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 $A \to B$ 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 $A \to B$ 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 $A \to B$ 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 $A \to B$ 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 $A \to B$ 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	Negative Computation
products (eager)	products (lazy)
sums	functions
natural numbers	streams
inductive types	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 (Vec[n: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:

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

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

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):

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

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

Minimal logic (this talk):

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

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

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

$$\dfrac{\dfrac{A\ \mathit{false}}{\neg A\ \mathit{true}} \qquad \dfrac{\dfrac{B\ \mathit{false}}{\neg B\ \mathit{true}}}{\neg B \oplus \neg C\ \mathit{true}}}{\neg A \otimes (\neg B \oplus \neg C)\ \mathit{true}}$$

Patterns

$$\boxed{A_1 \ \mathit{false}, \ldots, A_n \ \mathit{false} \Vdash A \ \mathit{true}}$$

Patterns

$$\boxed{\underbrace{A_1\ \mathit{false}, \ldots, A_n\ \mathit{false}}_{\Delta} \Vdash A\ \mathit{true}}$$

$\Delta \Vdash A\ \mathit{true}$: there is a proof pattern for A, leaving holes for refutations of $A_1 \ldots A_n$

Pattern rules

$$\frac{}{A\ \mathit{false} \Vdash \neg A\ \mathit{true}}$$

$$\frac{\Delta_1 \Vdash A\ \mathit{true} \qquad \Delta_2 \Vdash B\ \mathit{true}}{\Delta_1\Delta_2 \Vdash A \otimes B\ \mathit{true}} \qquad \frac{}{\cdot \Vdash I\ \mathit{true}}$$

$$\frac{\Delta \Vdash A\ \mathit{true}}{\Delta \Vdash A \oplus B\ \mathit{true}} \qquad \frac{\Delta \Vdash B\ \mathit{true}}{\Delta \Vdash A \oplus B\ \mathit{true}}$$

(no rule for 0)

Proof terms

$$\frac{\dfrac{}{A\ \textit{false} \Vdash \neg A\ \textit{true}} \qquad \dfrac{\dfrac{}{B\ \textit{false} \Vdash \neg B\ \textit{true}}}{B\ \textit{false} \Vdash \neg B \oplus \neg C\ \textit{true}}}{A\ \textit{false}, B\ \textit{false} \Vdash \neg A \otimes (\neg B \oplus \neg C)\ \textit{true}}$$

$$\approx$$

$$(\kappa_1, \mathrm{inl}\ \kappa_2)$$

continuation variables

Proof terms

$$\dfrac{\kappa_1\,\dfrac{}{A\ \mathit{false} \Vdash \neg A\ \mathit{true}} \qquad \dfrac{\dfrac{}{B\ \mathit{false} \Vdash \neg B\ \mathit{true}}\,\kappa_2}{B\ \mathit{false} \Vdash \neg B \oplus \neg C\ \mathit{true}}\,\mathrm{inl}}{A\ \mathit{false}, B\ \mathit{false} \Vdash \neg A \otimes (\neg B \oplus \neg C)\ \mathit{true}}\,(\_,\_)$$

$$\approx$$

$$(\kappa_1, \mathrm{inl}\ \kappa_2)$$

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

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

$$\frac{A\ \mathit{false} \in \Delta \longrightarrow \Gamma \vdash A\ \mathit{false}}{\Gamma \vdash \Delta} \qquad\qquad \frac{A\ \mathit{false} \in \Delta \qquad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash \#}$$

Focused proofs

$$\frac{\Delta \Vdash A\ \mathit{true} \qquad \Gamma \vdash \Delta}{\Gamma \vdash A\ \mathit{true}}$$

focus

iterated inductive definition

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

inversion

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

$$\frac{A\ \mathit{false} \in \Delta \qquad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash \#}$$

Example continuation

K deriv. of $\dfrac{\overset{p}{\Delta \Vdash \neg A \otimes (\neg B \oplus \neg C)\ \mathit{true}} \quad \longrightarrow \quad \overset{E}{\Gamma, \Delta \vdash \#}}{\Gamma \vdash \neg A \otimes (\neg B \oplus \neg C)\ \mathit{false}}$

Example continuation

$$\text{K deriv. of} \quad \dfrac{\overset{p}{\Delta \Vdash \neg A \otimes (\neg B \oplus \neg C)\ \mathit{true}} \quad \longrightarrow \quad \overset{E}{\Gamma, \Delta \vdash \#}}{\Gamma \vdash \neg A \otimes (\neg B \oplus \neg C)\ \mathit{false}}$$

$$K \begin{cases} (\kappa_1, \mathrm{inl}\ \kappa_2) \mapsto & \overset{E_1}{\Gamma, \kappa_1{:}A\ \mathit{false}, \kappa_2{:}\ B\ \mathit{false}, \vdash \#} \\ (\kappa_1, \mathrm{inr}\ \kappa_3) \mapsto & \overset{E_2}{\Gamma, \kappa_1{:}A\ \mathit{false}, \kappa_3{:}\ C\ \mathit{false}, \vdash \#} \end{cases}$$

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: $\dfrac{}{\cdot \Vdash \mathtt{nat}\ \mathit{true}}\ \mathtt{z} \qquad \dfrac{\Delta \Vdash \mathtt{nat}\ \mathit{true}}{\Delta \Vdash \mathtt{nat}\ \mathit{true}}\ \mathtt{s}$

Patterns

nat:

$$\frac{}{\cdot \Vdash \mathrm{nat}\ \mathit{true}}\ \mathrm{z} \qquad \frac{\Delta \Vdash \mathrm{nat}\ \mathit{true}}{\Delta \Vdash \mathrm{nat}\ \mathit{true}}\ \mathrm{s}$$

vec[p :: $\cdot \Vdash \mathrm{nat}\ \mathit{true}$]:

$$\frac{}{\cdot \Vdash \mathrm{vec}[\mathrm{z}]\ \mathit{true}}\ \mathrm{nil}$$

$$\frac{\Delta_1 \Vdash \mathrm{bool}\ \mathit{true} \qquad \Delta_2 \Vdash \mathrm{vec}[p]\ \mathit{true}}{\Delta_1 \Delta_2 \Vdash \mathrm{vec}[\mathrm{s}\ p]\ \mathit{true}}\ \mathrm{cons}$$

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

$$\frac{A\ \mathit{type} \qquad\qquad [\,] \Vdash A\ \mathit{true} \xrightarrow{\;p\;} \tau(p)\ \mathit{type}}{\Sigma\, A\, \tau\ \mathit{type}}$$

Dependent pairs

$$\frac{A\ \mathit{type} \qquad [] \Vdash \overset{p}{A}\ \mathit{true} \longrightarrow \tau(p)\ \mathit{type}}{\Sigma\, A\, \tau\ \mathit{type}}$$

$$\frac{\cdot \Vdash \overset{p}{A}\ \mathit{true} \qquad \Delta \Vdash \tau(p)\ \mathit{true}}{\Delta \Vdash \Sigma\, A\, \tau\ \ \mathit{true}}\ \text{pair}$$

Dependent pairs

List: Σ nat (p $\mapsto$ vec[p])

Pattern: (pair 2 (cons true (cons false nil)))

$$\frac{A\ \mathit{type} \qquad [\,] \Vdash \overset{p}{A}\ \mathit{true} \longrightarrow \tau(p)\ \mathit{type}}{\Sigma\, A\, \tau\ \mathit{type}}$$

$$\frac{\cdot \Vdash \overset{p}{A}\ \mathit{true} \qquad \Delta \Vdash \tau(p)\ \mathit{true}}{\Delta \Vdash \Sigma\, A\, \tau\ \ \mathit{true}}\ \text{pair}$$

Dependent pairs

Check: Σ bool (true ↦ 1; false ↦ 0)

Only pattern: pair true <>

$$\frac{A \textit{ type} \qquad [\,] \Vdash \overset{p}{A} \textit{ true} \longrightarrow \tau(p) \textit{ type}}{\Sigma\, A\, \tau \textit{ type}}$$

$$\frac{\cdot \Vdash \overset{p}{A} \textit{ true} \qquad \Delta \Vdash \tau(p) \textit{ true}}{\Delta \Vdash \Sigma\, A\, \tau \;\; \textit{true}}\ \text{pair}$$

Dependent pairs

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

$$\frac{A\ \mathit{type} \qquad [] \Vdash A\ \mathit{true} \overset{p}{\longrightarrow} \tau(p)\ \mathit{type}}{\Sigma\, A\, \tau\ \mathit{type}}$$

$$\frac{\cdot \Vdash A\ \mathit{true} \text{ (via } p) \qquad \Delta \Vdash \tau(p)\ \mathit{true}}{\Delta \Vdash \Sigma\, A\, \tau\ \ \mathit{true}}\ \text{pair}$$

Dependent pairs

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

$$\frac{A\ \mathit{type} \qquad [] \Vdash \overset{p}{A}\ \mathit{true} \longrightarrow \tau(p)\ \mathit{type}}{\Sigma\, A\, \tau\ \mathit{type}}$$

$$\frac{\cdot \Vdash \overset{p}{A}\ \mathit{true} \qquad \Delta \Vdash \tau(p)\ \mathit{true}}{\Delta \Vdash \Sigma\, A\, \tau\ \ \mathit{true}}\ \mathrm{pair}$$

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
   - Σ A τ 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
   - Σ A τ quantifies over patterns

Why does this make sense?

Induction-Recursion

1. Inductively define the syntax of positive types

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

2. Simultaneously, recursively define patterns for A^+

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

Induction-Recursion

$$\frac{A\ \mathit{type} \qquad [\,] \Vdash A\ \mathit{true} \xrightarrow{\;p\;} \tau(p)\ \mathit{type}}{\Sigma\, A\ \tau\ \mathit{type}}$$

τ quantifies over type A, which is **smaller than** Σ A τ

1. Define the type A
2. Define the patterns for A
3. Define the types Σ A τ (quantifies over pats for A)
4. Define the patterns for Σ A τ
5. ...

Example

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

Example

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

...contrapositive...

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

Example

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

...contrapositive...

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

...one premise...

$$head :: \quad \Delta \Vdash \Sigma\ \mathrm{nat}\ (n \mapsto \mathrm{vec}[\mathrm{s}\ n])\ \mathit{true} \longrightarrow (\kappa : \mathrm{bool}\ \mathit{false}), \Delta \vdash \#$$

Example

$$head :: \quad \Delta \Vdash \Sigma\, \mathrm{nat}\, (n \mapsto \mathrm{vec}[\mathrm{s}\; n])\; true \longrightarrow (\kappa : \mathrm{bool}\; false), \Delta \vdash \#$$

$$head\; (\mathrm{pair}\; \_\; (\mathrm{cons}\; x\; \_)) \mapsto throw\; x\; to\; \kappa$$

(no case for *head* ($\mathrm{pair}\; n\; \mathrm{nil}$) !)

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!