

Positively Dependent Types

Dan Licata and Robert Harper
Carnegie Mellon University

Dan's thesis

New dependently typed programming language
for programming with **binding and scope**

Applications:

- * Domain-specific logics for reasoning about code
- * Mechanized metatheory

Dan's thesis

New dependently typed programming language
for programming with **binding and scope**

Applications:

- * Domain-specific logics for reasoning about code
- * Mechanized metatheory

Based on polarized type theory

Polarity [Girard '93]

Sums $A + B$ are **positive data**:

- * Introduced by choosing `inl` or `inr`
- * Eliminated by pattern-matching

ML functions $A \rightarrow B$ are **negative computation**:

- * Introduced by pattern-matching on `A`
- * Eliminated by choosing an `A` to apply to

Focusing [Andreoli '92]

Sums $A + B$ are **positive data**:

- * Introduced by **choosing** inl or inr
- * Eliminated by pattern-matching

ML functions $A \rightarrow B$ are **negative computation**:

- * Introduced by pattern-matching on A
- * Eliminated by **choosing** an A to apply to

Focusing [Andreoli '92]

Sums $A + B$ are **positive data**:

- * Introduced by **choosing** inl or inr
- * Eliminated by pattern-matching

**Focus =
make choices**

ML functions $A \rightarrow B$ are **negative computation**:

- * Introduced by pattern-matching on A
- * Eliminated by **choosing** an A to apply to

Focusing [Andreoli '92]

Sums $A + B$ are **positive data**:

- * Introduced by choosing `inl` or `inr`
- * Eliminated by **pattern-matching**

ML functions $A \rightarrow B$ are **negative computation**:

- * Introduced by **pattern-matching** on A
- * Eliminated by choosing an A to apply to

Focusing [Andreoli '92]

Sums $A + B$ are **positive data**:

- * Introduced by choosing `inl` or `inr`
- * Eliminated by **pattern-matching**

**Inversion =
respond to all
possible choices**

ML functions $A \rightarrow B$ are **negative computation**:

- * Introduced by **pattern-matching** on A
- * Eliminated by choosing an A to apply to

Higher-order focusing

Zeilberger's higher-order formalism:

- * Type theory organized around distinction between positive data and negative computation

Positive Data

products (eager)
sums
natural numbers
inductive types

Negative Computation

products (lazy)
functions
streams
coinductive types

Higher-order focusing

Zeilberger's *higher-order formalism*:

- * Type theory organized around distinction between positive data and negative computation
- * Pattern matching represented abstractly by *meta-level functions* from patterns to expressions, using an iterated inductive definition

Higher-order focusing

Applications so far:

Higher-order focusing

Applications so far:

- * Curry-Howard for pattern matching
[Zeilberger POPL'08; cf. Krishnaswami POPL'09]

Higher-order focusing

Applications so far:

- * Curry-Howard for pattern matching
[Zeilberger POPL'08; cf. Krishnaswami POPL'09]
- * Logical account of evaluation order
[Zeilberger APAL]

Higher-order focusing

Applications so far:

- * Curry-Howard for pattern matching
[Zeilberger POPL'08; cf. Krishnaswami POPL'09]
- * Logical account of evaluation order
[Zeilberger APAL]
- * Analysis of operationally sensitive
typing phenomena [Zeilberger PLPV'09]

Higher-order focusing

Applications so far:

- * Curry-Howard for pattern matching
[Zeilberger POPL'08; cf. Krishnaswami POPL'09]
- * Logical account of evaluation order
[Zeilberger APAL]
- * Analysis of operationally sensitive
typing phenomena [Zeilberger PLPV'09]
- * **Positive** function space for
representing variable binding [LZH LICS'08]

Positive function space

- * Permits LF-style representation of binding:
framework provides α -equivalence, substitution
- * Eliminated by pattern matching =
structural induction modulo α

Positive function space

- * Permits LF-style representation of binding:
framework provides α -equivalence, substitution
- * Eliminated by pattern matching =
structural induction modulo α

But no dependent types...

Positively dependent types

Contributions:

1. Extend higher-order focusing with a simple form of dependency
2. Formalize the language in Agda

Positively dependent types

Key idea: Allow dependency on **positive data**,
but not **negative computation**

Positively dependent types

Key idea: Allow dependency on **positive data**,
but not **negative computation**

Enough for simple applications:

- * Lists indexed by their lengths ($\text{Vec}[n:\text{nat}]$)
- * Judgements on higher-order abstract syntax represented with **positive** functions

Positively dependent types

Key idea: Allow dependency on **positive data**,
but not **negative computation**

Avoids complications of **negative** dependency:

- * Equality is easy for **data**, hard for **computation**
- * **Computations** are free to be effectful

Positively dependent types

1. Type and term levels share the same **data**
(like Agda, Epigram, Cayenne, NuPRL, ...)
2. But have different notions of **computation**
(like DML, Omega, ATS, ...)

Polarized type theory

Intuitionistic logic:

$A^+ ::= \text{nat} \mid A^+ \otimes B^+ \mid 1 \mid A^+ \oplus B^+ \mid 0 \mid \downarrow A^-$

$A^- ::= A^+ \rightarrow B^- \mid A^- \& B^- \mid \top \mid \uparrow A^+$

Polarized type theory

Intuitionistic logic:

$A^+ ::= \text{nat} \mid A^+ \otimes B^+ \mid 1 \mid A^+ \oplus B^+ \mid 0 \mid \downarrow A^-$

$A^- ::= A^+ \rightarrow B^- \mid A^- \& B^- \mid \top \mid \uparrow A^+$

Allow dependency on values of
purely positive types (no $\downarrow A^-$)

Polarized type theory

Intuitionistic logic (see paper):

$$A^+ ::= \text{nat} \mid A^+ \otimes B^+ \mid 1 \mid A^+ \oplus B^+ \mid 0 \mid \downarrow A^-$$
$$A^- ::= A^+ \rightarrow B^- \mid A^- \& B^- \mid \top \mid \uparrow A^+$$

Minimal logic (this talk):

$$A^+ ::= \text{nat} \mid A^+ \otimes B^+ \mid 1 \mid A^+ \oplus B^+ \mid 0 \mid \neg A^+$$

Purely positive types: no $\neg A^+$ (= $\downarrow(A^+ \rightarrow \#)$)

Outline

1. Simply typed higher-order focusing
2. Positively dependent types

Outline

- 1. Simply typed higher-order focusing**
2. Positively dependent types

Higher-order focusing

- * Specify types by their patterns
- * Type-independent focusing framework
 - * Focus phase = choose a pattern
 - * Inversion phase = pattern-matching

Higher-order focusing

- ✱ **Specify types by their patterns**
- ✱ Type-independent focusing framework
 - ✱ Focus phase = choose a pattern
 - ✱ Inversion phase = pattern-matching

Patterns

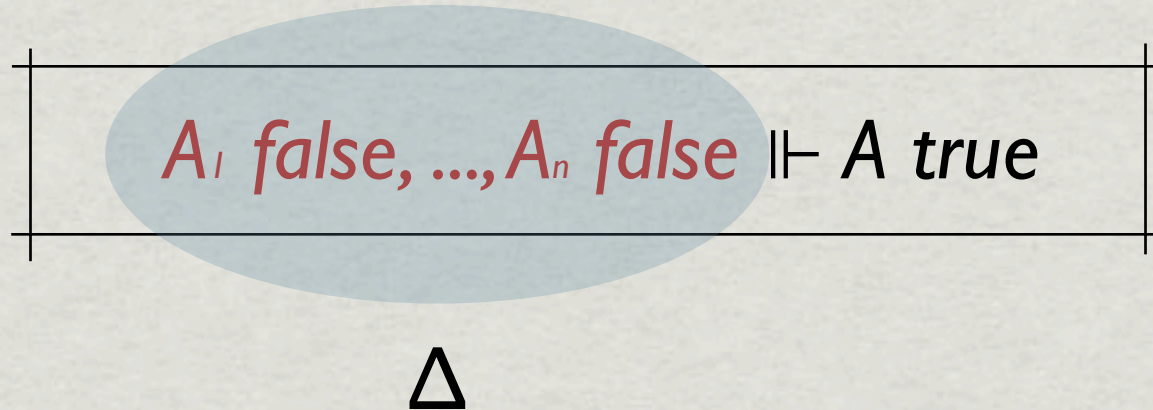
Proof pattern gives us the outline of a proof, but leaves holes for refutations

$$\frac{\frac{\frac{A \text{ false}}{\neg A \text{ true}}}{\neg A \otimes (\neg B \oplus \neg C) \text{ true}} \quad \frac{\frac{B \text{ false}}{\neg B \text{ true}}}{\neg B \oplus \neg C \text{ true}}}{\neg A \otimes (\neg B \oplus \neg C) \text{ true}}$$

Patterns

$$A_1 \text{ false}, \dots, A_n \text{ false} \Vdash A \text{ true}$$

Patterns



$\Delta \vdash A \text{ true}$: there is a proof pattern for A ,
leaving holes for refutations of $A_1 \dots A_n$

Pattern rules

A false $\Vdash \neg A$ *true*

$\Delta_1 \Vdash A$ *true* $\Delta_2 \Vdash B$ *true*

$\Delta_1 \Delta_2 \Vdash A \otimes B$ *true*

$\cdot \Vdash I$ *true*

$\Delta \Vdash A$ *true*

$\Delta \Vdash A \oplus B$ *true*

$\Delta \Vdash B$ *true*

$\Delta \Vdash A \oplus B$ *true*

(no rule for 0)

Proof terms

$$\frac{\frac{}{A \text{ false} \Vdash \neg A \text{ true}} \quad \frac{\frac{}{B \text{ false} \Vdash \neg B \text{ true}}}{B \text{ false} \Vdash \neg B \oplus \neg C \text{ true}}}{A \text{ false}, B \text{ false} \Vdash \neg A \otimes (\neg B \oplus \neg C) \text{ true}}$$

\cong

$(K_1, \text{inl } K_2)$

continuation variables

Proof terms

$$\frac{\frac{\frac{K_1}{A \text{ false} \Vdash \neg A \text{ true}}{B \text{ false} \Vdash \neg B \oplus \neg C \text{ true}}{B \text{ false} \Vdash \neg B \text{ true}} \quad K_2}{(_ , _)} \text{ inl}}{A \text{ false}, B \text{ false} \Vdash \neg A \otimes (\neg B \oplus \neg C) \text{ true}}$$

\cong

$(K_1, \text{inl } K_2)$

continuation variables

Higher-order focusing

- * Specify types by their patterns
- * **Type-independent focusing framework**
 - * Focus phase = choose a pattern
 - * Inversion phase = pattern-matching

Focused proofs

iterated inductive definition

$$\frac{\Delta \Vdash A \text{ true} \quad \Gamma \vdash \Delta}{\Gamma \vdash A \text{ true}}$$

$$\frac{\Delta \Vdash A \text{ true} \longrightarrow \Gamma, \Delta \vdash \#}{\Gamma \vdash A \text{ false}}$$

$$\frac{A \text{ false} \in \Delta \longrightarrow \Gamma \vdash A \text{ false}}{\Gamma \vdash \Delta}$$

$$\frac{A \text{ false} \in \Delta \quad \Gamma \vdash A \text{ true}}{\Gamma \vdash \#}$$

Focused proofs

iterated inductive definition

$$\frac{\Delta \Vdash A \text{ true} \quad \Gamma \vdash \Delta}{\Gamma \vdash A \text{ true}}$$

$$\Gamma \vdash A \text{ true}$$

focus

$$\frac{\Delta \Vdash A \text{ true} \longrightarrow \Gamma, \Delta \vdash \#}{\Gamma \vdash A \text{ false}}$$

$$\Gamma \vdash A \text{ false}$$

inversion

$$\frac{A \text{ false} \in \Delta \longrightarrow \Gamma \vdash A \text{ false}}{\Gamma \vdash \Delta}$$

$$\Gamma \vdash \Delta$$

$$\frac{A \text{ false} \in \Delta \quad \Gamma \vdash A \text{ true}}{\Gamma \vdash \#}$$

$$\Gamma \vdash \#$$

Example continuation

$$\text{K deriv. of } \frac{\Delta \Vdash \neg A \otimes (\neg B \oplus \neg C) \text{ true} \quad \xrightarrow{E} \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \neg A \otimes (\neg B \oplus \neg C) \text{ false}}$$

p (green) above the top line, *E* (blue) above the arrow.

Example continuation

$$\text{K deriv. of } \frac{\Delta \Vdash \neg A \otimes (\neg B \oplus \neg C) \text{ true} \quad \overset{p}{\phantom{\Delta \Vdash \neg A \otimes (\neg B \oplus \neg C) \text{ true}}} \quad \overset{E}{}}{\Gamma \vdash \neg A \otimes (\neg B \oplus \neg C) \text{ false}}$$

$$\text{K} \left\{ \begin{array}{l}
 (\text{K}_1, \text{inl } \text{K}_2) \mapsto \overset{E_1}{\Gamma, \text{K}_1:A \text{ false}, \text{K}_2: B \text{ false}, \vdash \#} \\
 (\text{K}_1, \text{inr } \text{K}_3) \mapsto \overset{E_2}{\Gamma, \text{K}_1:A \text{ false}, \text{K}_3: C \text{ false}, \vdash \#}
 \end{array} \right.$$


Outline

1. Simply typed higher-order focusing

2. Positively dependent types

Higher-order focusing

all the changes are here

- * Specify types by their patterns 
- * Type-independent focusing framework
 - * Focus phase = choose a pattern
 - * Inversion phase = pattern-matching

Positively dependent types

1. Allow indexing by closed patterns
= values of purely positive types

Patterns

nat: $\frac{}{\cdot \Vdash \text{nat } \textit{true}}$ z $\frac{\Delta \Vdash \text{nat } \textit{true}}{\Delta \Vdash \text{nat } \textit{true}}$ s

Patterns

nat: $\frac{}{\cdot \Vdash \text{nat } \textit{true}}$ *z* $\frac{\Delta \Vdash \text{nat } \textit{true}}{\Delta \Vdash \text{nat } \textit{true}}$ *s*

vec[$p :: \cdot \Vdash \text{nat } \textit{true}$]:

$\frac{}{\cdot \Vdash \text{vec}[z] \textit{true}}$ *nil*

$\frac{\Delta_1 \Vdash \text{bool } \textit{true} \quad \Delta_2 \Vdash \text{vec}[p] \textit{true}}{\Delta_1 \Delta_2 \Vdash \text{vec}[s p] \textit{true}}$ *cons*

Positively dependent types

1. Allow indexing by closed patterns
= values of purely positive types
2. Syntax of $(\Sigma x:A.B)$ specified by **pattern-matching**:
gives type-level computation (large eliminations)

Dependent pairs

$$\frac{A \text{ type} \quad [] \Vdash A \text{ true} \xrightarrow{p} \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$

Dependent pairs

$$\frac{A \text{ type} \quad \square \Vdash A \text{ true} \xrightarrow{p} \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$

$$\frac{\cdot \Vdash A \text{ true} \quad \Delta \Vdash \tau(p) \text{ true}}{\Delta \Vdash \Sigma A \tau \text{ true}} \text{ pair}$$

Dependent pairs

List: $\Sigma \text{ nat } (p \mapsto \text{vec}[p])$

Pattern: $(\text{pair } \mathcal{Z} (\text{cons true } (\text{cons false nil})))$

$$\frac{A \text{ type} \quad \square \Vdash^p A \text{ true} \longrightarrow \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$

$$\frac{\cdot \Vdash^p A \text{ true} \quad \Delta \Vdash \tau(p) \text{ true}}{\Delta \Vdash \Sigma A \tau \text{ true}} \text{ pair}$$

Dependent pairs

Check: $\Sigma \text{bool} (\text{true} \mapsto 1; \text{false} \mapsto 0)$

Only pattern: $\text{pair true} \langle \rangle$

$$\frac{A \text{ type} \quad [] \Vdash A \text{ true} \xrightarrow{p} \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$

$$\frac{\cdot \Vdash A \text{ true} \quad \Delta \Vdash \tau(p) \text{ true}}{\Delta \Vdash \Sigma A \tau \text{ true}} \text{pair}$$

Dependent pairs

Recursive Vec: $\Sigma \text{ nat } (z \mapsto \mathbf{1};$
 $s(z) \mapsto \mathbf{bool};$
 $s(s(z)) \mapsto \mathbf{bool} \otimes \mathbf{bool};$
 $\dots)$

$$\frac{A \text{ type} \quad \square \Vdash A \text{ true} \longrightarrow \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$

$$\frac{\bullet \Vdash A \text{ true} \quad \Delta \Vdash \tau(p) \text{ true}}{\Delta \Vdash \Sigma A \tau \text{ true}} \text{ pair}$$

Dependent pairs

Logical relations: define predicate by recursion on representation of object-language type

$$\frac{A \text{ type} \quad \square \Vdash^p A \text{ true} \longrightarrow \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$
$$\frac{\bullet \Vdash^p A \text{ true} \quad \Delta \Vdash \tau(p) \text{ true}}{\Delta \Vdash \Sigma A \tau \text{ true}} \text{ pair}$$

Well-defined?

Well-defined?

1. Simply-typed: Iterated inductive definition
 - Patterns defined first
 - Pattern-matching quantifies over them

Well-defined?

1. Simply-typed: Iterated inductive definition
 - Patterns defined first
 - Pattern-matching quantifies over them
2. Dependent: Mutual definition
 - Patterns classified by types
 - $\Sigma A \tau$ quantifies over patterns

Well-defined?

1. Simply-typed: Iterated inductive definition
 - Patterns defined first
 - Pattern-matching quantifies over them
2. Dependent: Mutual definition
 - Patterns classified by types
 - $\Sigma A \tau$ quantifies over patterns

Why does this make sense?

Induction-Recursion

1. **Inductively** define the syntax of positive types

$$A^+ ::= A^+ \otimes B^+ \mid 1 \mid A^+ \oplus B^+ \mid 0 \mid \neg A^+ \\ \mid \text{nat} \mid \text{vec}[p] \mid \Sigma A^+ \tau$$

2. Simultaneously, **recursively** define patterns for A^+

$$\Delta \Vdash A \oplus B \text{ =def= } \text{Either } (\Delta \Vdash A) (\Delta \Vdash B)$$

Induction-Recursion

$$\frac{A \text{ type} \quad [] \Vdash A \text{ true} \xrightarrow{p} \tau(p) \text{ type}}{\Sigma A \tau \text{ type}}$$

τ quantifies over type A , which is **smaller than** $\Sigma A \tau$

1. Define the type A
2. Define the patterns for A
3. Define the types $\Sigma A \tau$ (quantifies over pats for A)
4. Define the patterns for $\Sigma A \tau$
5.

Example

$head :: (\Sigma \text{ nat } (n \mapsto \text{vec}[s \ n])) \rightarrow \text{bool}$

Example

$head :: (\Sigma \text{ nat } (n \mapsto \text{vec}[s \ n])) \rightarrow \text{bool}$

...contrapositive...

$head :: (\kappa : \text{bool } \text{false}) \vdash \Sigma \text{ nat } (n \mapsto \text{vec}[s \ n]) \ \text{false}$

Example

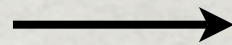
$head :: (\Sigma \text{ nat } (n \mapsto \text{vec}[s \ n])) \rightarrow \text{bool}$

...contrapositive...

$head :: (\kappa : \text{bool } \text{false}) \vdash \Sigma \text{ nat } (n \mapsto \text{vec}[s \ n]) \text{ false}$

...one premise...

$head :: \Delta \Vdash \Sigma \text{ nat } (n \mapsto \text{vec}[s \ n]) \text{ true}$



$(\kappa : \text{bool } \text{false}), \Delta \vdash \#$

Example

$$\begin{array}{c} \text{head} :: \Delta \Vdash \Sigma \text{ nat } (n \mapsto \text{vec}[s \ n]) \text{ true} \\ \longrightarrow \\ (\kappa : \text{bool } \text{false}), \Delta \vdash \# \end{array}$$
$$\text{head } (\text{pair } _ (\text{cons } x \ _)) \mapsto \text{throw } x \text{ to } \kappa$$

(no case for $\text{head } (\text{pair } n \ \text{nil})$!)

See Paper

- * Agda encoding
- * Examples coded using Agda representation
- * Discussion of type equality
 - * Types are equal iff they have the same patterns:
induces an identity coercion
 - * $(\Sigma A \tau) = (\Sigma A' \tau')$:
compare τ and τ' extensionally

Positively dependent types

Contributions:

1. Extend higher-order focusing with a simple form of dependency
2. Formalize the language in Agda

Positively dependent types

1. Type and term levels share the same **data**

2. But different notions of **computation**

- Terms: Pattern-match results in $E :: \Gamma \vdash \#$
(can add effects to this judgement)
- Types: Pattern-match τ results in types
(pure)

Future work

- * Integrate with LICS work on **variable binding**
- * Implement **positively** dependent types in GHC or ML
- * **Negatively** dependent types, too?

Thanks for listening!