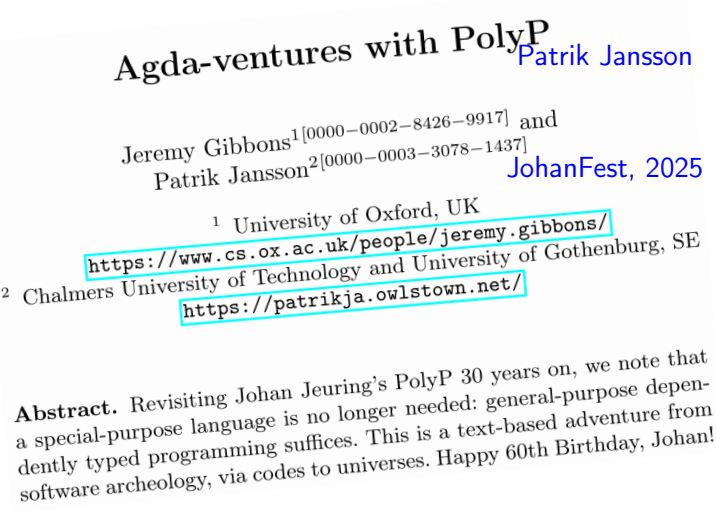
Agda-ventures with PolyP and Johan Jeuring from 30 to 60

Patrik Jansson

JohanFest, 2025

Agda-ventures with PolyP and Johan Jeuring from 30 to 60



Agda-ventures with PolyP

Patrik Jansson

JohanFest, 2025

Jeremy Gibbons[1][0000−0002−8426−9917] and
Patrik Jansson[2][0000−0003−3078−1437]

[1] University of Oxford, UK
https://www.cs.ox.ac.uk/people/jeremy.gibbons/

[2] Chalmers University of Technology and University of Gothenburg, SE
https://patrikja.owlstown.net/

Abstract. Revisiting Johan Jeuring's PolyP 30 years on, we note that a special-purpose language is no longer needed: general-purpose dependently typed programming suffices. This is a text-based adventure from software archeology, via codes to universes. Happy 60th Birthday, Johan!

Agda-ventures with PolyP and Johan Jeuring from 30 to 60

Agda-ventures with PolyP

Patrik Jansson

JohanFest, 2025

Jeremy Gibbons[1][0000−0002−8426−9917] and
Patrik Jansson[2][0000−0003−3078−1437]

[1] University of Oxford, UK
https://www.cs.ox.ac.uk/people/jeremy.gibbons/
[2] Chalmers University of Technology and University of Gothenburg, SE
https://patrikja.owlstown.net/

Abstract. Revisiting Johan Jeuring's PolyP 30 years on, we note that a special-purpose language is no longer needed: general-purpose dependently typed programming suffices. This is a text-based adventure from software archeology, via codes to universes. Happy 60th Birthday, Johan!

PolyP: early history

- PolyP, developed by Johan and Patrik, was a new language for "polytypic programming".
- Developed 1995–2006 to explore programs *parameterized by the shape of datatypes*.
- Examples: map, cata, flatten, ...

Early releases / revision history from the repo: github.com/patrikja/PolyP/

```
980413: version 0.6 for hbc, ghc, hugs (Haskell 1.4)
980312: version 0.5 for hbc, ghc, hugs (Haskell 1.4)
970510: version 0.4 for hbc 0.9999.3
960922: version 0.3 for hbc 0.9999.3
960830: version 0.2 for hbc 0.9999.3
960819: version 0.1 for hbc 0.9999.3 (Haskell 1.3)
960717: version 0   for Hugs (in Haskell 1.2)
```

Historical detail: first release of git was in 2005, PolyP was developed using CVS for version control.

PolyP, polytypic programming, and generic programming

The work on PolyP has been quite influential (but perhaps in a small bubble;-)

Patrik Jansson

FOLLOWING

Professor of Computer Science, Chalmers University of Technology and University of Gothenburg
Verified email at chalmers.se - Homepage

Computer Science Software Technology Functional Programming Programming Languages Climate Impact

TITLE	CITED BY	YEAR
PolyP—a polytypic programming language extension P Jansson, J Jeuring Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of …	366	1997
Generic programming R Backhouse, P Jansson, J Jeuring, L Meertens International School on Advanced Functional Programming, 28-115	243	1998
Polytypic programming J Jeuring, P Jansson International School on Advanced Functional Programming, 68-114	163	1996
Fast and loose reasoning is morally correct	128	2006

What is Polytypic Programming?

- From Webster's Dictionary:

  poly·typ·ic *[ˌpä-lē-ˈti-pik], adj.: having or involving several different types*

- It's about writing a single program that operates on an entire family of types.
- Other names:
  - Structurally polymorphic (Ralf Hinze)
  - Shape polymorphic (Barry Jay)
  - Type parametric (Tim Sheard)
  - Datatype-generic (Roland Backhouse, Jeremy Gibbons)
- PolyP was created to solve structural programming problems like:
  - Equality and pattern matching
  - Folding and mapping
  - Traversal and pretty-printing
  - Unification, ...
- The key goal was to maintain strong, static type safety.

The PolyP implementation

- PolyP was not a standalone language – it was implemented as a *preprocessor* for Haskell.
- This meant that polytypic code was possible, but not “first class”, or properly integrated
- PolyP was a major advancement, but it limited the expressiveness of the polytypic code.

The PolyP implementation and the new era

- PolyP was not a standalone language – it was implemented as a *preprocessor* for Haskell.
- This meant that polytypic code was possible, but not "first class", or properly integrated
- PolyP was a major advancement, but it limited the expressiveness of the polytypic code.

The new era

- What required a preprocessor in 1995 can be achieved today by programming in Agda.
- Thanks to the advances in language design over the past 30 years.
- Dependent types, type driven development, universes, ...

From PolyP to an Agda-adventure

- Dependent types allow for types and operations on types to be treated as values.
- This means types can be *analyzed and manipulated directly*[1] within the language itself.
- A separate preprocessor is no longer required.

[1] Not quite true, see next slide!

From PolyP to an Agda-adventure

- Dependent types allow for types and operations on types to be treated as values.
- This means types can be *analyzed and manipulated directly*[1] within the language itself.
- A separate preprocessor is no longer required.
- The paper uses the dependently typed language *Agda* as the modern example.
- Interactive development of programs and proofs in Agda is a bit like a "text-based adventure game"
- The type checker guides you through the process, helping you solve problems and fill in "holes" in your code.

[1] Not quite true, see next slide!

A challenge, and universes to the rescue

The general idea with PolyP is that "a polytypic function can be viewed as a family of functions: one function for each datatype" [Jeuring and Jansson 1996], defined by induction over the structure of the datatype.
The core challenge:

- Even with dependent types, you cannot directly analyze the types themselves; they are "black boxes".
- You need a way to represent the type's structure in a format that can be manipulated.

A challenge, and universes to the rescue

The general idea with PolyP is that "a polytypic function can be viewed as a family of functions: one function for each datatype" [Jeuring and Jansson 1996], defined by induction over the structure of the datatype.

The core challenge:

- Even with dependent types, you cannot directly analyze the types themselves; they are "black boxes".
- You need a way to represent the type's structure in a format that can be manipulated.

The solution: codes for types

- The solution is to create a universe of *codes* for types.
- Codes are terms in an algebraic datatype that represent the structure of a type.
- This allows the type's structure to be analyzed and manipulated like regular data.
- The actual types are recovered by an evaluator (semantics) for the codes.

PolyP reminder (based on the AFP'96 lecture notes)

- PolyP used *polynomial types*: sums and products of some basic types
- For recursive datatypes: *regular functors*: initial algebras for functors constructed from polynomial operations on a type parameter
- And to handle polymorphic (container) datatypes: *regular bifunctors*.

For example Jeuring and Jansson 1996, the Haskell datatypes of lists and rose trees

$$\begin{array}{lcl} \text{data } \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\ \text{data } \mathit{Rose}\ a & = & \mathit{Fork}\ a\ (\mathit{List}\ (\mathit{Rose}\ a)) \end{array}$$

are the initial algebras respectively of the bifunctors written in PolyP as

$$\begin{array}{lcl} \mathit{FList} & = & () + \mathsf{Par} \times \mathsf{Rec} \\ \mathit{FRose} & = & \mathsf{Par} \times (\mathit{List} \mathbin{@} \mathsf{Rec}) \end{array}$$

Early PolyP examples cont.

Inductive datatypes Mu f a for bifunctor f have a constructor and a destructor:

$$\begin{array}{lcl} \mathit{inn} & :: & f\ a\ (\mathsf{Mu}\ f\ a) \to \mathsf{Mu}\ f\ a \\ \mathit{out} & :: & \mathsf{Mu}\ f\ a \to f\ a\ (\mathsf{Mu}\ f\ a) \end{array}$$

A polytypic 'map' function for inductive datatypes (pmap) and regular bifunctors (fmap):

$$\begin{array}{lcl} \mathit{pmap} & :: & (a \to b) \to \mathsf{Mu}\ f\ a \to \mathsf{Mu}\ f\ b \\ \mathit{pmap}\ p & = & \mathit{inn} \cdot \mathit{fmap}\ p\ (\mathit{pmap}\ p) \cdot \mathit{out} \end{array}$$

$$\begin{array}{l} \mathsf{polytypic}\ \mathit{fmap} :: (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d \\ \quad = \lambda\ p\ r \to \mathsf{case}\ f\ \mathsf{of} \\ \qquad \begin{array}{lcl} f + g & \to & \mathit{fmap}\ p\ r \mathbin{\text{-+-}} \mathit{fmap}\ p\ r \\ () & \to & \mathit{id} \\ \mathsf{Con}\ t & \to & \mathit{id} \\ f \times g & \to & \mathit{fmap}\ p\ r \mathbin{\text{-}\times\text{-}} \mathit{fmap}\ p\ r \\ d\ @\ g & \to & \mathit{pmap}\ (\mathit{fmap}\ p\ r) \\ \mathsf{Par} & \to & p \\ \mathsf{Rec} & \to & r \end{array} \end{array}$$

Agda version: first simple codes for types

We start with an algebraic datatype of codes for the types in a small universe:

```
data Code₀ : Set where
  NatT BoolT UnitT : Code₀
  _*_ _+_ : Code₀ → Code₀ → Code₀
```

Agda version: first simple codes for types

We start with an algebraic datatype of codes for the types in a small universe:

```
data Code₀ : Set where
  NatT BoolT UnitT : Code₀
  _*_ _+_ : Code₀ → Code₀ → Code₀
```

Example code: roughly *Maybe* (*Nat*, *Bool*) in Haskell:

```
MaybeNatBoolCode : Code₀
MaybeNatBoolCode = UnitT + (NatT * BoolT)
```

Agda version: first simple codes for types

We start with an algebraic datatype of codes for the types in a small universe:

```
data Code₀ : Set where
  NatT BoolT UnitT : Code₀
  _*_ _+_ : Code₀ → Code₀ → Code₀
```

Example code: roughly *Maybe* (*Nat*, *Bool*) in Haskell:

```
MaybeNatBoolCode : Code₀
MaybeNatBoolCode = UnitT + (NatT * BoolT)
```

We can then define the interpretation of codes as types:

```
⟦_⟧₀ : Code₀ → Set
⟦ NatT ⟧₀   = Nat
⟦ BoolT ⟧₀  = Bool
⟦ UnitT ⟧₀  = ⊤
⟦ c * c' ⟧₀ = ⟦ c ⟧₀ × ⟦ c' ⟧₀
⟦ c + c' ⟧₀ = Sum ⟦ c ⟧₀ ⟦ c' ⟧₀
```

Simple polytypic equality check

- First example of a polytypic function in Agda: equality.

  $\mathsf{equal}_0 : \{\ c : \mathsf{Code}_0\ \} \to [\![\, c \,]\!]_0 \to [\![\, c \,]\!]_0 \to \mathsf{Bool}$

- Note the dependent type: it takes the code for some type in the universe, then two elements whose types are computed from the code.

Simple polytypic equality check

- First example of a polytypic function in Agda: equality.

$$\mathsf{equal}_0 : \{\ c : \mathsf{Code}_0\ \} \to [\![\, c \,]\!]_0 \to [\![\, c \,]\!]_0 \to \mathsf{Bool}$$

- Note the dependent type: it takes the code for some type in the universe, then two elements whose types are computed from the code.
- The implementation is straightforward:

$$\begin{array}{lllll}
\mathsf{equal}_0\ \{\mathsf{NatT}\} & n & m & = & (n ==_N m) \\
\mathsf{equal}_0\ \{\mathsf{BoolT}\} & x & y & = & (x ==_B y) \\
\mathsf{equal}_0\ \{\mathsf{UnitT}\} & x & y & = & (x ==_U y) \\
\mathsf{equal}_0\ \{c * c'\} & (x\ ,\ x') & (y\ ,\ y') & = & \mathsf{equal}_0\ x\ y \wedge \mathsf{equal}_0\ x'\ y' \\
\mathsf{equal}_0\ \{c + c'\} & (\mathsf{inj}_1\ x) & (\mathsf{inj}_1\ y) & = & \mathsf{equal}_0\ x\ y \\
\mathsf{equal}_0\ \{c + c'\} & (\mathsf{inj}_2\ x') & (\mathsf{inj}_2\ y') & = & \mathsf{equal}_0\ x'\ y' \\
\mathsf{equal}_0\ \{c + c'\} & _ & _ & = & \mathsf{false}
\end{array}$$

Codes for PolyP-style generic programming

```
mutual
  data Type : Set where
    NatTy BoolTy UnitTy : Type

  data Functor : Set where
    Fix : Bifunctor → Functor

  data Bifunctor : Set where
    _*_ _+_ : Bifunctor → Bifunctor → Bifunctor
    Const   : Type                  → Bifunctor
    _•_     : Functor   → Bifunctor → Bifunctor
    Par Rec :                         Bifunctor

data Mu (f : Set → Set) : Set where        In : f (Mu f) → Mu f

out : { f : Set → Set } → Mu f → f (Mu f)
out (In xs) = xs
```

Interpreting codes as types

```
mutual
  [[_]]T : Type      →                 Set
  [[_]]F : Functor   → Set →           Set
  [[_]]B : Bifunctor → Set → Set →     Set

  [[ NatTy ]]T  = Nat
  [[ BoolTy ]]T = Bool
  [[ UnitTy ]]T = ⊤

  [[ Fix f ]]F p = Mu ([[ f ]]B p)

  [[ f ∗ g ]]B   p r = [[ f ]]B p r × [[ g ]]B p r
  [[ f + g ]]B   p r = Sum ([[ f ]]B p r) ([[ g ]]B p r)
  [[ Const t ]]B p r = [[ t ]]T
  [[ d • f ]]B   p r = [[ d ]]F ([[ f ]]B p r)
  [[ Par ]]B     p r = p
  [[ Rec ]]B     p r = r
```

Examples: codes for the (polymorphic) list type

```
ListF  : Bifunctor
ListF  = Const UnitTy + (Par ∗ Rec)

ListC  : Functor
ListC  = Fix ListF

MyList : Set → Set
MyList = [[ ListC ]]F
```

A conversion function to built-in lists (using cata — defined on the next slide):

```
fromMyList : MyList a → List a
fromMyList = cata ListF alg where
  alg : [[ ListF ]]B a (List a) → List a
  alg (inj1 tt)          = []
  alg (inj2 ( x , xs )) = x :: xs
```

PolyP-style classics: pmap, fmap, and cata

```
mutual
  {-# TERMINATING #-}
  pmap : (d : Functor)      → (a → b) → ⟦ d ⟧F a → ⟦ d ⟧F b
  fmap : (f : Bifunctor)    → (a → b) → (c → d) → ⟦ f ⟧B a c → ⟦ f ⟧B b d
  cata : (f : Bifunctor)    → (⟦ f ⟧B a b → b) → ⟦ Fix f ⟧F a → b

  cata f h (In xs)          = h (fmap f id (cata f h) xs)

  pmap (Fix f) g            = cata f (In ∘ fmap f g id)

  fmap (f ∗ g) p r (x , y)  = (fmap f p r x , fmap g p r y)
  fmap (f + g) p r (inj₁ x) = inj₁ (fmap f p r x)
  fmap (f + g) p r (inj₂ y) = inj₂ (fmap g p r y)
  fmap (Const t) p r x      = x
  fmap (d • g) p r xs       = pmap d (fmap g p r) xs
  fmap Par p r              = p
  fmap Rec p r              = r
```

More early examples (redone): crush and flatten

- One canonical example of a polytypic function on polymorphic container datatypes is to "crush" it [Meertens 1996], aggregating the elements using a monoid:

```
mutual
  crush : (a → a → a) → a → (d : Functor) → ⟦ d ⟧F a → a
  crush _⊕_ e (Fix f) = cata f (crushB _⊕_ e f)

  crushB : (a → a → a) → a → (f : Bifunctor) → ⟦ f ⟧B a a → a
```

- with the bifunctor parts elided.

More early examples (redone): crush and flatten

- One canonical example of a polytypic function on polymorphic container datatypes is to "crush" it [Meertens 1996], aggregating the elements using a monoid:

```
mutual
  crush : (a → a → a) → a → (d : Functor) → ⟦ d ⟧F a → a
  crush _⊕_ e (Fix f) = cata f (crushB _⊕_ e f)

  crushB : (a → a → a) → a → (f : Bifunctor) → ⟦ f ⟧B a a → a
```

- with the bifunctor parts elided.
- Using crush we can flatten a container to a list:

```
flatten : (d : Functor) → ⟦ d ⟧F a → List a
flatten d = crush _++_ [] d ∘ pmap d (λ x → [ x ])
```

Polytypic Data Conversion [Jansson and Jeuring 2002]

- pack/unpack, print/parse, show/read, ...
- Ensures $parse \circ print = id$ [a]
- Triples: (pack, unpack, round-trip proof)

Polytypic Data Conversion [Jansson and Jeuring 2002]

- pack/unpack, print/parse, show/read, ...
- Ensures $parse \circ print = id$ [a]
- Triples: (pack, unpack, round-trip proof)
- The festschrift chapter continues with data conversion, but we will take a more scenic route to the end of this talk.

[a] The 2002 paper uses Arrows.

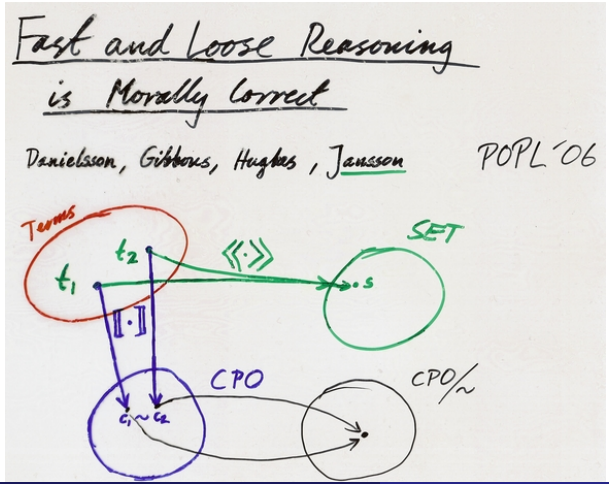
Figure: Johan@37 & Patrik@30, 2002, Hovås, Göteborg.

Mentoring: IFIP WG 2.1 meeting #61 in Turkey (2006)

"We have arranged for a total solar eclipse to take place during the excursion"
Lambert Meertens

Mentoring: IFIP WG 2.1 meeting #61 in Turkey (2006)

"We have arranged for a total solar eclipse to take place during the excursion"
Lambert Meertens



More mentoring: IFIP WG 2.1 meeting in Kyoto (2007)

- WG 2.1 on "Algorithmic Languages and Calculi"
- Johan presented "Strategies feedback",

  Exercises in mathematics . . . [this talk] introduces a formalism for specifying strategies for solving exercises.
- Andres: Implementing dependent types in Haskell
- Patrik: a tutorial on Agda [Ulf Norell]

Figure: Johan & Andres, 2007, Kyoto, Japan.

IFIP WG 2.1 group photo in Kyoto (2007)

M&Bernhard Möller, Zhenjiang Hu, Patrik Jansson, Kim Solin, Atsushi Igarashi, Johan jeuring, Michel Sintzoff, Bruno Oliveira, Peter Pepper, Lambert Meertens, Akimasa Morihata, Carroll Morgan, Andres Löh, Roland&H. Backhouse, Eiiti Wada, Varmo Vene, Alberto Pardo, Shin-Cheng Mu

IFIP WG 2.1@50 in Rome: three academic generations

Figure: Johan Jeuring, Patrik Jansson, Nils Anders Danielsson; Rome, Italy, 2012

Johan @ 60, PolyP @ 30

Johan Jeuring

FOLLOW

Professor of Software Technology for Learning and Teaching, ICS & FI, Utrecht University
Verified email at uu.nl - Homepage

Technology-enhanced learn… intelligent tutoring systems artificial intelligence in educ… Computer Science Education

TITLE	CITED BY	YEAR
A systematic literature review of automated feedback generation for programming exercises H Keuning, J Jeuring, B Heeren ACM Transactions on Computing Education (TOCE) 19 (1), 3	440	2019
PolyP—a polytypic programming language extension P Jansson, J Jeuring Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of …	366	1997
Generic programming R Backhouse, P Jansson, J Jeuring, L Meertens Advanced Functional Programming: Third International School, AFP'98, Braga …	243	1999
Towards a Systematic Review of Automated Feedback Generation for Programming	221	2016

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.
- You are in a hole, with some objects at your disposal, and you have to find a way out.

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.
- You are in a hole, with some objects at your disposal, and you have to find a way out.
- You keep getting sent on side-quests.

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.
- You are in a hole, with some objects at your disposal, and you have to find a way out.
- You keep getting sent on side-quests.
- Sometimes it feels like you are fighting the typechecker;

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.
- You are in a hole, with some objects at your disposal, and you have to find a way out.
- You keep getting sent on side-quests.
- Sometimes it feels like you are fighting the typechecker;
- but sometimes it feels like the universe is on your side,

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.
- You are in a hole, with some objects at your disposal, and you have to find a way out.
- You keep getting sent on side-quests.
- Sometimes it feels like you are fighting the typechecker;
- but sometimes it feels like the universe is on your side,
- and the obstacles are magically eliminated.

Wrapping up: type-driven development, Agda adventures,

- Programming in this type-driven style feels like a text-based adventure game.
- You are in a hole, with some objects at your disposal, and you have to find a way out.
- You keep getting sent on side-quests.
- Sometimes it feels like you are fighting the typechecker;
- but sometimes it feels like the universe is on your side,
- and the obstacles are magically eliminated.
- In recent years, Johan's research interests have shifted from programming languages
- to technology-enhanced learning, including 'serious games':
- perhaps Johan can see scope for closing the circle by bringing the two back together?

Happy 60th birthday, Johan! (also new chair of IFIP WG 2.1)

Agda-ventures with PolyP

Jeremy Gibbons[1][0000−0002−8426−9917] and
Patrik Jansson[2][0000−0003−3078−1437]

[1] University of Oxford, UK
https://www.cs.ox.ac.uk/people/jeremy.gibbons/
[2] Chalmers University of Technology and University of Gothenburg, SE
https://patrikja.owlstown.net/

Abstract. Revisiting Johan Jeuring's PolyP 30 years on, we note that a special-purpose language is no longer needed: general-purpose dependently typed programming suffices. This is a text-based adventure from software archeology, via codes to universes. Happy 60th Birthday, Johan!

Extra slides coming up

The rest of the presentation was not presented, but just kept as extra slides.

IFIP WG 2.1 in Kyoto (2007-09)

Johan Jeuring, Roland Backhouse, Lambert Meertens, Sept. 2007, Kyoto, Japan.
Photo by Carroll Morgan.

Johan climbing the promotion ladder? (2010)

IFIP WG 2.1 meeting in Rome (2012)

Figure: Doaitse Swierstra, Wouter Swierstra, Johan Jeuring, Andres Löh, Nils Anders Danielsson, Rome, 2012

IFIP WG 2.1 meeting 83 in Portugal (2025-09-08)

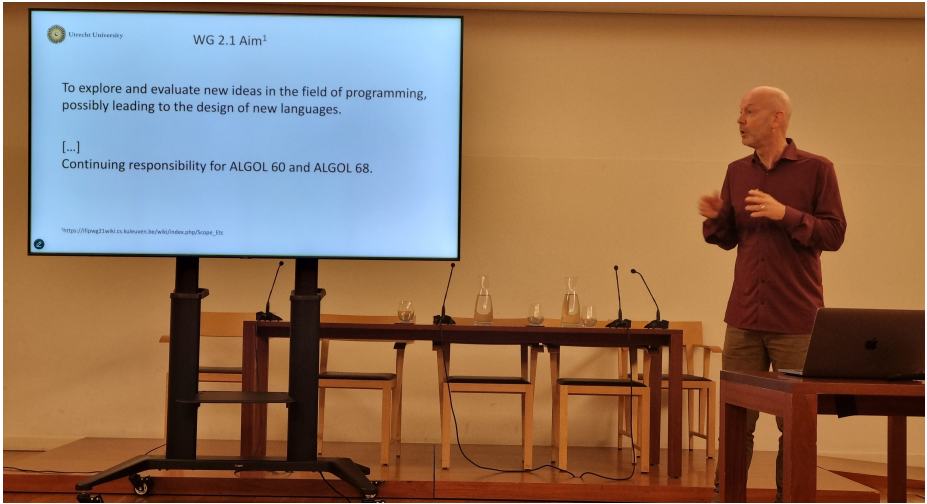


Figure: Johan Jeuring, Viana do Castelo, Portugal, 2025

References I

- Backhouse, Roland Carl et al. (1998). "Generic Programming: An Introduction". In: *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*. Ed. by S. Doaitse Swierstra, Pedro Rangel Henriques, and José Nuno Oliveira. Vol. 1608. Lecture Notes in Computer Science. Springer, pp. 28–115. DOI: 10.1007/10704973_2.
- Danielsson, Nils Anders et al. (2006). "Fast and loose reasoning is morally correct". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, pp. 206–217. ISBN: 1595930272. DOI: 10.1145/1111037.1111056.

References II

- Jansson, Patrik and Johan Jeuring (1997). "PolyP - A Polytypic Programming Language". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. Ed. by Peter Lee, Fritz Henglein, and Neil D. Jones. ACM Press, pp. 470–482. ISBN: 0897918533. DOI: 10.1145/263699.263763.
- — (1998). "Polytypic Unification". In: *J. Funct. Program.* 8.5, pp. 527–536. DOI: 10.1017/S095679689800313X. URL: http://journals.cambridge.org/action/displayAbstract?aid=44195.
- — (2002). "Polytypic data conversion programs". In: *Sci. Comput. Program.* 43.1, pp. 35–75. DOI: 10.1016/S0167-6423(01)00020-X.
- Jeuring, Johan (1995). "Polytypic Pattern Matching". In: ACM, pp. 238–248. DOI: 10.1145/224164.224212.

References III


Jeuring, Johan and Patrik Jansson (1996). "Polytypic Programming". In: *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*. Ed. by John Launchbury, Erik Meijer, and Tim Sheard. Vol. 1129. Lecture Notes in Computer Science. Springer, pp. 68–114. DOI: 10.1007/3-540-61628-4_3.

Meertens, Lambert (1996). "Calculate Polytypically!" In: *Programming Languages: Implementations, Logics, and Programs*. Ed. by H. Kuchen and S. D. Swierstra. Vol. 1140. Springer-Verlag, pp. 1–16. DOI: 10.1007/3-540-61756-6_73.