Focusing on Binding and Computation

Robert Harper
Carnegie Mellon University

(Joint work with Dan Licata and Noam Zeilberger)

June 18, 2008

The Payload

Main Results and Ideas

Integrate Logical Frameworks and Functional Programming.

- LF level provides a generalized datatype mechanism adequate for syntax, judgements, rules, proofs.
- FP level provides the means to compute over these datatypes.

In this talk we restrict attention to simple (non-indexed) types (to appear, LICS 2008).
Current work on extending to dependent types and indexed types (not to appear, ICFP 2008).

Main Results and Ideas

Polarized type systems.

- Positive types are inductively defined by intro/focusing rules, manipulated by elim/inversion rules.
- Negative types are inductively defined by elim/inversion rules, manipulated by intro/focusing rules.

Contextual modal type systems.

- $\langle\Psi\rangle A$ has as elements "open terms" with parameters specified by context Ψ.
- Treats binding and scope without reliance on effects/state.

Motivation

Representation and Computation

Goal: integrate representation and computation in a functional language.

1. Representation: types for syntax including binding and scope.
2. Computation: type of higher-order computations over these types.

Representation and Computation

Goal: integrate representation and computation in a functional language.

1. Representation: types for syntax including binding and scope.
2. Computation: type of higher-order computations over these types.

Requirements:

1. Sufficiently powerful to represent syntax, judgements, rules, proofs.
2. Sufficiently flexible to permit computation by structural induction modulo α-equivalence.
3. Purely functional, so that we may index types by syntax.

Example: Domain-Specific Logics

Access control logic (excerpts):

```
sort  : type.
princ : sort.
res   : sort.

term             : sort => type.
dan              : term princ.
bob              : term princ.
/home/dan/pub : term res.

prop : type.
owns  : term princ => term res => prop.
mayrd : term princ => term res => prop.
```

Example: Domain-Specific Logics

Access control logic (excerpts):

```
true    : prop => type.
affirms : term princ => prop => type.

impi : (imp A B) true <= (A true => B true).
impe : B true <= A true <= (imp A B) true.

aff   : K affirms A <= A true.

saysi : (K says A) true <= K affirms A.
sayse : (K affirms C) <= (says K A) <=
        (K affirms A => K affirms C).
```

Example: Domain-Specific Logics

Signature for proof-carrying access control:

```
type file[r:term res]
val  paper.tex : file[/home/dan/pub]

type iam[p:term princ]
val  iambob : iam[bob]

val  read :
  ∀r.∀p.∀pf:atom (p mayrd r) true.
  file[r] -> iam[p] -> string
```

Implementation of `read` structurally analyzes proofs at run-time!

Representation and Computation

There are two *different* function spaces in play here!

1. Representational: $A \Rightarrow B$ (aka $B \Leftarrow A$).
2. Computational: $A \rightarrow B$ (aka $B \leftarrow A$).

Representational functions:

Representation and Computation

There are two *different* function spaces in play here!

1. Representational: $A \Rightarrow B$ (aka $B \Leftarrow A$).
2. Computational: $A \to B$ (aka $B \leftarrow A$).

Representational functions:

- Adequate for syntax, rules, proofs.

Representation and Computation

There are two *different* function spaces in play here!

1. Representational: $A \Rightarrow B$ (aka $B \Leftarrow A$).
2. Computational: $A \to B$ (aka $B \leftarrow A$).

Representational functions:

- Adequate for syntax, rules, proofs.
- Closed-ended: schemas built from parameters by composing rules.

Representation and Computation

There are two *different* function spaces in play here!

1. Representational: $A \Rightarrow B$ (aka $B \Leftarrow A$).
2. Computational: $A \rightarrow B$ (aka $B \leftarrow A$).

Representational functions:

- Adequate for syntax, rules, proofs.
- Closed-ended: schemas built from parameters by composing rules.

Computational functions:

Representation and Computation

There are two *different* function spaces in play here!

1. Representational: $A \Rightarrow B$ (aka $B \Leftarrow A$).
2. Computational: $A \rightarrow B$ (aka $B \leftarrow A$).

Representational functions:

- Adequate for syntax, rules, proofs.
- Closed-ended: schemas built from parameters by composing rules.

Computational functions:

- Compute by pattern matching.

Representation and Computation

There are two *different* function spaces in play here!

1. Representational: $A \Rightarrow B$ (aka $B \Leftarrow A$).
2. Computational: $A \rightarrow B$ (aka $B \leftarrow A$).

Representational functions:

- Adequate for syntax, rules, proofs.
- Closed-ended: schemas built from parameters by composing rules.

Computational functions:

- Compute by pattern matching.
- Open-ended: any form of computation allowable.

Derivability and Admissibility

Representational functions witness derivabilities, $J_1 \vdash J_2$.

Derivability and Admissibility

Representational functions witness derivabilities, $J_1 \vdash J_2$.

- J_2 is derivable, taking J_1 as a fresh axiom.
- Evidence is *uniform*: $\lambda x{:}J_1.M : J_1 \Rightarrow J_2$.

Derivability and Admissibility

Representational functions witness derivabilities, $J_1 \vdash J_2$.

- J_2 is derivable, taking J_1 as a fresh axiom.
- Evidence is *uniform*: $\lambda x{:}J_1.M : J_1 \Rightarrow J_2$.

Computational functions witness admissibilities, $J_1 \models J_2$.

Derivability and Admissibility

Representational functions witness derivabilities, $J_1 \vdash J_2$.

- J_2 is derivable, taking J_1 as a fresh axiom.
- Evidence is *uniform*: $\lambda x{:}J_1.M : J_1 \Rightarrow J_2$.

Computational functions witness admissibilities, $J_1 \models J_2$.

- Derivability of J_1 implies derivability of J_2.
- Evidence is *non-uniform*: any function mapping derivations of J_1 to derivations of J_2.

Derivability and Admissibility

Representational functions witness derivabilities, $J_1 \vdash J_2$.

- J_2 is derivable, taking J_1 as a fresh axiom.
- Evidence is *uniform*: $\lambda x{:}J_1.M : J_1 \Rightarrow J_2$.

Computational functions witness admissibilities, $J_1 \models J_2$.

- Derivability of J_1 implies derivability of J_2.
- Evidence is *non-uniform*: any function mapping derivations of J_1 to derivations of J_2.

Side conditions correspond to rules that mix both forms:

$$\frac{\neg(l \in \mathit{dom}(M))}{(M, l) \Uparrow} \quad \textit{i.e.} \quad \frac{l \in \mathit{dom}(M) \models \bot}{(M, l) \Uparrow}$$

Representation and Computation

Representational functions are

Representation and Computation

Representational functions are

- Introduced by composing rules from parameters.

Representation and Computation

Representational functions are

- Introduced by composing rules from parameters.
- Eliminated by pattern matching / structural analysis.

Representation and Computation

Representational functions are

- Introduced by composing rules from parameters.
- Eliminated by pattern matching / structural analysis.

Computational functions are

Representation and Computation

Representational functions are

- Introduced by composing rules from parameters.
- Eliminated by pattern matching / structural analysis.

Computational functions are

- Introduced by pattern matching / structural analysis.

Representation and Computation

Representational functions are

- Introduced by composing rules from parameters.
- Eliminated by pattern matching / structural analysis.

Computational functions are

- Introduced by pattern matching / structural analysis.
- Eliminated by application to an argument.

Representation and Computation

Representational functions are

- Introduced by composing rules from parameters.
- Eliminated by pattern matching / structural analysis.

Computational functions are

- Introduced by pattern matching / structural analysis.
- Eliminated by application to an argument.

Focusing provides a general framework for such dualities!

Focusing

Intro vs. Elim

Sums $A \oplus B$:

Intro vs. Elim

Sums $A \oplus B$:

- Introduced by choosing inl or inr

Intro vs. Elim

Sums $A \oplus B$:

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

Intro vs. Elim

Sums $A \oplus B$:

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

Computational functions $A \to B$:

Intro vs. Elim

Sums $A \oplus B$:

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

Computational functions $A \longrightarrow B$:

- Introduced by pattern-matching on A

Intro vs. Elim

Sums $A \oplus B$:

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

Computational functions $A \rightarrow B$:

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

Positive vs. Negative Polarity

Sums $A \oplus B$ are positive:

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

Computational functions $A \longrightarrow B$ are negative:

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

Positive vs. Negative Polarity

Sums $A \oplus B$ are positive:

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

Computational functions $A \rightarrow B$ are negative:

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

Operationally: positive = eager, negative = lazy

Focus vs. Inversion

Sums $A \oplus B$ are positive:

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

Computational functions $A \longrightarrow B$ are negative:

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

Focus vs. Inversion

Sums $A \oplus B$ are positive:

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

Computational functions $A \longrightarrow B$ are negative:

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

Focus = make choices

Focus vs. Inversion

Sums $A \oplus B$ are positive:

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

Computational functions $A \longrightarrow B$ are negative:

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

Focus vs. Inversion

Sums $A \oplus B$ are positive:

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

Computational functions $A \rightarrow B$ are negative:

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

Inversion = respond to all possible choices

Polarity and Focusing

	Positive type	Negative type
Intro	Focus	Inversion
Elim	Inversion	Focus

Higher-order Focusing

A concise way to define a language:

- Specify a type by its focused behavior
- Derive the inversion phase generically

Polarized Type Theory

A concise way to define a language:

- Specify a type by its focused behavior
  - Choices = patterns
- Derive the inversion phase generically
  - Response = pattern matching

Patterns for Positive Types

$$
\begin{array}{lcl}
A^{+} & ::= & A^{+} \oplus B^{+} \mid A^{+} \otimes B^{+} \mid {\downarrow} A^{-} \\
A^{-} & ::= & A^{+} \rightarrow B^{-} \mid \ldots
\end{array}
$$

$$
\frac{\Delta \Vdash p :: A^{+}}{\Delta \Vdash \mathsf{inl}\, p :: A^{+} \oplus B^{+}} \qquad \frac{\Delta \Vdash p :: B^{+}}{\Delta \Vdash \mathsf{inr}\, p :: A^{+} \oplus B^{+}}
$$

Patterns for Positive Types

$$A^+ \quad ::= \quad A^+ \oplus B^+ \mid A^+ \otimes B^+ \mid {\downarrow} A^-$$
$$A^- \quad ::= \quad A^+ \to B^- \mid \ldots$$

$$\frac{\Delta \Vdash p :: A^+}{\Delta \Vdash \mathsf{inl}\, p :: A^+ \oplus B^+} \qquad \frac{\Delta \Vdash p :: B^+}{\Delta \Vdash \mathsf{inr}\, p :: A^+ \oplus B^+}$$

$$\frac{\Delta_1 \Vdash p_1 :: A^+ \quad \Delta_2 \Vdash p_2 :: B^+}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) :: A^+ \otimes B^+}$$

Patterns for Positive Types

$$
\begin{array}{lcl}
A^{+} & ::= & A^{+} \oplus B^{+} \mid A^{+} \otimes B^{+} \mid {\downarrow} A^{-} \\
A^{-} & ::= & A^{+} \to B^{-} \mid \ldots
\end{array}
$$

$$
\frac{\Delta \Vdash p :: A^{+}}{\Delta \Vdash \mathsf{inl}\, p :: A^{+} \oplus B^{+}} \qquad \frac{\Delta \Vdash p :: B^{+}}{\Delta \Vdash \mathsf{inr}\, p :: A^{+} \oplus B^{+}}
$$

$$
\frac{\Delta_1 \Vdash p_1 :: A^{+} \quad \Delta_2 \Vdash p_2 :: B^{+}}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) :: A^{+} \otimes B^{+}}
$$

$$
\frac{}{x : A^{-} \Vdash x :: {\downarrow} A^{-}}
$$

Positive Focus

- positive value is pattern p with substitution σ
- σ substitutes negative values v^- / x for $x : A^- \in \Delta$

$$\frac{\Delta \Vdash p :: C^+ \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash p\,[\sigma] :: C^+}$$

Positive Inversion

- positive continuation is a case-analysis
- specified by meta-level function $\phi = \{p \mapsto e, \ldots\}$ from patterns to expressions

$$\frac{\forall(\Delta \Vdash p :: C^{+}).\ \Gamma, \Delta \vdash \phi(p) : D^{+}}{\Gamma \vdash \mathsf{val}^{+}(\phi) : C^{+} > D^{+}}$$

Example

Define

```
and* (true  , true ) = true[·]
and* (true  , false) = false[·]
and* (false , true ) = false[·]
and* (false , false) = false[·]
```

Then $\cdot \vdash \mathsf{val}^{+}(\texttt{and*}) : (\mathsf{bool} \otimes \mathsf{bool}) > \mathsf{bool}$

Negative Focus and Inversion is Dual

- *Continuation* specified by destructor pattern (focus)
- *Value* defined by pattern-matching ϕ (inversion)

Negative Focus and Inversion is Dual

- *Continuation* specified by destructor pattern (focus)
- *Value* defined by pattern-matching ϕ (inversion)

Simplification for this talk:

- Equate $\Gamma \vdash v^{-} : A^{+} \longrightarrow B^{+}$ with $\Gamma \vdash k^{+} : A^{+} > B^{+}$
  e.g. $\cdot \vdash \texttt{add*} : (\text{bool} \otimes \text{bool}) \longrightarrow \text{bool}$
- Eliminated by choosing a value to apply it to

Generalized Datatypes

Datatypes

- Class of datatypes P
- Datatype constructors u specified by signature $\Psi = \ldots, u : R, \ldots$
- Rules R have the form $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$ (construct P from $A_1^+, \ldots, A_n^+$)

Natural numbers:

$$\Psi_{\mathsf{nat}} = \mathsf{zero} : \mathsf{nat}, \mathsf{succ} : \mathsf{nat} \Leftarrow \mathsf{nat}$$

Datatype Patterns

Add signature to pattern judgement: $\Delta \,;\, \Psi \Vdash p :: A^{+}$

$$\frac{u : P \Leftarrow A_1^{+} \cdots \Leftarrow A_n^{+} \in \Psi \qquad \Delta_1 \,;\, \Psi \Vdash p_1 :: A_1^{+} \quad \cdots \quad \Delta_n \,;\, \Psi \Vdash p_n :: A_n^{+}}{\Delta_1, \ldots, \Delta_n \,;\, \Psi \Vdash u\ p_1 \ldots p_n :: P}$$

Datatype Continuations

Meta-functions ϕ now require infinitely many cases:

$$\Psi_{\mathsf{nat}} = \mathsf{zero} : \mathsf{nat}, \mathsf{succ} : \mathsf{nat} \Leftarrow \mathsf{nat}$$

To prove

$$\Psi_{\mathsf{nat}}; \cdot \vdash \mathsf{val}^{+}(\mathtt{double*}) : \mathsf{nat} > \mathsf{nat}$$

STS

$$\forall(\Delta\,;\,\Psi_{\mathsf{nat}} \Vdash p :: \mathsf{nat}).\ \Psi_{\mathsf{nat}}; \Delta \vdash \mathtt{double*}(p) : \mathsf{nat}$$

Datatype Continuations

$$\forall(\Delta\,;\,\Psi_{\mathsf{nat}} \Vdash p :: \mathsf{nat}).\ \Psi_{\mathsf{nat}}; \Delta \vdash \mathtt{double*}(p) : \mathsf{nat}$$

```
double* 0 = 0
double* 1 = 2
double* 2 = 4
...
```

Open-endedness:
compatible with any concrete presentation of ϕ

Contextual Hypotheses

Make hypotheses contextual:

$$\Delta \quad ::= \quad \cdot \mid \Delta, x : \langle \Psi \rangle\ A^{-}$$

$$\frac{}{x : \langle \Psi \rangle\ A^{-}\ ;\ \Psi \Vdash x :: {\downarrow} A^{-}}$$

Rule from before:

$$\frac{}{x : A^{-} \Vdash x :: {\downarrow} A^{-}}$$

Contextual Continuations

Make continuations transform *contextualized types*:

$$\frac{\forall(\Delta\,;\,\Psi \Vdash p :: A^{+}).\ \Gamma, \Delta \vdash \phi(p) : \langle \Psi_1 \rangle\ A^{+}_{1}}{\Gamma \vdash \mathsf{val}^{+}(\phi) : \langle \Psi \rangle\ A^{+} > \langle \Psi_1 \rangle\ A^{+}_{1}}$$

Rule from before:

$$\frac{\forall(\Delta \Vdash p :: C^{+}).\ \Gamma, \Delta \vdash \phi(p) : D^{+}}{\Gamma \vdash \mathsf{val}^{+}(\phi) : C^{+} > D^{+}}$$

Contextual Continuations

Make continuations transform *contextualized types*:

$$\frac{\forall(\Delta\,;\,\Psi \Vdash p :: A^{+}).\ \Gamma, \Delta \vdash \phi(p) : \langle\Psi_1\rangle\ A_1^{+}}{\Gamma \vdash \mathsf{val}^{+}(\phi) : \langle\Psi\rangle\ A^{+} > \langle\Psi_1\rangle\ A_1^{+}}$$

Rule from before:

$$\frac{\forall(\Delta \Vdash p :: C^{+}).\ \Gamma, \Delta \vdash \phi(p) : D^{+}}{\Gamma \vdash \mathsf{val}^{+}(\phi) : C^{+} > D^{+}}$$

Allows for types that manipulate $\Psi \ldots$

Representational Functions

Represent binding with a positive function space:

$$\frac{\Delta\,;\,\Psi, u : R \Vdash p :: A^{+}}{\Delta\,;\,\Psi \Vdash \lambda\, u.\, p :: R \Rightarrow A^{+}}$$

- Representational arrow $R \Rightarrow A^{+}$ binds a scoped datatype constructor
- Pattern-matching gives induction over HOAS

Example

$$e \quad ::= \quad \mathsf{num}[k] \mid e_1 \odot_f e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

Represent with a datatype ari:

$$\begin{array}{l} \mathsf{zero} : \mathsf{nat},\ \mathsf{succ} : \mathsf{nat} \Leftarrow \mathsf{nat}, \\ \mathsf{num} : \mathsf{ari} \Leftarrow \mathsf{nat} \\ \mathsf{binop} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{nat} \otimes \mathsf{nat} \rightarrow \mathsf{nat}) \Leftarrow \mathsf{ari} \\ \mathsf{let} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{ari} \Rightarrow \mathsf{ari}) \end{array}$$

Example

Evaluator:

$$\cdot \vdash \mathsf{fix}(ev.\mathsf{ev}^*) : \langle \Psi_{\mathsf{ari}} \rangle\, (\mathsf{ari} \longrightarrow \mathsf{nat})$$

STS:

$$\forall (\Delta \Vdash p :: \langle \Psi_{\mathsf{ari}} \rangle\, \mathsf{ari}).$$
$$(ev : \langle \Psi_{\mathsf{ari}} \rangle\, \mathsf{ari} \longrightarrow \mathsf{nat}, \Delta) \vdash (\mathsf{ev}^*\ p) : \langle \Psi_{\mathsf{ari}} \rangle\, \mathsf{nat}$$

Example

$$\forall(\Delta \Vdash p :: \langle\Psi_{\mathsf{ari}}\rangle\, \mathsf{ari}).$$
$$(ev : \langle\Psi_{\mathsf{ari}}\rangle\, \mathsf{ari} \longrightarrow \mathsf{nat}, \Delta) \vdash (\textit{ev}^{*}\ p) : \langle\Psi_{\mathsf{ari}}\rangle\, \mathsf{nat}$$

$$\begin{array}{lll} \textit{ev}^{*} & (\mathsf{num}\ p) & \longmapsto p \\ \textit{ev}^{*} & (\mathsf{binop}\ p_1\ \ f\ \ p_2) & \longmapsto f\ (ev\ p_1)\ (ev\ p_2) \\ \textit{ev}^{*} & (\mathsf{let}\ p_0\ \ (\lambda\, u.\, p)) & \longmapsto ev\ (\textit{apply}\ (\lambda\, u.\, p, p_0)) \end{array}$$

Example

$$\forall(\Delta \Vdash p :: \langle\Psi_{\mathsf{ari}}\rangle\, \mathsf{ari}).$$
$$(ev : \langle\Psi_{\mathsf{ari}}\rangle\, \mathsf{ari} \to \mathsf{nat}, \Delta) \vdash (\mathit{ev}^*\ p) : \langle\Psi_{\mathsf{ari}}\rangle\, \mathsf{nat}$$

$$\begin{array}{lll} \mathit{ev}^* & (\mathsf{num}\ p) & \mapsto p \\ \mathit{ev}^* & (\mathsf{binop}\ p_1\ \ f\ \ p_2) & \mapsto f\ (ev\ p_1)\ (ev\ p_2) \\ \mathit{ev}^* & (\mathsf{let}\ p_0\ \ (\lambda\, u.\, p)) & \mapsto ev\ (\mathit{apply}\ (\lambda\, u.\, p, p_0)) \end{array}$$

What is apply?

Substitution

$$\textit{apply} : \langle \Psi \rangle \, ((P \Rightarrow A) \otimes P) \rightarrow A$$

- Just a program: not forced by the type theory
- Should it always be defined?

Substitution

$$\mathit{apply} : \langle\Psi\rangle\,((P \Rightarrow A) \otimes P) \rightarrow A$$

- Just a program: not forced by the type theory
- Should it always be defined?

Substitution requires weakening...

Weakening

$$\textit{weaken} : \langle \Psi \rangle\ A \to (P \Rightarrow A)$$

Can you weaken

- ...an ari to ari $\Rightarrow$ ari?

Weakening

$$\mathit{weaken} : \langle\Psi\rangle\ A \to (P \Rightarrow A)$$

Can you weaken

- . . . an ari to $\mathsf{ari} \Rightarrow \mathsf{ari}$?

  Hint: $\mathsf{let} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{ari} \Rightarrow \mathsf{ari})$

Weakening

$$\textit{weaken} : \langle \Psi \rangle\, A \rightarrow (P \Rightarrow A)$$

Can you weaken

- ... an ari to ari ⇒ ari?

  Hint: let : ari ⇐ ari ⇐ (ari ⇒ ari)

- ... a nat to ari ⇒ nat?

Weakening

$$\textit{weaken} : \langle \Psi \rangle\, A \to (P \Rightarrow A)$$

Can you weaken

- …an ari to ari ⇒ ari?

  Hint: $\mathsf{let} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{ari} \Rightarrow \mathsf{ari})$

- …a nat to ari ⇒ nat?

- …an ari to nat ⇒ ari?

Weakening

$$\textit{weaken} : \langle \Psi \rangle\, A \rightarrow (P \Rightarrow A)$$

Can you weaken

- $\ldots$an ari to $\mathsf{ari} \Rightarrow \mathsf{ari}$?

  Hint: $\mathsf{let} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{ari} \Rightarrow \mathsf{ari})$

- $\ldots$a nat to $\mathsf{ari} \Rightarrow \mathsf{nat}$?

- $\ldots$an ari to $\mathsf{nat} \Rightarrow \mathsf{ari}$?

  Hint: $\mathsf{binop} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{nat} \otimes \mathsf{nat} \rightarrow \mathsf{nat}) \Leftarrow \mathsf{ari}$

Structural Properties

- Structural properties hold when types are not circumscribed (includes all LF rule systems)
- Exploiting open-endedness, implement *apply*, *weaken*, . . . once as datatype-generic programs at the meta-level

Conclusion

Conclusion

- Logical framework for rules that mix $\Rightarrow$ and $\longrightarrow$
  - Representation is positive
  - Computation is negative
- Get structural properties "for free" under conditions

  Otherwise you have to implement them, if they're even true
- Lots more to the story. . . (see LICS'08 paper and follow-ups).