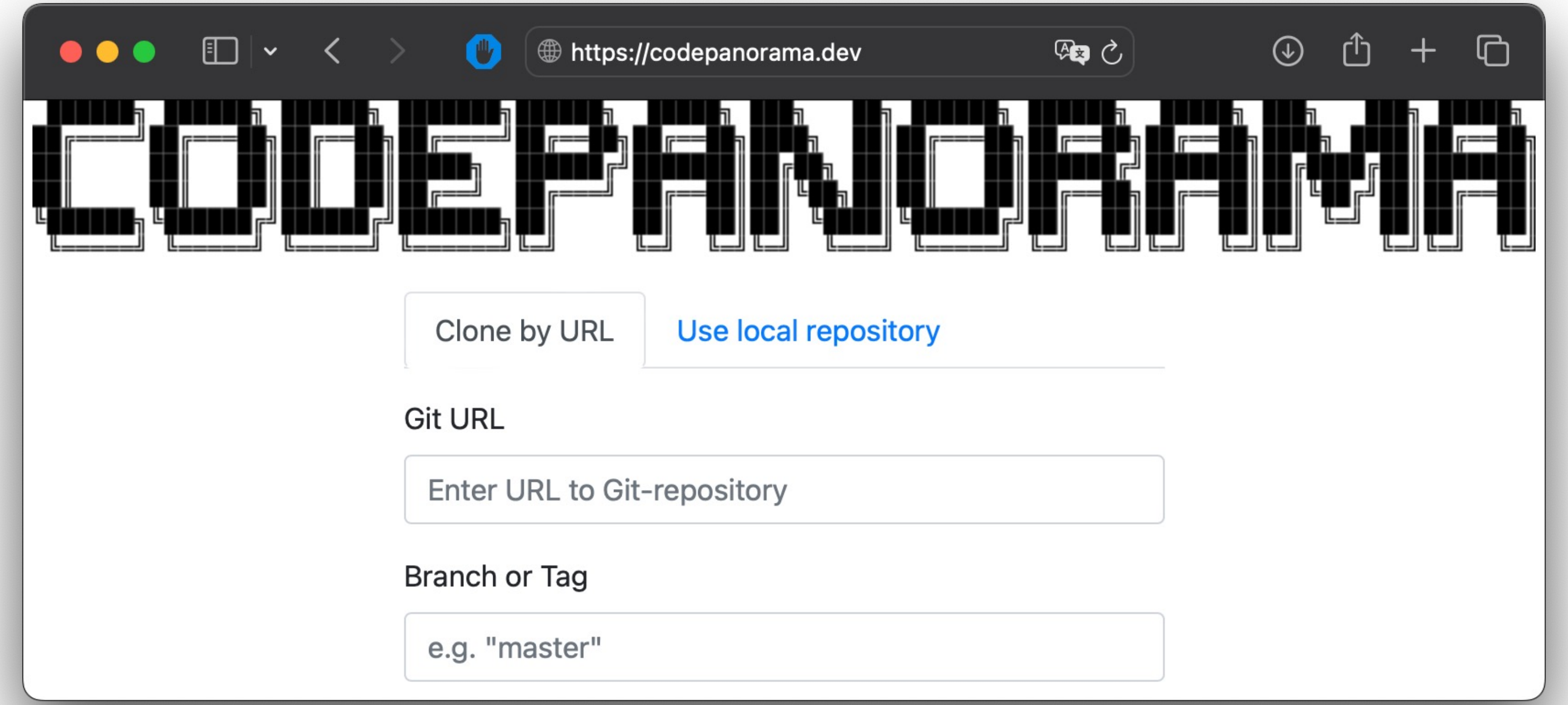
- Targeted at mainstream (typed OO) languages
- Inspiration: Curry-Howard Isomorphism
- Type Search is Proof Search!
- AdHoc → General
- Code synthesis also possible

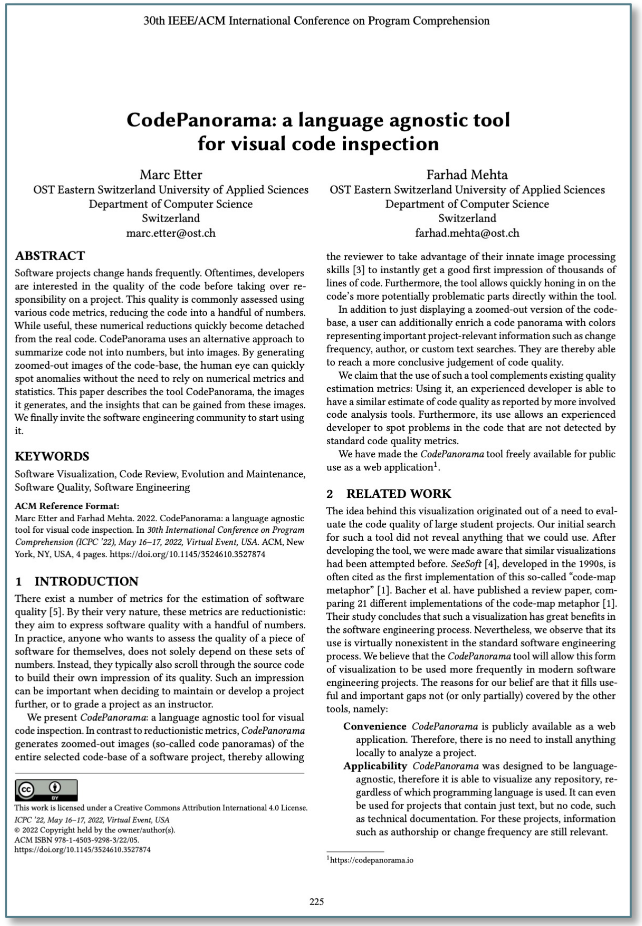
- TyDe Workshop ICFP 2024 (ACM)

CodePanorama

CodePanorama

The 10 ms code review





30th IEEE/ACM International Conference on Program Comprehension

CodePanorama: a language agnostic tool for visual code inspection

Marc Etter
OST Eastern Switzerland University of Applied Sciences
Department of Computer Science
Switzerland
marc.etter@ost.ch

Farhad Mehta
OST Eastern Switzerland University of Applied Sciences
Department of Computer Science
Switzerland
farhad.mehta@ost.ch

ABSTRACT

Software projects change hands frequently. Oftentimes, developers are interested in the quality of the code before taking over responsibility on a project. This quality is commonly assessed using various code metrics, reducing the code into a handful of numbers. While useful, these numerical reductions quickly become detached from the real code. CodePanorama uses an alternative approach to summarize code not into numbers, but into images. By generating zoomed-out images of the code-base, the human eye can quickly spot anomalies without the need to rely on numerical metrics and statistics. This paper describes the tool CodePanorama, the images it generates, and the insights that can be gained from these images. We finally invite the software engineering community to start using it.

KEYWORDS

Software Visualization, Code Review, Evolution and Maintenance, Software Quality, Software Engineering

ACM Reference Format:

Marc Etter and Farhad Mehta. 2022. CodePanorama: a language agnostic tool for visual code inspection. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3524610.3527874

1 INTRODUCTION

There exist a number of metrics for the estimation of software quality [5]. By their very nature, these metrics are reductionistic: they aim to express software quality with a handful of numbers. In practice, anyone who wants to assess the quality of a piece of software for themselves, does not solely depend on these sets of numbers. Instead, they typically also scroll through the source code to build their own impression of its quality. Such an impression can be important when deciding to maintain or develop a project further, or to grade a project as an instructor.

We present *CodePanorama*: a language agnostic tool for visual code inspection. In contrast to reductionistic metrics, *CodePanorama* generates zoomed-out images (so-called code panoramas) of the entire selected code-base of a software project, thereby allowing the reviewer to take advantage of their innate image processing skills [3] to instantly get a good first impression of thousands of lines of code. Furthermore, the tool allows quickly honing in on the code's more potentially problematic parts directly within the tool.

In addition to just displaying a zoomed-out version of the code-base, a user can additionally enrich a code panorama with colors representing important project-relevant information such as change frequency, author, or custom text searches. They are thereby able to reach a more conclusive judgement of code quality.

We claim that the use of such a tool complements existing quality estimation metrics: Using it, an experienced developer is able to have a similar estimate of code quality as reported by more involved code analysis tools. Furthermore, its use allows an experienced developer to spot problems in the code that are not detected by standard code quality metrics.

We have made the *CodePanorama* tool freely available for public use as a web application[1].

2 RELATED WORK

The idea behind this visualization originated out of a need to evaluate the code quality of large student projects. Our initial search for such a tool did not reveal anything that we could use. After developing the tool, we were made aware that similar visualizations had been attempted before. *SeeSoft* [4], developed in the 1990s, is often cited as the first implementation of this so-called "code-map metaphor" [1]. Bacher et al. have published a review paper, comparing 21 different implementations of the code-map metaphor [1]. Their study concludes that such a visualization has great benefits in the software engineering process. Nevertheless, we observe that its use is virtually nonexistent in the standard software engineering process. We believe that the *CodePanorama* tool will allow this form of visualization to be used more frequently in modern software engineering projects. The reasons for our belief are that it fills useful and important gaps not (or only partially) covered by the other tools, namely:

Convenience *CodePanorama* is publicly available as a web application. Therefore, there is no need to install anything locally to analyze a project.

Applicability *CodePanorama* was designed to be language-agnostic, therefore it is able to visualize any repository, regardless of which programming language is used. It can even be used for projects that contain just text, but no code, such as technical documentation. For these projects, information such as authorship or change frequency are still relevant.

ACM ISBN 978-1-4503-9298-3/22/05.
https://doi.org/10.1145/3524610.3527874

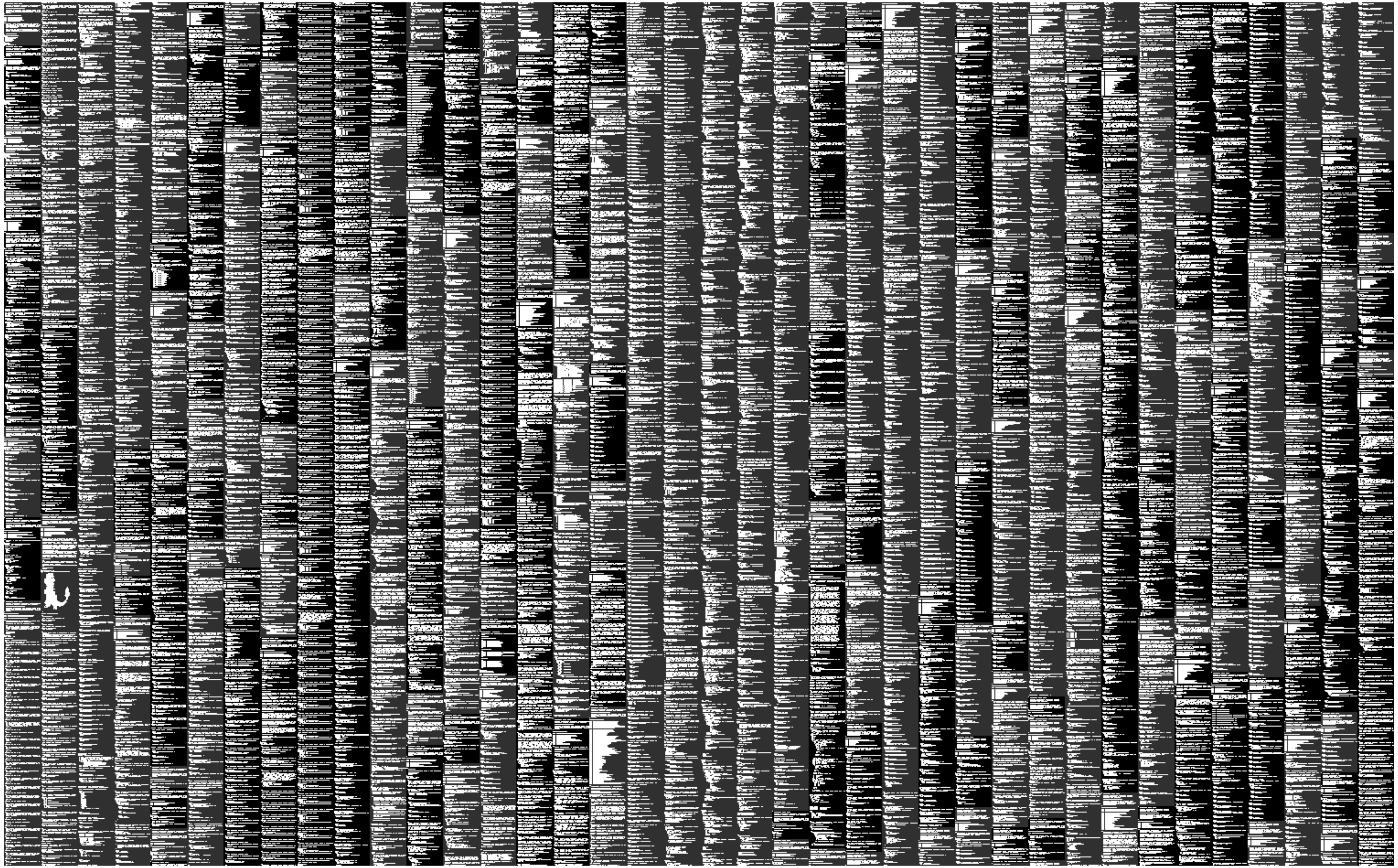
[1] https://codepanorama.io

225

ICPC 2022 (ACM/IEEE)

CodePanorama

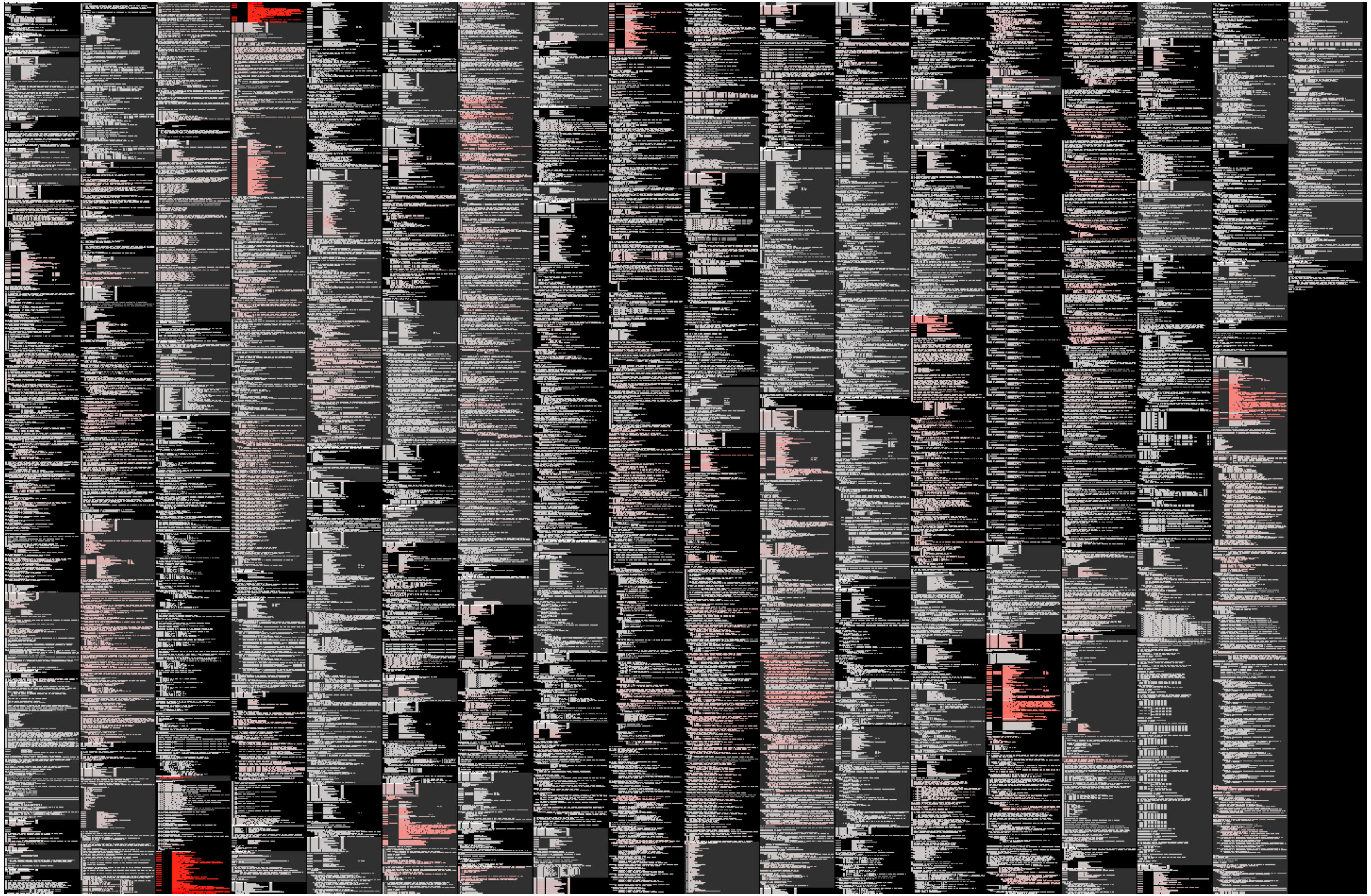
The 10 ms code review



https://github.com/google/guava

CodePanorama

The 10 ms code review

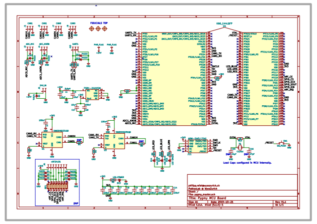


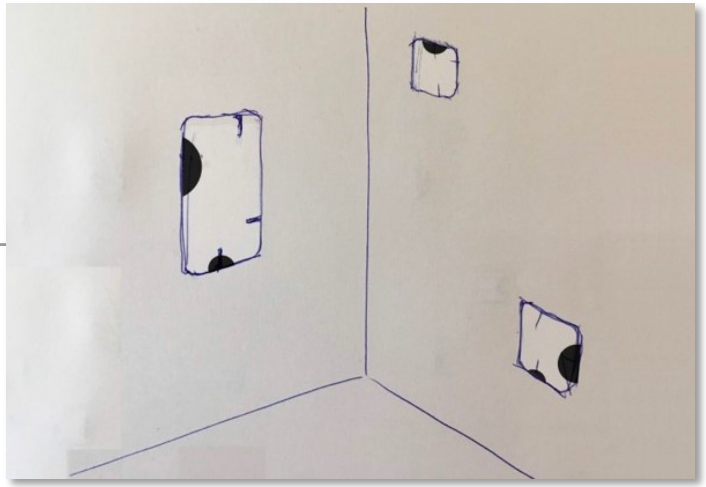
https://github.com/haskell-servant/servant
Highlights: Change Frequency

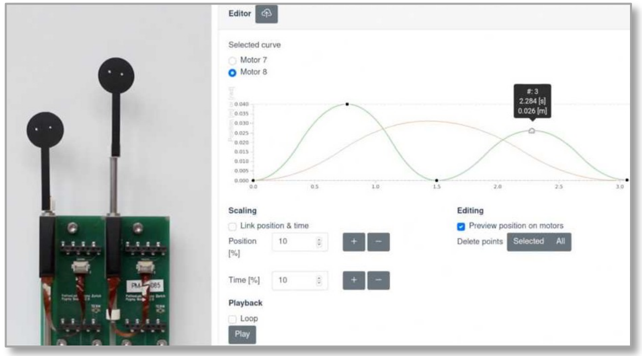
Robotics

Robotic Artwork

Joint work with artis duo Pors & Rao





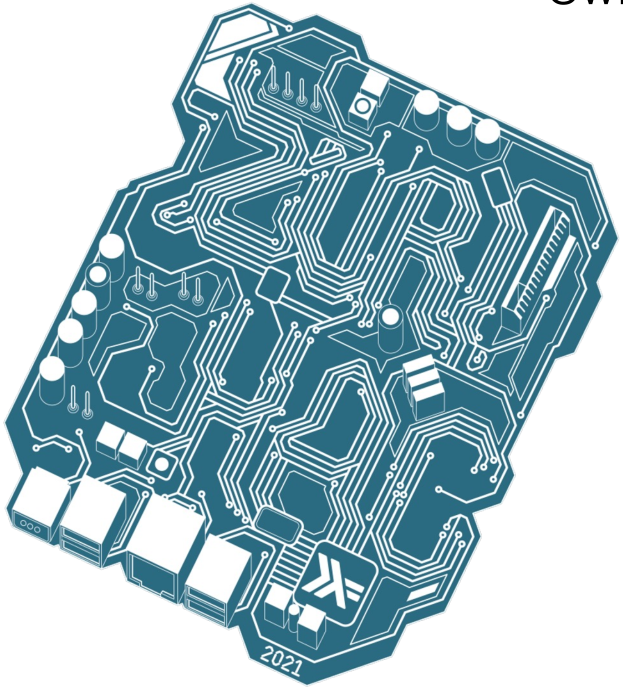


- “Untitled” https://www.youtube.com/watch?v=RlDoAHKzZu0&t=110s
- Using Functional Reactive Programming (FRP) to improve developer experience and control
- FARM ICFP 2024

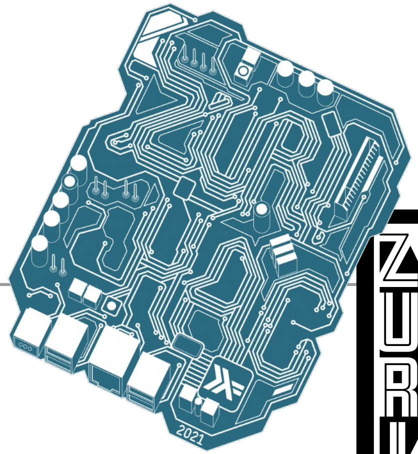
ZuriHac

ZuriHac

The biggest Haskell community event in the world: a three-day grassroots coding festival co-organized by the Zürich Friends of Haskell and the OST Eastern Switzerland University of Applied Science.



ZuriHac



- > 400 Participants
  - 50% Europe, 35% CH, 15% Overseas
  - 75% Industry, 25% Academia
- Organizing since 2017 at the OST (Before: Google, ETHZ, Better AG)
- Main aim: To promote and contribute to the state of the art in principled computer programming.
- Focus: Functional Programming and Haskell
- But not only...
  - Applications: FRP, Build Systems, SE Practices, Metaprogramming, Hardware Design, Verification...
  - Other Programming Languages: Agda, Verse, Racket, Dhal, Nix, ...
  - Fundamental Concepts: Type Systems, PL Semantics, Logik, Category Theory, Combinators, ...

Highlights ZuriHac 2024

- Keynotes (others were great too)
  - Low level: “Functional Hardware Description and verification” (Mary Sheeran)
  - High level: “Haskell in Space”: Runtime verification at NASA (Ivan Perez)
  - With both feet on the ground: “Making people dance with Haskell”: (Alex McLean)
- 3 Advanced tracks: FRP, Generic Programming, Dependent Types
- Beginner Track with 30 Participants (Eliane Schmidli)
- For some, just too overwhelming: "I just arrived in Paris and noticed that I may have forgot my luggage at the OST. Could you please take a look"

The biggest Haskell community event in the world: a three-day grassroots coding festival co-organized by the Zürich Friends of Haskell and the OST Eastern Switzerland University of Applied Science.

Saturday 7 June — Monday 9 June 2025
Rapperswil, Switzerland

Registration is open and free.

Come talk to me if you have any questions – Farhad Mehta

https://zurihac.info
https://zfoh.ch/zurihac2025/