FLOLAC'12

Logic

Lecture 3: Curry–Howard correspondence

29 August 2012

柯向上

Department of Computer Science
University of Oxford

Hsiang-Shang.Ko@cs.ox.ac.uk

Annotated derivation

$$\dfrac{\dfrac{\dfrac{}{\mathtt{A},\ \mathtt{A} \to \mathtt{B} \vdash \mathtt{A} \to \mathtt{B}} \qquad \dfrac{}{\mathtt{A},\ \mathtt{A} \to \mathtt{B} \vdash \mathtt{A}}}{\mathtt{A},\ \mathtt{A} \to \mathtt{B} \vdash \mathtt{B}}\,(\to\mathsf{E})}{\dfrac{\mathtt{A} \vdash (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}{\vdash \mathtt{A} \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}\,(\to\mathsf{I})}\,(\to\mathsf{I})$$

Annotated derivation

$$\dfrac{\dfrac{}{\mathtt{x}:\mathtt{A},\ \mathtt{y}:\mathtt{A}\to\mathtt{B}\vdash\mathtt{A}\to\mathtt{B}} \qquad \dfrac{}{\mathtt{x}:\mathtt{A},\ \mathtt{y}:\mathtt{A}\to\mathtt{B}\vdash\mathtt{A}}}{\dfrac{\dfrac{\mathtt{x}:\mathtt{A},\ \mathtt{y}:\mathtt{A}\to\mathtt{B}\vdash\mathtt{B}}{\mathtt{x}:\mathtt{A}\vdash(\mathtt{A}\to\mathtt{B})\to\mathtt{B}}\,(\to\mathsf{I})}{\vdash\mathtt{A}\to(\mathtt{A}\to\mathtt{B})\to\mathtt{B}}\,(\to\mathsf{I})}\,(\to\mathsf{E})$$

- Label elements in contexts with (distinct) names.

Annotated derivation

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \mathtt{A},\ \mathtt{y} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{y} : \mathtt{A} \to \mathtt{B}} \qquad \dfrac{}{\mathtt{x} : \mathtt{A},\ \mathtt{y} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{x} : \mathtt{A}}}{\mathtt{x} : \mathtt{A},\ \mathtt{y} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{B}}\,(\to\mathsf{E})}{\dfrac{\mathtt{x} : \mathtt{A} \vdash (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}{\vdash \mathtt{A} \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}\,(\to\mathsf{I})}\,(\to\mathsf{I})$$

- Label elements in contexts with (distinct) names.
- Represent (assum) by the name of the assumption used.

Annotated derivation

$$\dfrac{\dfrac{\dfrac{\overline{\mathtt{x : A,\ y : A \to B \vdash \textcolor{blue}{y} : A \to B}} \qquad \overline{\mathtt{x : A,\ y : A \to B \vdash \textcolor{blue}{x} : A}}}{\mathtt{x : A,\ y : A \to B \vdash \textcolor{red}{y\ x} : B}}\ (\to\mathsf{E})}{\mathtt{x : A \vdash (A \to B) \to B}}\ (\to\mathsf{I})}{\mathtt{\vdash A \to (A \to B) \to B}}\ (\to\mathsf{I})$$

- Label elements in contexts with (distinct) names.
- Represent (assum) by the name of the assumption used.
- Represent ($\to$E) by juxtaposing the representations of its two sub-derivations.

Annotated derivation

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \mathtt{A},\ \mathtt{y} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{y} : \mathtt{A} \to \mathtt{B}} \qquad \dfrac{}{\mathtt{x} : \mathtt{A},\ \mathtt{y} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{x} : \mathtt{A}}}{\dfrac{\mathtt{x} : \mathtt{A},\ \mathtt{y} : \mathtt{A} \to \mathtt{B} \vdash \mathtt{y}\ \mathtt{x} : \mathtt{B}}{\mathtt{x} : \mathtt{A} \vdash \lambda \mathtt{y}.\ \mathtt{y}\ \mathtt{x} : (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}\ (\to\mathsf{I})}\ (\to\mathsf{E})}{\vdash \lambda \mathtt{x}.\ \lambda \mathtt{y}.\ \mathtt{y}\ \mathtt{x} : \mathtt{A} \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{B}}\ (\to\mathsf{I})$$

- Label elements in contexts with (distinct) names.
- Represent (assum) by the name of the assumption used.
- Represent ($\to$E) by juxtaposing its the representations of two sub-derivations.
- Represent ($\to$I) by prefixing $\lambda\, v.$ to the representation of its sub-derivation, where v is the name of the new assumption.

Annotated derivation

$$\dfrac{\dfrac{}{\mathtt{x : A,\ y : A \to B \vdash y : A \to B}}\,(\mathsf{var}) \quad \dfrac{}{\mathtt{x : A,\ y : A \to B \vdash x : A}}\,(\mathsf{var})}{\dfrac{\mathtt{x : A,\ y : A \to B \vdash y\ x : B}}{\dfrac{\mathtt{x : A \vdash \lambda\, y.\ y\ x : (A \to B) \to B}}{\mathtt{\vdash \lambda\, x.\ \lambda\, y.\ y\ x : A \to (A \to B) \to B}}\,(\mathsf{abs})}\,(\mathsf{abs})}\,(\mathsf{app})$$

- Label elements in contexts with (distinct) names.
- Represent (assum) by the name of the assumption used.
- Represent ($\to$E) by juxtaposing the representations of its two sub-derivations.
- Represent ($\to$I) by prefixing $\lambda\, v.$ to the representation of its sub-derivation, where v is the name of the new assumption.

This is a typing derivation for the λ-term $\mathtt{\lambda\, x.\ \lambda\, y.\ y\ x}$!

Simply typed λ-calculus (á la Curry)

Let the set of *types* be the *implicational fragment* of $\textsc{Prop}$, i.e., the subset of the propositional language generated by variables and implication only.

A λ-term t is said to *have type τ under context Γ* if, using the following rules, there is a closed typing derivation whose conclusion is $\Gamma \vdash t : \tau$. In this case we simply write $\Gamma \vdash t : \tau$.

$$\frac{}{\Gamma \vdash v : \tau}\ (\text{var}) \quad \text{if} \quad (v : \tau) \in \Gamma$$

$$\frac{\Gamma, v : \sigma \vdash t : \tau}{\Gamma \vdash \lambda v.\ t : \sigma \to \tau}\ (\text{abs}) \qquad \frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\, s : \tau}\ (\text{app})$$

Curry–Howard correspondence

Deduction systems and programming calculi can be put in correspondence — a corresponding pair of a deduction system and a programming calculus can be regarded as logical and computational interpretations of essentially the same set of syntactic objects.

Slogan: *propositions are types; proofs are programs.*

Natural deduction for full propositional logic corresponds to simply typed λ-calculus with constants: defining the set of types to be $\textsc{Prop}$, the derivations in natural deduction (the proofs) correspond exactly to the well-typed λ-terms (the programs).

BHK interpretation revised

A ~~proposition~~ type is ~~an expression~~ a specification of what counts as ~~its proof~~ a conforming program.

- There is no program of type $\perp$.
- A program of type $\varphi \wedge \psi$ is one that computes a program of type φ and a program of type ψ.
- A program of type $\varphi \vee \psi$ is one that computes either a program of type φ or a program of type ψ.
- A program of type $\varphi \to \psi$ is a function which computes a program of type ψ given a program of type φ as its input.

Cartesian products

Conjunctions correspond to cartesian products: the introduction rule gives type to the pairing operator,

$$\frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \langle s, t \rangle : \sigma \wedge \tau} \; (\wedge\mathsf{I})$$

and the two elimination rules give types to the projections.

$$\frac{\Gamma \vdash t \, : \, \sigma \wedge \tau}{\Gamma \vdash \mathtt{outl} \; t \, : \, \sigma} \; (\wedge\mathsf{EL}) \qquad \frac{\Gamma \vdash t \, : \, \sigma \wedge \tau}{\Gamma \vdash \mathtt{outr} \; t \, : \, \tau} \; (\wedge\mathsf{ER})$$

Note that we are adding the constants $\langle \_, \_ \rangle$, `outl`, and `outr` into the language of λ-calculus.

Disjoint sums

Disjunctions correspond to disjoint sums (unions): the introduction rules give types to the injections,

$$\frac{\Gamma \vdash s : \sigma}{\Gamma \vdash \mathtt{inl}\ s : \sigma \vee \tau}\ (\vee\mathsf{IL}) \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathtt{inr}\ t : \sigma \vee \tau}\ (\vee\mathsf{IR})$$

and the elimination rule gives type to the conditional operator.

$$\frac{\Gamma \vdash c : \sigma \vee \tau \qquad \Gamma, u : \sigma \vdash s : \vartheta \qquad \Gamma, v : \tau \vdash t : \vartheta}{\Gamma \vdash \mathtt{case}\ c \left[\begin{array}{l} u \rightsquigarrow s \\ v \rightsquigarrow t \end{array} \right. : \vartheta}\ (\vee\mathsf{E})$$

Again we add the constants $\mathtt{inl}$, $\mathtt{inr}$, and $\mathtt{case}\_\left[\begin{array}{l} \_ \rightsquigarrow \_ \\ \_ \rightsquigarrow \_ \end{array} \right.$ to the language of λ-calculus.

Example: distributivity

The type

$$\mathtt{A} \wedge (\mathtt{B} \vee \mathtt{C}) \to (\mathtt{A} \wedge \mathtt{B}) \vee (\mathtt{A} \wedge \mathtt{C})$$

is inhabited by the λ-term

$$\lambda\, \mathtt{x}.\ \mathtt{case}\ (\mathtt{outr}\ \mathtt{x}) \left[\begin{array}{l} \mathtt{y} \leadsto \mathtt{inl}\ \langle \mathtt{outl}\ \mathtt{x}, \mathtt{y} \rangle \\ \mathtt{z} \leadsto \mathtt{inr}\ \langle \mathtt{outl}\ \mathtt{x}, \mathtt{z} \rangle \end{array} \right.\ .$$

Empty set

$\perp$ is interpreted as the empty set. The elimination rule gives type to a variant of Dijkstra's `abort` operator.

$$\frac{\Gamma \vdash t : \perp}{\Gamma \vdash \mathtt{abort}\ t : \varphi}\ (\perp\mathsf{E})$$

Example. The type $\top$, i.e., $\perp \to \perp$, is inhabited by $\lambda\, \mathtt{x}.\ \mathtt{abort}\ \mathtt{x}$.

δ-reduction

In pure λ-calculus we have β-reduction that rewrites β-redexes.

$$(\lambda\, v.\ s)\ t \ \rightsquigarrow_{\beta}\ s\,[t/v]$$

Note that this is how an introduction form (λ-abstraction) interacts with an elimination form (application).

For λ-calculus with constants, we should also specify how to reduce the *δ-redexes*, which involve the introduction and elimination forms of the additional constants.

$$\mathtt{outl}\ \langle s, t\rangle \ \rightsquigarrow_{\delta}\ s \qquad \mathtt{outr}\ \langle s, t\rangle \ \rightsquigarrow_{\delta}\ t$$

$$\mathtt{case}\ (\mathtt{inl}\ p) \left[\begin{array}{l} u \rightsquigarrow s \\ v \rightsquigarrow t \end{array}\right. \ \rightsquigarrow_{\delta}\ s\,[p/u]$$

$$\mathtt{case}\ (\mathtt{inr}\ q) \left[\begin{array}{l} u \rightsquigarrow s \\ v \rightsquigarrow t \end{array}\right. \ \rightsquigarrow_{\delta}\ t\,[q/v]$$

Proof normalisation

β-/δ-redexes in λ-terms correspond to *detours* in derivations, and evaluation of λ-terms corresponds to *proof normalisation*.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\mathtt{B} \to \mathtt{C} \to \mathtt{B}, \mathtt{A} \vdash \mathtt{B} \to \mathtt{C} \to \mathtt{B}}
    }{\mathtt{B} \to \mathtt{C} \to \mathtt{B} \vdash {\color{blue}\mathtt{A} \to \mathtt{B} \to \mathtt{C} \to \mathtt{B}}}\;{\color{blue}(\to\mathsf{I})}
  }{\vdash (\mathtt{B} \to \mathtt{C} \to \mathtt{B}) \to \mathtt{A} \to \mathtt{B} \to \mathtt{C} \to \mathtt{B}}\;{\color{red}(\to\mathsf{I})}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{}{\mathtt{B}, \mathtt{C} \vdash \mathtt{B}}
    }{\mathtt{B} \vdash \mathtt{C} \to \mathtt{B}}\;(\to\mathsf{I})
  }{\vdash \mathtt{B} \to \mathtt{C} \to \mathtt{B}}\;(\to\mathsf{I})
}{\vdash \mathtt{A} \to \mathtt{B} \to \mathtt{C} \to \mathtt{B}}\;{\color{red}(\to\mathsf{E})}
$$

normalises to

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\mathtt{A}, \mathtt{B}, \mathtt{C} \vdash \mathtt{B}}
    }{\mathtt{A}, \mathtt{B} \vdash \mathtt{C} \to \mathtt{B}}\;(\to\mathsf{I})
  }{\mathtt{A} \vdash \mathtt{B} \to \mathtt{C} \to \mathtt{B}}\;(\to\mathsf{I})
}{\cancel{\mathtt{B} \to \mathtt{C} \to \mathtt{B}} \vdash {\color{blue}\mathtt{A} \to \mathtt{B} \to \mathtt{C} \to \mathtt{B}}}\;{\color{blue}(\to\mathsf{I})}
$$

The corresponding reduction is

$$(\lambda\,\mathtt{x}.\ \lambda\,\mathtt{y}.\ \mathtt{x})\ (\lambda\,\mathtt{z}.\ \lambda\,\mathtt{w}.\ \mathtt{z}) \ \rightsquigarrow_\beta\ \lambda\,\mathtt{y}.\ \lambda\,\mathtt{z}.\ \lambda\,\mathtt{w}.\ \mathtt{z}.$$

Detours

We need a substitution function on derivations which has type

$$\Gamma, \varphi \vdash_{\mathrm{NJ}} \psi \;\rightarrow\; \Gamma \vdash_{\mathrm{NJ}} \varphi \;\rightarrow\; \Gamma \vdash_{\mathrm{NJ}} \psi,$$

corresponding to substitution on λ-terms.

Wherever the assumption φ is used in the first derivation we plug in a suitably weakened version of the second derivation.

Detours

Corresponding to the β-/δ-redexes, the possible forms of detours are:

$$\dfrac{\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\,(\to\mathsf{I}) \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\,(\to\mathsf{E})$$

$$\dfrac{\dfrac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\,(\wedge\mathsf{I})}{\Gamma \vdash \varphi}\,(\wedge\mathsf{EL}) \qquad \dfrac{\dfrac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\,(\wedge\mathsf{I})}{\Gamma \vdash \psi}\,(\wedge\mathsf{ER})$$

$$\dfrac{\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\,(\vee\mathsf{IL}) \qquad \Gamma, \varphi \vdash \vartheta \qquad \Gamma, \psi \vdash \vartheta}{\Gamma \vdash \vartheta}\,(\vee\mathsf{E})$$

$$\dfrac{\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}\,(\vee\mathsf{IR}) \qquad \Gamma, \varphi \vdash \vartheta \qquad \Gamma, \psi \vdash \vartheta}{\Gamma \vdash \vartheta}\,(\vee\mathsf{E})$$

Subject reduction and strong normalisation

For simply typed λ-calculus we have the following results.

Theorem (subject reduction)**.** If $\Gamma \vdash t : \tau$ and $t \rightsquigarrow_{\beta\delta} t'$, then $\Gamma \vdash t' : \tau$.

Theorem (strong normalisation)**.** Every reduction sequence of a well-typed λ-term terminates.

Corollary. Every well-typed λ-term has a normal form.

They are readily translated into theorems about derivations.

Theorem. Elimination of a detour produces a derivation with the same conclusion.

Theorem. Every derivation can be normalised (to a derivation that does not contain detours).

Canonicity

Definition. A λ-term is in *canonical form* if its head position is an introduction form, i.e., one of the following:

- λ-abstraction,
- pairing $\langle \_, \_ \rangle$, and
- injections `inl` and `inr`.

Theorem (canonicity)**.** If $\vdash t : \tau$ and t is in normal form, then t is in canonical form.

PROOF Induction on the typing derivation of t. The elimination forms give rise to redexes, in contradiction to the assumption that t is in normal form.

Underivability

Corollary. NJ is consistent, i.e., $\nvdash_{\mathrm{NJ}} \perp$.

$\boxed{\text{PROOF}}$ If $\vdash_{\mathrm{NJ}} \perp$, then there is a λ-term of type $\perp$ in canonical form. But none of the canonical forms can have type $\perp$.

Remark. This notion of consistency, which is about the deduction system NJ itself, is different from the one about theories that we introduced in the first lecture.

Underivability

Corollary (disjunction property). If $\vdash_{\mathrm{NJ}} \varphi \vee \psi$, then either $\vdash_{\mathrm{NJ}} \varphi$ or $\vdash_{\mathrm{NJ}} \psi$.

PROOF A λ-term of type $\varphi \vee \psi$ under the empty context can be reduced to either `inl` p where $\vdash p : \varphi$ or `inr` q where $\vdash q : \psi$.

Remark. The disjunction property does not hold for NK.

Corollary. $\mathtt{A} \vee \neg\mathtt{A}$ is underivable in NJ.

PROOF If $\vdash_{\mathrm{NJ}} \mathtt{A} \vee \neg\mathtt{A}$, then either $\vdash_{\mathrm{NJ}} \mathtt{A}$ or $\vdash_{\mathrm{NJ}} \neg\mathtt{A}$ by the disjunction property, and thus either $\models \mathtt{A}$ or $\models \neg\mathtt{A}$ by soundness. But neither $\mathtt{A}$ nor $\neg\mathtt{A}$ is a tautology.

Unifying programming and reasoning

The Curry–Howard correspondence suggests that programs and proofs be identified. Both of them are *mental constructions*, which are all that intuitionistic mathematics cares about.

Per Martin-Löf: "If programming is understood

- not as the writing of instructions for this or that computing machine
- but as the design of methods of computation that it is the computer's duty to execute
  - (a difference that Dijkstra has referred to as the difference between comput**er** science and comput**ing** science),

then it no longer seems possible to distinguish the discipline of programming from constructive mathematics."

Martin-Löf Type Theory

Martin-Löf Type Theory is an influential framework in which programs and proofs are treated uniformly. It is simultaneously

- a computationally meaningful higher-order logic system and
- a very expressively typed functional programming language.

There are numerous variations, extensions, and applications of MLTT. The Coq proof assistant is one of its descendants.

Predicates as type functions/families

Let $\texttt{Set}$ be the type of all "small" propositions/sets/types.

A predicate on a set A is a function of type $A \to \texttt{Set}$, which can be regarded as *a family of types indexed by A*.

Example. Define the predicate $\mathit{Even} : \mathbb{N} \to \texttt{Set}$ by

$$\begin{array}{lcl} \mathit{Even}\ 0 & = & \top \\ \mathit{Even}\ 1 & = & \bot \\ \mathit{Even}\ (2+n) & = & \mathit{Even}\ n. \end{array}$$

Then $\mathit{Even}\ 4$ computes to $\top$ and is thus inhabited, whereas $\mathit{Even}\ 3$ computes to $\bot$ and has no inhabitant.

Allowing such type functions means that types can depend on values and that non-trivial computation can happen at type level. Such type disciplines are called *dependent types*.

Operations on type families

Let $A : \texttt{Set}$ and $B : A \to \texttt{Set}$. Over the type family B we can form

- the *dependent product type* $\Pi\ A\ B\ :\ \texttt{Set}$ and
- the *dependent sum type* $\Sigma\ A\ B\ :\ \texttt{Set}$.

Dependent product types

Let $A : \texttt{Set}$ and $B : A \to \texttt{Set}$.

An element of the set $\Pi\ A\ B$ is a function that, given $a : A$, returns an element of $B\ a$.

Dependent product types

- provide universal quantification,
- generalise conjunction, and
- subsume implication.

They are also known as *dependent function types*.

Dependent sum types

Let $A : \texttt{Set}$ and $B : A \to \texttt{Set}$.

An element of the set $\Sigma\ A\ B$ is a pair whose first component is an element $a : A$ and whose second component is an element of $B\ a$.

Dependent sum types

- provide existential quantification,
- generalise disjunction, and
- subsume conjunction.

They are also known as *dependent pair types*.

Algebraic datatypes

Inductively defined sets are algebraic datatypes.

```
-- PV : Set

data Prop⁻ : Set where
  bot : Prop⁻
  var : PV → Prop⁻
  imp : Prop⁻ → Prop⁻ → Prop⁻

-- Membership : Prop⁻ → List Prop⁻ → Set

data NJ⁻ : List Prop⁻ → Prop⁻ → Set where
  assum     : Membership φ Γ → NJ⁻ Γ φ
  botElim   : NJ⁻ Γ bot → NJ⁻ Γ φ
  impIntro  : NJ⁻ (φ :: Γ) ψ → NJ⁻ Γ (imp φ ψ)
  impElim   : NJ⁻ Γ (imp φ ψ) → NJ⁻ Γ φ → NJ⁻ Γ ψ
```

Induction principle

The induction principle for an algebraic datatype is the type of a variant of the fold operator on the datatype.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\texttt{indProp}^{-} : (\texttt{P} : \texttt{Prop}^{-} \to \texttt{Set}) \to} \\
\multicolumn{3}{l}{\qquad\qquad \texttt{P bot} \to} \\
\multicolumn{3}{l}{\qquad\qquad ((\texttt{v} : \texttt{PV}) \to \texttt{P (var v)}) \to} \\
\multicolumn{3}{l}{\qquad\qquad ((\texttt{phi} : \texttt{Prop}^{-}) \to (\texttt{psi} : \texttt{Prop}^{-}) \to} \\
\multicolumn{3}{l}{\qquad\qquad\quad \texttt{P phi} \to \texttt{P psi} \to \texttt{P (imp phi psi)}) \to} \\
\multicolumn{3}{l}{\qquad\quad (\texttt{phi} : \texttt{Prop}^{-}) \to \texttt{P phi}} \\
\texttt{indProp}^{-}\ \texttt{P pbot pvar pimp bot} & = & \texttt{pbot} \\
\texttt{indProp}^{-}\ \texttt{P pbot pvar pimp (var v)} & = & \texttt{pvar v} \\
\texttt{indProp}^{-}\ \texttt{P pbot pvar pimp (imp phi psi)} & = & \\
\multicolumn{3}{l}{\quad \texttt{pimp phi psi (}\texttt{indProp}^{-}\ \texttt{P pbot pvar pimp phi)}} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\quad \texttt{(}\texttt{indProp}^{-}\ \texttt{P pbot pvar pimp psi)}}
\end{array}
$$

Notation. We abbreviate $\Pi\, A\, (\lambda\, x.\ B\, x)$ to $(x : A) \to B\, x$.

Programming with more precise types

Rather than giving a sorting function on lists of natural numbers the simple type

$$\texttt{List}\ \mathbb{N} \to \texttt{List}\ \mathbb{N},$$

we can assign to it a more informative type

$$(\texttt{xs} : \texttt{List}\ \mathbb{N}) \to (\texttt{ys} : \texttt{List}\ \mathbb{N}) \times \texttt{Perm xs ys} \times \texttt{Ordered ys}.$$

A program of this type

- not only describes a computational process
- but also includes a correctness proof that the process performs sorting, whose validity can be checked by a typechecker.

Notation. We abbreviate $\Sigma\ A\ (\lambda\, x.\ B\ x)$ to $(x : A) \times B\ x$.

Summary: the triangle

Languages
(Propositional, First-order)
Inductive syntax

$[\![\_]\!]$

Semantics
(BHK, Truth-value)
Semantic consequence

Soundness
Semantic completeness

Deduction systems
(NJ, NK)
Derivability
Consistency
Syntactic completeness

Curry–Howard correspondence: programming and reasoning go hand in hand.