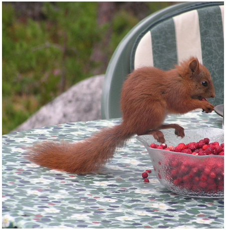
Strongly Typed Programs and Proofs

Patrik Jansson, fp.st.cse.chalmers.se

Chalmers University of Technology and University of Gothenburg

2015-08-21



"It is one of the first duties of a professor, in any subject, to exaggerate a little both the importance of his subject and his own importance in it" [A Mathematician's Apology, G. H. Hardy]

This talk: `https://github.com/patrikja/ProfLect`

My best results over the years (part 1)

Among my best results I count

- early work on Generic Programming [Backhouse et al., 1999, Jansson and Jeuring, 1997] (well cited)
- Polytypic Data Conversion Programs [Jansson and Jeuring, 2002]
- the Bologna structure (3y BSc + 2y MSc) at cse.chalmers.se in my role as Vice Head of Department
- self-evaluation reports for the CSE degrees (in my role as Head of the CSE programme). The BSc got “very high quality”.
- Global Systems Science work [Jaeger et al., 2013] leading to the FETPROACT1 call, the GRACeFUL project and the CoEGSS project.

My best results over the years (part 2)

. . . continued

- my PhD graduates: Norell, Danielsson, and Bernardy
- Fast and Loose Reasoning [Danielsson et al., 2006]
- Parametricity and dependent types [Bernardy et al., 2010]
- Algebra of Programming in Agda [Mu et al., 2009]
- Feat: functional enumeration of algebraic types [Duregård et al., 2012]

Functional Polytypic Programming (1995–2000)

What is a "polytypic function"?
Start from the normal *sum* function on lists:

$$
\begin{aligned}
&\mathit{sum} :: \mathit{Num}\ a \ \Rightarrow\ [a] \ \rightarrow\ a\\
&\mathit{sum}\ [\,] \qquad\quad = \ 0\\
&\mathit{sum}\ (x : \mathit{xs}) \ = \ x + \mathit{sum}\ \mathit{xs}
\end{aligned}
$$

then generalise to other datatypes like these

$$
\begin{array}{lcll}
\textbf{data}\ [a] & = & [\,] & \mid\ a\ :\ [a]\\
\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Leaf}\ a & \mid\ \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
\textbf{data}\ \mathit{Maybe}\ a & = & \mathit{Nothing} & \mid\ \mathit{Just}\ a\\
\textbf{data}\ \mathit{Rose}\ a & = & \mathit{Fork}\ a\ [\mathit{Rose}\ a] &
\end{array}
$$

The Haskell language extension PolyP (POPL'97)

We obtain

$$
\begin{aligned}
&\mathit{psum} :: (\mathit{Regular}\ d,\ \mathit{Num}\ a) \Rightarrow d\ a \to a\\
&\mathit{psum} = \mathit{cata}\ \mathit{fsum}
\end{aligned}
$$

where *fsum* is defined by induction over the pattern functor f of the regular datatype $d\ a$.

$$
\begin{aligned}
&\textbf{polytypic}\ \mathit{fsum} :: \mathit{Num}\ a \Rightarrow f\ a\ a \to a\\
&\quad = \textbf{case}\ f\ \textbf{of}\\
&\qquad\qquad g + h \quad\to \mathit{either}\ \mathit{fsum}\ \mathit{fsum}\\
&\qquad\qquad g * h \quad\to \lambda\,(x,\ y) \to \mathit{fsum}\ x + \mathit{fsum}\ y\\
&\qquad\qquad \mathit{Empty} \to \backslash x \to 0\\
&\qquad\qquad \mathit{Par} \quad\ \to \mathit{id}\\
&\qquad\qquad \mathit{Rec} \quad\ \to \mathit{id}\\
&\qquad\qquad d\,@\,g \quad\to \mathit{psum} \circ \mathit{pmap}\ \mathit{fsum}\\
&\qquad\qquad \mathit{Const}\ t \to \backslash x \to 0
\end{aligned}
$$

Summer schools lecture notes ($>$ *150* citations each):

- Polytypic Programming [Jeuring & Jansson, 1996]
- Generic Programming - An Introduction [Backhouse, Jansson, Jeuring & Meertens, 1999]

$$\begin{tikzcd}
F\ \mu F \arrow[r, "inn"] \arrow[d, "F\ (\![\alpha]\!)"] & \mu F \arrow[d, "(\![\alpha]\!)"] \\
F\ A \arrow[r, "\alpha"] & A
\end{tikzcd}$$

Notation:

$$\begin{aligned}
(\![\alpha]\!) &= \mathit{cata}\ \alpha \\
F\ h &= \mathit{fmap}\ h
\end{aligned}$$

Theoretical tools

Categories, functors and algebras

Category C, (endo-)functor $F : C \rightarrow C$, F-algebra $(A, \alpha : F\ A \rightarrow A)$,

Theoretical tools

Categories, functors and algebras

Category C, (endo-)functor $F : C \to C$, F-algebra $(A, \alpha : F\ A \to A)$,

Homomorphisms between algebras

$h : (A, \alpha) \to (B, \beta)$ with
$$\begin{tikzcd} F\ A \arrow[r, "\alpha"] \arrow[d, "F\ h"] & A \arrow[d, "h"] \\ F\ B \arrow[r, "\beta"] & B \end{tikzcd}$$

Theoretical tools

Categories, functors and algebras

Category C, (endo-)functor $F : C \to C$, F-algebra $(A, \alpha : F\ A \to A)$,

Homomorphisms between algebras

$h : (A, \alpha) \to (B, \beta)$ with
$$\begin{array}{ccc} F\ A & \xrightarrow{\alpha} & A \\ \big\downarrow{\scriptstyle F\ h} & & \big\downarrow{\scriptstyle h} \\ F\ B & \xrightarrow{\beta} & B \end{array}$$

Catamorphisms

$(\!|\_|\!) : (F\ A \to A) \to (\mu F \to A)$ with
$$\begin{array}{ccc} F\ \mu F & \xrightarrow{inn} & \mu F \\ \big\downarrow{\scriptstyle F\ (\!|\alpha|\!)} & & \big\downarrow{\scriptstyle (\!|\alpha|\!)} \\ F\ A & \xrightarrow{\alpha} & A \end{array}$$

Implementing the theory ($cata = (\![\cdot]\!)$ in Haskell)

Catamorphisms towards implementation

$$
\begin{array}{ccc}
F\ \mu F & \xrightarrow{inn} & \mu F \\
\Big\downarrow{\scriptstyle F\,(\![\alpha]\!)} & & \Big\downarrow{\scriptstyle (\![\alpha]\!)} \\
F\ A & \xrightarrow{\alpha} & A
\end{array}
$$

Implementing the theory ($cata \;=\; (\![\cdot]\!)$ in Haskell)

Catamorphisms towards implementation

$$\begin{tikzcd} F\ \mu F \arrow[d, "F\ (\![\alpha]\!)"] & \mu F \arrow[l, "out"'] \arrow[d, "(\![\alpha]\!)"] \\ F\ A \arrow[r, "\alpha"] & A \end{tikzcd}$$

$\textbf{data}\ \mathit{Mu}\ f\ \ \textbf{where}$ -- Notation: $\mathit{Mu}\ f\ =\ \mu F$
$\quad \mathit{Inn} :: f\ (\mathit{Mu}\ f) \to \mathit{Mu}\ f$

$\mathit{out} :: \mathit{Mu}\ f \to f\ (\mathit{Mu}\ f)$ -- The inverse of Inn
$\mathit{out}\ (\mathit{Inn}\ x) = x$

$\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to (\mathit{Mu}\ f \to a)$
$\mathit{cata}\ \alpha = \alpha \circ \mathit{fmap}\ (\mathit{cata}\ \alpha) \circ \mathit{out}$

Implementing the theory ($cata = (\![\cdot]\!)$ in Haskell)

Catamorphisms towards implementation

$$
\begin{tikzcd}
F\ (F\ \mu F) \arrow[d, "F\ (F\ (\![\alpha]\!))"] & F\ \mu F \arrow[l, "F\ out"'] \arrow[d, "F\ (\![\alpha]\!)"] & \mu F \arrow[l, "out"'] \arrow[d, "(\![\alpha]\!)"] \\
F\ (F\ A) \arrow[r, "F\ \alpha"] & F\ A \arrow[r, "\alpha"] & A
\end{tikzcd}
$$

data $Mu\ f$ **where** -- Notation: $Mu\ f = \mu F$
  $Inn :: f\ (Mu\ f) \rightarrow Mu\ f$

$out :: Mu\ f \rightarrow f\ (Mu\ f)$ -- The inverse of Inn
$out\ (Inn\ x) = x$

$cata :: Functor\ f \Rightarrow (f\ a \rightarrow a) \rightarrow (Mu\ f \rightarrow a)$
$cata\ \alpha = \alpha \circ fmap\ (cata\ \alpha) \circ out$

Implementing the theory ($cata = (\![\cdot]\!)$ in Haskell)

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Mu}\ f\ \ \mathbf{where} \\
\quad \mathit{Inn} :: f\ (\mathit{Mu}\ f) \rightarrow \mathit{Mu}\ f \\
\mathit{out} :: \mathit{Mu}\ f \rightarrow f\ (\mathit{Mu}\ f) \\
\mathit{out}\ (\mathit{Inn}\ x) = x \\
\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ a \rightarrow a) \rightarrow (\mathit{Mu}\ f \rightarrow A) \\
\mathit{cata}\ \alpha = \alpha \circ \mathit{fmap}\ (\mathit{cata}\ \alpha) \circ \mathit{out}
\end{array}
$$

Example: *Mu FTree* is the datatype of binary trees with *Int* leaves.

$$
\begin{array}{l}
\mathbf{data}\ \mathit{FTree}\ \mathit{subtree}\ \ \mathbf{where} \\
\quad \mathit{Leaf} :: \mathit{Int} \rightarrow \mathit{FTree}\ \mathit{subtree} \\
\quad \mathit{Bin}\ \ :: \mathit{subtree} \rightarrow \mathit{subtree} \rightarrow \mathit{FTree}\ \mathit{subtree}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{class}\ \mathit{Category}\ \mathit{cat}\ \textbf{where} & \text{-- In the Haskell library } \mathit{Control.Category} \\
\quad \mathit{id}\ \ ::\ \mathit{cat}\ a\ a & \text{-- the identity arrow} \\
\quad (\circ)\ ::\ \mathit{cat}\ b\ c \rightarrow \mathit{cat}\ a\ b \rightarrow \mathit{cat}\ a\ c & \text{-- arrow composition} \\
\qquad \text{-- Identity laws: } \mathit{id} \circ p = p = p \circ \mathit{id} = p & \\
\qquad \text{-- Associativity: } (p \circ q) \circ r = p \circ (q \circ r) &
\end{array}
$$

Implementing the theory (arrows in Haskell)

```
class Category cat where      -- In the Haskell library Control.Category
  id  :: cat a a                                -- the identity arrow
  (∘) :: cat b c → cat a b → cat a c            -- arrow composition
    -- Identity laws: id ∘ p = p = p ∘ id = p
    -- Associativity: (p ∘ q) ∘ r = p ∘ (q ∘ r)
```

```
instance Category (→)     where id x = x; (f ∘ g) x = f (g x)
instance Category (SA s)  where   -- ...
data SA s a b = SA ((a, s) → (b, s))   -- “Stateful functions”
```

and many other instances.

Polytypic Data Conversion Programs

While John Hughes wrote "Generalising Monads to Arrows" [SCP'00] we used them for data conversion [SCP'02].

Motivation:

- save / load documents in editors should preserve the meaning
- but the source code for saving is not connected to that for loading
- proofs of pretty-print / parse round-trip properties are rare

Observations / contributions:

- we can describe both the saving and the loading using arrows
- we formalize the properties required
- we provide generic proofs of the round-trip properties

Polytypic Data Conversion Programs (cont.)

The starting point was separation of a datastructure (of type $d\ a$) into its shape ($d\ ()$) and contents ($[a]$).

$$
\begin{aligned}
&\mathit{separate} :: \mathit{Regular}\ d \ \Rightarrow\ \mathit{SA}\ [a]\ (d\ a)\ \ (d\ ())\\
&\mathit{separate}\ =\ \mathit{pmapAr}\ \mathit{put}\\
&\mathit{combine} :: \mathit{Regular}\ d \ \Rightarrow\ \mathit{SA}\ [a]\ (d\ ())\ (d\ a)\\
&\mathit{combine}\ =\ \mathit{pmapAl}\ \mathit{get}\\
&\mathit{put} ::\ \ \mathit{SA}\ [a]\ a\ \ ()\\
&\mathit{get} ::\ \ \mathit{SA}\ [a]\ ()\ a\\
&\mathit{put}\ =\ \mathit{SA}\ (\lambda\ (a,\ \ xs\ \ \ )\ \rightarrow\ ((),\ \ a : xs))\\
&\mathit{get}\ =\ \mathit{SA}\ (\lambda\ ((),\ \ a : xs)\ \rightarrow\ (a,\ \ xs\ \ \ ))
\end{aligned}
$$

Pedagogical development and leadership (2002–)

- 2002: Director of studies
- 2005: Vice Head of Department for education
- 2008: Deputy project leader of “Pedagogical development of Master’s Programmes for the Bologna Structure at Chalmers”
- 2011: Head of the 5-year education programme in Computer Science and Engineering (Civilingenjör Datateknik, Chalmers).
- 2013: Head of the Division of Software Technology

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell

  $\Rightarrow$ Agda,

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell

  $\Rightarrow$ Agda,

- on program correctness through types with Danielsson

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell

  $\Rightarrow$ Agda,
- on program correctness through types with Danielsson

  $\Rightarrow$ Fast'n Loose Reasoning,

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell
  $\Rightarrow$ Agda,
- on program correctness through types with Danielsson
  $\Rightarrow$ Fast'n Loose Reasoning, Chasing Bottoms, ...

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell
  $\Rightarrow$ Agda,
- on program correctness through types with Danielsson
  $\Rightarrow$ Fast'n Loose Reasoning, Chasing Bottoms, ...
- parametricity for dependent types & testing with Bernardy

My PhD graduates: Norell, Danielsson, and Bernardy

I worked on

- generic programs and proofs with Norell
  $\Rightarrow$ Agda,
- on program correctness through types with Danielsson
  $\Rightarrow$ Fast'n Loose Reasoning, Chasing Bottoms, ...
- parametricity for dependent types & testing with Bernardy
  Proofs for free:

$$[\![\_]\!] \; : \; PTS \to PTS$$

$$\Gamma \vdash A \; : \; B \;\; \Rightarrow \;\; [\![\, \Gamma \,]\!] \vdash [\![A]\!] \; : \; [\![B]\!] \; \overline{A}$$

where

$[\![A]\!]$ *is the free proof and*

$[\![B]\!] \; \overline{A}$ *is the free theorem*

and PTS = Pure Type System (Barendregt, et al.)

Global Systems Science (GSS)

with the Potsdam institute for Climate Impact Research (PIK)

- Collaboration from 2007 onwards (main contact: Cezar Ionescu)
- Aim: *correct* software models for simulating global systems

Global Systems Science (GSS)

with the Potsdam institute for Climate Impact Research (PIK)

- Collaboration from 2007 onwards (main contact: Cezar Ionescu)
- Aim: *correct* software models for simulating global systems
- Algebra of Programming [PhD course and two papers]
- Global Systems Dynamics and Policy (GSDP) [FET-Open 2010–13]
- Workshops including "Domain Specific Languages for Economical and Environmental Modelling", 2011

Global Systems Science (GSS)

with the Potsdam institute for Climate Impact Research (PIK)

- Collaboration from 2007 onwards (main contact: Cezar Ionescu)
- Aim: *correct* software models for simulating global systems
- Algebra of Programming [PhD course and two papers]
- Global Systems Dynamics and Policy (GSDP) [FET-Open 2010–13]
- Workshops including "Domain Specific Languages for Economical and Environmental Modelling", 2011
- The call **FETPROACT1** (Future and Emerging Technology, Proactive support for GSS) in Horizon 2020 is concrete evidence on the success of this line of work.
- Project GRACeFUL: "Global systems Rapid Assessment tools through Constraint FUnctional Languages" granted 2015–2018 by the FETPROACT1 call.

Global Systems Science (GSS)

with the Potsdam institute for Climate Impact Research (PIK)

- Collaboration from 2007 onwards (main contact: Cezar Ionescu)
- Aim: *correct* software models for simulating global systems
- Algebra of Programming [PhD course and two papers]
- Global Systems Dynamics and Policy (GSDP) [FET-Open 2010–13]
- Workshops including "Domain Specific Languages for Economical and Environmental Modelling", 2011
- The call **FETPROACT1** (Future and Emerging Technology, Proactive support for GSS) in Horizon 2020 is concrete evidence on the success of this line of work.
- Project GRACeFUL: "Global systems Rapid Assessment tools through Constraint FUnctional Languages" granted 2015–2018 by the FETPROACT1 call.
- Upcoming project CoEGSS: "Center of Excellence for Global Systems Science", start 2015-10-01, 3y.

Algebra of Programming in Agda

While Agda was implemented by Norell, Danielsson et al. we used it for the Algebra of Programming.
One highlight is the notation for equality proofs

```
begin
  term1
≡⟨ step1 ⟩    -- step1 : term1 ≡ term2
  term2
≡⟨ step2 ⟩    -- step2 : term2 ≡ term3
  term3
■
```

Roughly equivalent to *trans step1 step2* but often more readable (at least in more complicated cases).

An example proof in Agda, part 1

$$
\begin{array}{l}
\mathit{expLemma} : (x : \mathbb{R}) \to (m\ n : \mathbb{N}) \to (x \,\hat{}\, m *_{\mathbb{R}} x \,\hat{}\, n \equiv x \,\hat{}\, (m + n)) \\
\mathit{baseCase} : (x : \mathbb{R}) \to (n : \mathbb{N}) \to (x \,\hat{}\, Z *_{\mathbb{R}} x \,\hat{}\, n \equiv x \,\hat{}\, (Z + n)) \\
\mathit{stepCase} : (x : \mathbb{R}) \to (m\ n : \mathbb{N}) \to \\
\qquad (\mathit{ih} : x \,\hat{}\, m *_{\mathbb{R}} x \,\hat{}\, n \equiv x \,\hat{}\, (m + n)) \to \\
\qquad (x \,\hat{}\, (S\ m) *_{\mathbb{R}} x \,\hat{}\, n \equiv x \,\hat{}\, ((S\ m) + n))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{expLemma}\ x\ Z\ n & = & \mathit{baseCase}\ x\ n \\
\mathit{expLemma}\ x\ (S\ m)\ n & = & \mathit{stepCase}\ x\ m\ n\ (\mathit{expLemma}\ x\ m\ n)
\end{array}
$$

An example proof in Agda, part 2

```
baseCase : (x : ℝ) → (n : ℕ) → (x ^ Z *ℝ x ^ n ≡ x ^ (Z + n))
baseCase x n =
  begin
    x ^ Z    *ℝ  x ^ n
  ≡⟨ refl ⟩                  -- By definition of _^_
    one  *ℝ  x ^ n
  ≡⟨ unitMult (x ^ n) ⟩      -- Use one *ℝ y = y for y = x ^ n
    x ^ n
  ≡⟨ refl ⟩                  -- By definition of _+_
    x ^ (Z + n)
  ∎
```

Feat: functional enumeration of algebraic types
[with Duregård and Wang, Haskell Symposium 2012]

An efficiently computable bijective function $index_a :: \mathbb{N} \rightarrow a$, much like *toEnum* in the *Enum* class.

Example: enumerate "raw abstract syntax trees" for Haskell.

```
*Main> index (10^5) :: Exp
AppE (LitE (StringL ""))
     (CondE (ListE []) (ListE []) (LitE (IntegerL 1)))
```

Feat: functional enumeration of algebraic types
[with Duregård and Wang, Haskell Symposium 2012]

An efficiently computable bijective function $index_a :: \mathbb{N} \rightarrow a$, much like *toEnum* in the *Enum* class.

Example: enumerate "raw abstract syntax trees" for Haskell.

```
*Main> index (10^5) :: Exp
AppE (LitE (StringL ""))
     (CondE (ListE []) (ListE []) (LitE (IntegerL 1)))
```

```
*Main> index (10^100) :: Exp
ArithSeqE (FromR (AppE (AppE (ArithSeqE (FromR (ListE [])))
... -- and 20 more lines!
```

Ongoing work

DSLM: Presenting Math. Analysis Using Functional Programming

$$\forall\ \epsilon\ \in\ \mathbb{R}.\ \ (\epsilon > 0)\ \Rightarrow\ \exists\ a\ \in\ A.\ \ (|a - sup\ A| < \epsilon)$$

Sequential Decision Problems

"Sequential Decision Problems, dependent types and generic solutions"
"A computational theory of policy advice and avoidability"

AUTOSAR calculus

"A semantics of core AUTOSAR"
(AUTOSAR = AUTomotive Open System ARchitecture)

ValiantAgda

Certified Context-Free Parsing: A form. of Valiant's Algorithm in Agda
Solve $C\ =\ W + C * C$ for matrices of sets of non-terminals!

Solve $C = W + C * C$ for strictly upper triangular matrices of something.

ValiantAgda (the chocolate part;-)

Solve $C = W + C * C$ for strictly upper triangular matrices of something.

sages Search Twitter

Patrik Jansson
@patrikja

Applied algebra: an upper triangular matrix. Eat the diagonal to make it strictly upper triangular. #Agda #chocolate

ages Search Twitter

Patrik Jansson @patrikja · 9 Dec 2013
@patrikja Strictly upper triangular matrix. Aim: use these #chocolate block matrices to explain Valiants alg. #Agda

ValiantAgda (a part in the middle)

[Valiant, 1975] provides a rather awkward interated def. for all bracketings:

$$\begin{aligned} C \;=\; & W + W \cdot W + W \cdot (W \cdot W) + (W \cdot W) \cdot W + \\ & (W \cdot W) \cdot (W \cdot W) + \ldots \end{aligned}$$

We use the smallest solution to the following equation:

$$C \;\equiv\; W + C \cdot C$$

(for strictly upper triangular W). Or more precisely

$$\begin{aligned} & \mathit{Clo} : U \to U \to \mathit{Set} \\ & \mathit{Clo}\; W \;\; C = \quad C \equiv W + C \cdot C \end{aligned}$$

$$\begin{aligned} & \mathit{LowerBound} : \{A : \mathit{Set}\} \to (A \to \mathit{Set}) \to A \to \mathit{Set} \\ & \mathit{LowerBound}\; P\; x = \forall\, z \to (P\; z \to x \leqslant z) \end{aligned}$$

$$\begin{aligned} & \mathit{Minimal} : \{A : \mathit{Set}\} \to (A \to \mathit{Set}) \to A \to \mathit{Set} \\ & \mathit{Minimal}\; P\; x = P\; x \wedge \mathit{LowerBound}\; P\; x \end{aligned}$$

$$\begin{aligned} \mathit{Spec} = {} & \forall\, (W : U) \to \exists\, \lambda\, (C : U) \to \\ & \mathit{Minimal}\; (\mathit{Clo}\; W)\; C \end{aligned}$$

Summary

$$
\begin{tikzcd}
PolyP \arrow[r] \arrow[d] & DataConv & AoPAgda \arrow[r] & ValiantAgda \\
Norell \arrow[d] & Danielsson \arrow[dl] \arrow[d] & Bernardy \arrow[ur] \arrow[d] & Duregård \arrow[d] \\
Agda \arrow[uurr, bend left] & Fast'nLoose & ParaDep & Feat
\end{tikzcd}
$$

Summary

$$
\begin{tikzcd}
PolyP \arrow[r] \arrow[d] & DataConv & AoPAgda \arrow[r] & ValiantAgda \\
Norell \arrow[d] & Danielsson \arrow[dl] \arrow[d] & Bernardy \arrow[ur] \arrow[d] & Duregård \arrow[d] \\
Agda \arrow[uurr, bend left] & Fast'nLoose & ParaDep & Feat
\end{tikzcd}
$$

$$
DoS \longrightarrow VPref.Gru \xrightarrow{Bologna} PA@D \longrightarrow DivHead
$$

Summary

$$
\begin{tikzcd}
PolyP \arrow[r] \arrow[d] & DataConv & AoPAgda \arrow[r] & ValiantAgda \\
Norell \arrow[d] & Danielsson \arrow[dl] \arrow[d] & Bernardy \arrow[ur] \arrow[d] & Duregård \arrow[d] \\
Agda \arrow[uurr, bend left] & Fast'nLoose & ParaDep & Feat
\end{tikzcd}
$$

$$
\begin{tikzcd}
DoS \arrow[r] & VPref.Gru \arrow[r, "Bologna"] & PA@D \arrow[r] & DivHead \\
PIK \arrow[r] \arrow[dr] & GSDP \arrow[r] \arrow[d] \arrow[rr, bend left] & GRACeFUL & CoEGSS \\
 & Ionescu \arrow[ur] \arrow[r] & DSLsofMath &
\end{tikzcd}
$$

R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer, 1999. URL `http://www.cse.chalmers.se/~patrikj/poly/afp98/`.

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. of the 15th ACM SIGPLAN international conference on Funct. programming*, pages 345–356, Baltimore, Maryland, 2010. ACM. doi: 10.1145/1863543.1863592.

N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *POPL'06*, pages 206–217. ACM Press, 2006. doi: 10.1145/1111037.1111056.

J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Haskell'12*, pages 61–72. ACM, 2012. doi: 10.1145/2364506.2364515.

C. Jaeger, P. Jansson, S. van der Leeuw, M. Resch, and J. D. Tabara. GSS: Towards a research program for Global Systems Science. `http://blog.global-systems-science.eu/?p=1512`, 2013. ISBN 978.3.94.1663-12-1. Conference Version, prepared for the Second Open Global Systems Science Conference June 10-12, 2013, Brussels.

P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Proc. POPL'97: Principles of Programming Languages*, pages 470–482. ACM Press, 1997. doi: 10.1145/263699.263763.

P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1): 35–75, 2002. doi: 10.1016/S0167-6423(01)00020-X.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579, 2009. doi: 10.1017/S0956796809007345.