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 $\text{inl}$ or $\text{inr}$
- 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 $A \to B$ 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 $\text{inl}$ or $\text{inr}$
- 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

Focus = make choices
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 \textbf{positive data}:
\begin{itemize}
  \item Introduced by choosing \texttt{inl} or \texttt{inr}
  \item Eliminated by \texttt{pattern-matching}
\end{itemize}

ML functions $A \rightarrow B$ are \textbf{negative computation}:
\begin{itemize}
  \item Introduced by \texttt{pattern-matching} on $A$
  \item Eliminated by choosing an $A$ to apply to
\end{itemize}

Inversion = respond to all possible choices
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 $\alpha$-equivalence, substitution
- Eliminated by pattern matching = structural induction modulo $\alpha$
Positive function space

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

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:

\[ A^+ ::= \text{nat} \mid A^+ \times B^+ \mid 1 \mid A^+ \oplus B^+ \mid 0 \mid \downarrow A^- \]

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

\[
\begin{align*}
\text{A false} & \\
\therefore \neg A \text{ true} & \\
\hline
\therefore \neg A \otimes (\neg B \oplus \neg C) \text{ true}
\end{align*}
\]

\[
\begin{align*}
\text{B false} & \\
\therefore \neg B \text{ true} & \\
\hline
\therefore \neg B \oplus \neg C \text{ true}
\end{align*}
\]
Patterns

\( A_1 \text{ false, ..., } A_n \text{ false} \models A \text{ true} \)
Δ ⊩ A true : there is a proof pattern for A, leaving holes for refutations of $A_1$, ..., $A_n$
Pattern rules

\[ A \text{ false} \vdash \neg A \text{ true} \]

<table>
<thead>
<tr>
<th>( \Delta_1 \vdash A \text{ true} )</th>
<th>( \Delta_2 \vdash B \text{ true} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta_1 \Delta_2 \vdash A \odot B \text{ true} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \Delta \vdash A \text{ true} )</th>
<th>( \Delta \vdash B \text{ true} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \vdash A \oplus B \text{ true} )</td>
<td>( \Delta \vdash A \oplus B \text{ true} )</td>
</tr>
</tbody>
</table>

\( \cdot \vdash I \text{ true} \)

(no rule for 0)

\( A \text{ false} \vdash \neg A \text{ true} \)
Proof terms

\[
\begin{align*}
\text{A false} & \vdash \neg A \text{ true} \\
\text{B false} & \vdash \neg B \text{ true} \\
\text{A false, B false} & \vdash \neg A \otimes (\neg B \oplus \neg C) \text{ true}
\end{align*}
\]

\[
(k_1, \text{inl}\ k_2)
\]

continuation variables
Proof terms

\[ K_1 \]

\[ \text{A false} \vdash \neg \text{A true} \]

\[ K_2 \]

\[ \begin{array}{c}
B \text{ false} \vdash \neg B \text{ true} \\
B \text{ false} \vdash \neg B \oplus \neg \text{C true}
\end{array} \]

\[ (\neg B \oplus \neg \text{C}) \text{ true} \]

\[ \approx \]

\[ (K_1, \text{inl } K_2) \]

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

\[ \Delta \vdash A \text{ true} \quad \Gamma \vdash \Delta \quad \Gamma \vdash A \text{ true} \]

\[ \Delta \vdash A \text{ false} \quad \Gamma \vdash \Delta \quad \Gamma \vdash A \text{ false} \]

\[ A \text{ false} \in \Delta \quad \Gamma \vdash A \text{ false} \quad \Gamma \vdash \Delta \]

\[ A \text{ false} \in \Delta \quad \Gamma \vdash A \text{ true} \quad \Gamma \vdash \# \]

iterated inductive definition
Focused proofs

\[
\Delta \vdash A \text{ true} \quad \Gamma \vdash \Delta \\
\Gamma \vdash A \text{ true} \quad \text{focus}
\]

\[
A \text{ false} \in \Delta \quad \Gamma \vdash A \text{ false} \\
\Gamma \vdash \Delta
\]

\[
\Delta \vdash A \text{ true} \quad \Gamma, \Delta \vdash \# \\
\Gamma \vdash A \text{ false} \quad \text{inversion}
\]

\[
A \text{ false} \in \Delta \quad \Gamma \vdash A \text{ true} \\
\Gamma \vdash \#
\]

iterated inductive definition
Example continuation

K deriv. of

\[ \Delta \vdash \neg A \boxdot (\neg B \oplus \neg C) \text{ true} \]
\[ \Gamma \vdash \neg A \boxdot (\neg B \oplus \neg C) \text{ false} \]
$\Delta \vdash \neg A \otimes (\neg B \oplus \neg C)$ true

$\Gamma \vdash \neg A \otimes (\neg B \oplus \neg C)$ false

$\Gamma, \Delta \vdash \#$

\begin{align*}
\text{K deriv. of} & \quad \frac{}{\Delta \vdash \neg A \otimes (\neg B \oplus \neg C)} \\
\text{K} & \left\{ 
\begin{aligned}
(K_1, \text{inl } K_2) & \Rightarrow \
(K_1, \text{inr } K_3) & \Rightarrow
\end{aligned}
\right.
\end{align*}

$E_1$

$\text{Γ, } K_1 : A \text{ false, } K_2 : B \text{ false, } \vdash \#$

$E_2$

$\text{Γ, } K_1 : A \text{ false, } K_3 : C \text{ false, } \vdash \#$
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

all the changes are here
1. Allow indexing by closed patterns
   = values of purely positive types
Patterns

\( \text{nat:} \quad \frac{}{\frac{}{\Delta \vdash \text{nat } true}} \quad \frac{}{\frac{}{\Delta \vdash \text{nat } true}} \quad z \)
Patterns

\[ \text{nat: } \begin{array}{c}
\Delta_1 \vdash \text{nat } true \\
\Delta_1 \vdash \text{nat } true \\
\Delta_1 \vdash \text{nat } true
\end{array} \]

\[ \text{vec}[\rho :: \cdot \vdash \text{nat } true]: \]

\[ \begin{array}{c}
\Delta_1 \vdash \text{bool } true \\
\Delta_2 \vdash \text{vec}[\rho] \text{ true}
\end{array} \]

\[ \Delta_1 \Delta_2 \vdash \text{vec}[s \rho] \text{ true} \]

\[ \text{nil} \]

\[ \text{cons} \]
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

A type \[ \vdash A \text{ true} \rightarrow \tau(p) \text{ type} \]

\[ \Sigma A \text{ type} \]
Dependent pairs

\[
\begin{align*}
A \text{ type} & \quad \vdash \ A \text{ true} \rightarrow \tau(p) \text{ type} \\
\Sigma \ A \ \tau \text{ type} \\
\vdash \ A \text{ true} & \quad \Delta \vdash \tau(p) \text{ true} \\
\Delta & \vdash \Sigma \ A \ \tau \text{ true}
\end{align*}
\]
Dependent pairs

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

Pattern: \((\text{pair } 2 (\text{cons true (cons false nil)}))\)

\[
\frac{\text{A type}}{\Sigma \text{A \tau type}} \quad \frac{p}{\Sigma \text{A \tau type}}
\]

\[
\frac{\text{[]} \vdash \text{A true} \rightarrow \tau(p) \text{ type}}{\text{pair}}
\]

\[
\frac{\Delta \vdash \text{A true}}{\Delta \vdash \Sigma \text{A \tau true}}
\]
Dependent pairs

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

Only pattern: pair true <>

\[
\begin{align*}
A \text{ type} & \quad \frac{\text{\([\emptyset] \vdash A \text{ true } \rightarrow \tau(p) \text{ type}\)}}{\Sigma A \text{ type}}
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash A \text{ true} & \quad \Delta \vdash \tau(p) \text{ true} \\
\Delta \vdash \Sigma A \text{ }& \Delta \vdash \text{ pair }
\end{align*}
\]
Dependent pairs

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

\[
\begin{align*}
\text{A type} & \quad \Sigma \text{A type} \\
\text{pair} & \quad \\
\text{A true} & \quad \tau(p) \text{ true} \\
\text{pair} & \quad \\
\Sigma \text{A true} & \quad \tau(p) \text{ true} \\
\end{align*}
\]
Dependent pairs

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

\[
\begin{align*}
A & \text{ type} \\
\emptyset & \models A \text{ true} \rightarrow \tau(p) \text{ type} \\
\Sigma & \text{ A type} \\
\Delta & \models \Sigma A \text{ true} \\
\end{align*}
\]
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?
1. **Inductively** define the syntax of positive types

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

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

\[
\Delta \vdash A \oplus B \overset{\text{def}}{=} \text{Either} (\Delta \vdash A) (\Delta \vdash B)
\]
Induction-Recursion

\[ \text{A type} \quad \vdash \quad A \text{ true} \rightarrow \tau(p) \text{ type} \]

\[ \Sigma A \tau \text{ type} \]

\( \tau \) 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

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

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

...contrapositive...

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

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

...contrapositive...

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

...one premise...

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

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

\[
\text{head} :: \quad \Delta \vdash \Sigma \text{nat} \ (n \mapsto \text{vec}[s \ n]) \ \text{true} \\
\quad (\kappa : \text{bool} \ \text{false}), \Delta \vdash \# \\
\text{head} ((\text{pair } _ \ (\text{cons} \ x \ _)) \mapsto \text{throw} \ x \ \text{to} \ \kappa \\
\text{(no case for head (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 \(\tau\) and \(\tau'\) 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 $\tau$ 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!