A review of the Curry-Howard-De Bruijn formulas-as-types interpretation

Herman Geuvers

Foundations group, Intelligent Systems, ICIS
Radboud University Nijmegen
The Netherlands

Mathematical Logic in the Netherlands
May 26,, 2009,
Radboud University Nijmegen

What is mathematics about?

Foundations of mathematics:

What is mathematics about?

Foundations of mathematics:

- Formalism: mathematics is a formal game with formal rules. Meaning? Only finitary maths has a canonical meaning. A theory is good if it is consistent. Hilbert

What is mathematics about?

Foundations of mathematics:

- Formalism: mathematics is a formal game with formal rules. Meaning? Only finitary maths has a canonical meaning. A theory is good if it is consistent. Hilbert
- Realism: But mathematics also has a relation with the real world! Also infinitary mathematics!

What is mathematics about?

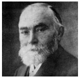
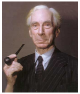
Foundations of mathematics:

- Formalism: mathematics is a formal game with formal rules. Meaning? Only finitary maths has a canonical meaning. A theory is good if it is consistent. Hilbert
- Realism: But mathematics also has a relation with the real world! Also infinitary mathematics!
- Platonism: Abstract and infinitary mathematical objects also "exist".

What is mathematics about?

Foundations of mathematics:

- Formalism: mathematics is a formal game with formal rules. Meaning? Only finitary maths has a canonical meaning. A theory is good if it is consistent. Hilbert
- Realism: But mathematics also has a relation with the real world! Also infinitary mathematics!
- Platonism: Abstract and infinitary mathematical objects also "exist".
- Logicism: Logics is the universal basis; build mathematics out of logics. Frege, Russell

What is mathematics about?

Foundations of mathematics:

- Formalism: mathematics is a formal game with formal rules. Meaning? Only finitary maths has a canonical meaning. A theory is good if it is consistent. Hilbert
- Realism: But mathematics also has a relation with the real world! Also infinitary mathematics!
- Platonism: Abstract and infinitary mathematical objects also “exist”.
- Logicism: Logics is the universal basis; build mathematics out of logics. Frege, Russell
- Intuitionism / Constructivism: Only the objects that one can construct (in time) exist. Brouwer

Brouwer's Intuitionism

Mathematics is primary and comes before logic. Logic is descriptive.
Basic intuition: construction of an object in time: $\mathbb{N}$
A proof (mathematical argument) is also a construction (in time).

Brouwer's Intuitionism

Mathematics is primary and comes before logic. Logic is descriptive.
Basic intuition: construction of an object in time: $\mathbb{N}$
A proof (mathematical argument) is also a construction (in time).

What can we construct? Which mathematical arguments are valid?

Theorem: $\exists p, q,$ irrational(p^q is rational)
Proof: $\sqrt{2}^{\sqrt{2}}$ is rational OR irrational.
- First case: done; $p = q = \sqrt{2}$
- Second case: $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$ is rational and so we are done: $p = \sqrt{2}^{\sqrt{2}}$, $q = \sqrt{2}$

Brouwer: A statement is true if we have a proof for it.

The intuitionistic notion of truth

Brouwer: A statement is true if we have a proof for it.

So the real question is:

What is a proof?

Brouwer has never made this formally precise, because Brouwer wasn't interested in logic. Heyting and Kolmogorov have.

Brouwer-Heyting-Kolmogorov interpretation (BHK)

Die formalen Regeln der intuitionistischen Logik.

Von Dr. A. Heyting
in Enschede (Niederlande).

(Vorgelegt von Hrn. Bieberbach am 19. Dezember 1929 [s. Jahrg. 1929 S. 686].)

Einleitung.

Die intuitionistische Mathematik ist eine Denktätigkeit, und jede Sprache, auch die formalistische, ist für sie nur Hilfsmittel zur Mitteilung. Es ist prinzipiell unmöglich, ein System von Formeln aufzustellen, das mit der intuitionistischen Mathematik gleichwertig wäre, denn die Möglichkeiten des Denkens lassen sich nicht auf eine endliche Zahl von im voraus aufstellbaren Regeln zurückführen. Der Versuch, die wichtigsten Teile der Mathematik in Formelsprache wiederzugeben[1], wird deshalb ausschließlich gerechtfertigt durch die größere Bündigkeit und Bestimmtheit der letzteren gegenüber der gewöhnlichen Sprache, Eigenschaften, welche sie geeignet machen, das Eindringen in die intuitionistischen Begriffe und ihre Verwendung bei Untersuchungen zu erleichtern.

Zum Aufbau der Mathematik ist die Aufstellung allgemeingültiger logischer Gesetze nicht notwendig; diese Gesetze werden in jedem einzelnen Fall gleichsam von neuem entdeckt als gültig für das eben betrachtete mathematische System. Die sprachliche Mitteilung aber, nach den Bedürfnissen des täglichen Lebens gebildet, schreitet in der Form der logischen Gesetze, welche sie als gegeben voraussetzt, fort. Eine Sprache, welche dem Gang der intuitionistischen Mathematik von Schritt zu Schritt nachgebildet wäre, würde so in allen Teilen von der gewohnten Form abweichen, daß sie die obengenannten günstigen Eigenschaften wieder gänzlich verlieren müßte. Diese Überlegungen haben mich dazu geführt, die Formalisierung der intuitionistischen Mathematik doch wieder mit einem Aussagenkalkül anzufangen.

Die Formeln des formalistischen Systems entstehen aus einer endlichen Zahl von Axiomen durch Anwendung einer endlichen Zahl von Operationsregeln. Sie enthalten außer den »konstanten« Zeichen auch Variabele. Das Verhältnis zwischen diesem System und der Mathematik ist nun dieses, daß bei einer bestimmten Interpretation der Konstanten und unter bestimmten Beschränkungen hinsichtlich der Ersetzung der Variabeln jede Formel einen richtigen mathematischen Satz darstellt. (Z. B. müssen die Variabeln im Aussagenkalkül nur durch sinnerfüllte mathematische Aussagen ersetzt werden.) Ist das System so beschaffen, daß es die letztgenannte Forderung erfüllt, so

[1] Diese Abhandlung bildet eine Umarbeitung des ersten Teiles einer von dem »Wiskundig Genootschap« in Amsterdam im Anfang 1928 gekrönten Preisschrift.

Brouwer-Heyting-Kolmogorov interpretation (BHK)

A proof of

$A \wedge B$	is a pair consisting of a proof of A and a proof of B
$A \vee B$	is a proof of A or a proof of B
$A \to B$	is a method for producing a proof of B, given a proof of A
$\bot$	doesn't exist
$\forall x \in D(A(x))$	is a method for producing a proof of $A(d)$, given an element $d \in D$,
$\exists x \in D(A(x))$	is a pair consisting of an element $d \in D$ and a proof of $A(d)$.

Brouwer-Heyting-Kolmogorov interpretation (BHK)

A proof of

$A \wedge B$	is a pair consisting of a proof of A and a proof of B
$A \vee B$	is a proof of A or a proof of B
$A \rightarrow B$	is a method for producing a proof of B, given a proof of A
$\bot$	doesn't exist
$\forall x \in D(A(x))$	is a method for producing a proof of $A(d)$, given an element $d \in D$,
$\exists x \in D(A(x))$	is a pair consisting of an element $d \in D$ and a proof of $A(d)$.

So: there is no proof of $A \vee \neg A$

Brouwer-Heyting-Kolmogorov interpretation (BHK)

A proof of

$A \wedge B$	is a pair consisting of a proof of A and a proof of B
$A \vee B$	is a proof of A or a proof of B
$A \to B$	is a method for producing a proof of B, given a proof of A
$\perp$	doesn't exist
$\forall x \in D(A(x))$	is a method for producing a proof of $A(d)$, given an element $d \in D$,
$\exists x \in D(A(x))$	is a pair consisting of an element $d \in D$ and a proof of $A(d)$.

So: there is no proof of $A \vee \neg A$

So: a proof of $\forall x \in D \exists y \in E(A(x,y))$ contains a method for constructing a $e \in E$ for every $d \in D$ such that $A(d,e)$ holds.

Kleene Realisability, Curry-Howard Formules as Types

We can make the BHK interpretation formal in various ways:
Kleene realisability

$$m \mathrel{\mathrm{r}} A$$

"m realises the formula A" ($m \in \mathbb{N}$, seen as the code of a Turing machine)

Kleene Realisability, Curry-Howard Formules as Types

We can make the BHK interpretation formal in various ways:
Kleene realisability

$$m \mathrel{\mathrm{r}} A$$

"m realises the formula A" ($m \in \mathbb{N}$, seen as the code of a Turing machine)
Curry-Howard formulas as types:

$$M : A$$

"M has type A" (M an algorithm / functional programma / data object)

- a formula is seen as a type (or a specification)
- a proof is seen as an algorithm (program)

Formulas as Types, Proofs as Terms

A proof of (term of type)

$A \wedge B$	is a term $\langle p, q \rangle$ with $p : A$ and $q : B$
$A \vee B$	is $\mathsf{inl}\, p$ with $p : A$ or $\mathsf{inr}\, q$ with $q : B$
$A \rightarrow B$	is a term $f : A \rightarrow B$
$\bot$	doesn't exist
$\forall x \in D(A(x))$	is a term $f : \Pi_{x \in D} A(x)$
$\exists x \in D(A(x))$	is a term $\langle d, p \rangle$ with $d : D$ and $p : A(d)$

Formulas as Types, Proofs as Terms

A proof of (term of type)

$A \wedge B$	is a term $\langle p, q \rangle$ with $p : A$ and $q : B$
$A \vee B$	is $\mathsf{inl}\, p$ with $p : A$ or $\mathsf{inr}\, q$ with $q : B$
$A \to B$	is a term $f : A \to B$
$\bot$	doesn't exist
$\forall x \in D(A(x))$	is a term $f : \Pi_{x \in D} A(x)$
$\exists x \in D(A(x))$	is a term $\langle d, p \rangle$ with $d : D$ and $p : A(d)$

Formulas and sets are both (data)types

Proofs and objects are both terms (data, programs)

Formulas as Types, Proofs as Terms

A proof of (term of type)

$A \wedge B$	is a term $\langle p, q \rangle$ with $p : A$ and $q : B$
$A \vee B$	is $\mathsf{inl}\, p$ with $p : A$ or $\mathsf{inr}\, q$ with $q : B$
$A \rightarrow B$	is a term $f : A \rightarrow B$
$\bot$	doesn't exist
$\forall x \in D(A(x))$	is a term $f : \Pi_{x \in D} A(x)$
$\exists x \in D(A(x))$	is a term $\langle d, p \rangle$ with $d : D$ and $p : A(d)$

Formulas and sets are both (data)types

Proofs and objects are both terms (data, programs)

Two "readings" of $M : A$:

- M is a proof of the formula A
- M is data of type A

The Formulas-as-Types notion of Construction (Howard 1980)

Paper dates back to 1969.
Original ideas go back to Curry (Combinatory Logic):
$\mathbf{K} := \lambda x\, \lambda y.x : A \to B \to A$
$\mathbf{S} := \lambda x\, \lambda y\, \lambda z.x\, z(y\, z) : (A \to B \to C) \to (A \to B) \to A \to C$
$\mathbf{I} := \lambda x.x : A \to A$

The Formulas-as-Types notion of Construction (Howard 1980)

Paper dates back to 1969.
Original ideas go back to Curry (Combinatory Logic):
$\mathbf{K} := \lambda x\, \lambda y.x : A \to B \to A$
$\mathbf{S} := \lambda x\, \lambda y\, \lambda z.x\, z(y\, z) : (A \to B \to C) \to (A \to B) \to A \to C$
$\mathbf{I} := \lambda x.x : A \to A$

Theorem: For (first order) proposition and predicate logic we have a formulas-as-types isomorphism between proofs and terms.

$$\varphi_1, \varphi_2, \ldots, \varphi_n \vdash_L^{\Pi} \sigma \iff x_1 : \varphi_1, x_2 : \varphi_2, \ldots, x_n : \varphi_n \vdash [\Pi] : \sigma$$

The Formulas-as-Types notion of Construction (Howard 1980)

Contribution of Tait (1965):
Cut-elimination in logic $=$ β-reduction in typed λ-calculus.

The Formulas-as-Types notion of Construction (Howard 1980)

Contribution of Tait (1965):
Cut-elimination in logic = β-reduction in typed λ-calculus.

$$\dfrac{\dfrac{\begin{matrix}[\sigma]^1 \\ \mathcal{D}_1 \\ \tau\end{matrix}}{\sigma \to \tau}\,1 \qquad \dfrac{\mathcal{D}_2}{\sigma}}{\tau} \quad \longrightarrow \quad \begin{matrix}\mathcal{D}_2 \\ \sigma \\ \mathcal{D}_1 \\ \tau\end{matrix}$$

The Formulas-as-Types notion of Construction (Howard 1980)

Contribution of Tait (1965):
Cut-elimination in logic = β-reduction in typed λ-calculus.

$$
\dfrac{\dfrac{\begin{matrix}[\sigma]^1 \\ \mathcal{D}_1 \\ \tau\end{matrix}}{\sigma \to \tau}\,1 \qquad \dfrac{\mathcal{D}_2}{\sigma}}{\tau}
\quad \longrightarrow \quad
\begin{matrix}\mathcal{D}_2 \\ \sigma \\ \mathcal{D}_1 \\ \tau\end{matrix}
$$

$$
\dfrac{\dfrac{\begin{matrix}[x : \sigma]^1 \\ \mathcal{D}_1 \\ M : \tau\end{matrix}}{\lambda x{:}\sigma.M : \sigma \to \tau}\,1 \qquad \dfrac{\mathcal{D}_2}{P : \sigma}}{(\lambda x{:}\sigma.M)P : \tau}
\quad \longrightarrow_\beta \quad
\begin{matrix}\mathcal{D}_2 \\ P : \sigma \\ \mathcal{D}_1 \\ M[P/x] : \tau\end{matrix}
$$

Formulas-as-Types: proof theory and type theory

proof theory		type theory
termination of cut-elimination	$\Leftrightarrow$	SN of β-reduction
every proof can be made cut-free	$\Leftrightarrow$	WN of β-reduction
disjunction property	$\Leftarrow$	CR and WNof β-reduction
existence property	$\Leftarrow$	CR and WN of β-reduction

SN = strong normalization,
WN = weak normalization,
CR = confluence

Formulas-as-Types: Arithmetic

Extend with recursor / induction:

$$\frac{F : P(0) \quad G : \forall n(P(x) \to P(S(x)))}{\mathsf{R}\, F\, G : \forall n(P(x))}$$

Formulas-as-Types: Arithmetic

Extend with recursor / induction:

$$\frac{F : P(0) \quad G : \forall n(P(x) \to P(S(x)))}{\mathsf{R}\, F\, G : \forall n(P(x))}$$

$$\begin{aligned} \mathsf{R}\, F\, G\, 0 &\to_\iota F \\ \mathsf{R}\, F\, G\, (S\, x) &\to_\iota G\, x\, (\mathsf{R}\, F\, G\, x) \end{aligned}$$

Formulas-as-Types: Inductive Types

Martin-Löf (Scott): take well-founded induction as basic type forming principle.

$\Rightarrow$ Induction principle

$\Rightarrow$ Recursion principle (well-founded)

Formulas-as-Types: Inductive Types

Martin-Löf (Scott): take well-founded induction as basic type forming principle.
⇒ Induction principle
⇒ Recursion principle (well-founded)

```
Inductive List (A : Set) : Set

nil : List

cons : A -> List -> List
```

Formulas-as-Types: Inductive Types

Martin-Löf (Scott): take well-founded induction as basic type forming principle.
⇒ Induction principle
⇒ Recursion principle (well-founded)

```
Inductive List (A : Set) : Set

nil : List

cons : A -> List -> List
```

$$\frac{F : P(\mathtt{nil}) \quad G : \forall a{:}A\,\forall l : \mathtt{List}_A\,(P(l) \to P(\mathtt{cons}\,a\,l))}{\mathrm{R}\,F\,G : \forall l : \mathtt{List}_A\,P(l)}$$

Formulas-as-Types: Inductive Types

Martin-Löf (Scott): take well-founded induction as basic type forming principle.
⇒ Induction principle
⇒ Recursion principle (well-founded)

```
Inductive List (A : Set) : Set
nil : List
cons : A -> List -> List
```

$$\frac{F : P(\mathtt{nil}) \quad G : \forall a{:}A\,\forall l : \mathtt{List}_A\,(P(l) \to P(\mathtt{cons}\,a\,l))}{\mathrm{R}\,F\,G : \forall l : \mathtt{List}_A\,P(l)}$$

If $P(x)$ is a proposition: "proof by induction"
If $P(x)$ is a set-type: "function def. by well-founded recursion"

Formulas-as-Types: Impredicativity

Girard has extended the formulas-as-types interpretation to higher order logic.
Higher order logic: $\forall P : A{\rightarrow}\mathrm{Prop}.\, \forall x : A.\, P\,x \rightarrow P\,x$
Polymorphic types: $\forall A : \mathrm{Set}. A \rightarrow A$

Formulas-as-Types: Impredicativity

Girard has extended the formulas-as-types interpretation to higher order logic.
Higher order logic: $\forall P : A \to \mathrm{Prop}. \forall x : A. P\,x \to P\,x$
Polymorphic types: $\forall A : \mathrm{Set}. A \to A$
Combining all these ideas: the type theory of the proof assistant Coq:

- inductive types
- dependent types
- impredicativity (higher order logic)

The SN proof of the type theory of Coq requires strongly inaccessible cardinals.

Formulas-as-Types: De Bruijn

the *desirability* of mechanical verification. In a short paper by E.W. Dijkstra on a number of processes that might sometimes block one another, the correctness of the algorithm was explained in a paragraph that ended with the remarkable sentence: "And this, the author believes, completes the proof". Indeed, the argument was a bit intuitive. I took it as a challenge and tried to build a proof that would be acceptable for mathematicians. What I achieved was long and very ugly. It might have been improved by developing efficient lemmas for avoiding the many repetitions in my argument, but I left it as it stood. Instead of improving the proof I got the idea that one should be able to instruct a machine to verify such long and tedious proofs. But of course I have to admit that it will be often more elegant and more efficient to try to streamline such an ugly proof before giving it to a machine.

The two roles of a proof in mathematics

1. A proof explains: why?
   Goal: understanding
2. A proof convinces: is it true?
   Goal: verification

For (2) one can use computer support.

Formulas as Types, Proofs as Objects

De Bruijn (re)invented the formulas-as-types principle (+/- 1968), emphasizing the proofs-as-objects aspect.

> An important thing I got from Heyting is the interpretation of a proof of an implication $A \to B$ as a kind of mapping of proofs of A to proofs of B. Later this became one of the motives to treat proof classes as types.

Automath

Isomorphism T between (names of) formulas and the types of their proofs:

$$\Gamma \vdash_{\text{logic}} \varphi \text{ iff } \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

M codes (as a λ-term) the logical derivation of φ.

Automath

Isomorphism T between (names of) formulas and the types of their proofs:

$$\Gamma \vdash_{\text{logic}} \varphi \text{ iff } \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

M codes (as a λ-term) the logical derivation of φ.
$\overline{\Gamma}$ consists of

- declarations $x : A$ of the free variables
- assumptions, of the form $y : T(\psi)$

Automath

Isomorphism T between (names of) formulas and the types of their proofs:

$$\Gamma \vdash_{\text{logic}} \varphi \text{ iff } \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

M codes (as a λ-term) the logical derivation of φ.
$\overline{\Gamma}$ consists of

- declarations $x : A$ of the free variables
- assumptions, of the form $y : T(\psi)$
- proven lemmas are definitions, stored as $y := p : T(\psi)$ (y is a name for the proof p of ψ).

Automath

Isomorphism T between (names of) formulas and the types of their proofs:

$$\Gamma \vdash_{\text{logic}} \varphi \text{ iff } \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

M codes (as a λ-term) the logical derivation of φ.
$\overline{\Gamma}$ consists of

- declarations $x : A$ of the free variables
- assumptions, of the form $y : T(\psi)$
- proven lemmas are definitions, stored as $y := p : T(\psi)$ (y is a name for the proof p of ψ).

Consequence:

proof checking = *type checking*

Automath as a Logical Framework

Automath is a language for dealing with the basic mathematical linguistic constructions, like substitution, variable binding, creation and unfolding of definitions etc.

Automath as a Logical Framework

Automath is a language for dealing with the basic mathematical linguistic constructions, like substitution, variable binding, creation and unfolding of definitions etc.

A user is free to add the logical rules that he/she wishes
$\Rightarrow$ Automath is a logical framework, where the user can do his/her own logic (or any other formal system).

Logical Framework encoding versus direct encoding

	proof	formula
direct encoding	$\lambda x{:}A.x$	$A{\to}A$
LF encoding	imp_intr $A\,A\,\lambda x{:}T\,A.x$	$T(A \Rightarrow A)$

Logical Framework encoding versus direct encoding

	proof	formula
direct encoding	$\lambda x{:}A.x$	$A{\to}A$
LF encoding	$\mathsf{imp\_intr}\, A\, A\, \lambda x{:}T\, A.x$	$T(A \Rightarrow A)$

Needed:

$$
\begin{array}{rcl}
\mathsf{prop} & : & \mathbf{type} \\
\Rightarrow & : & \mathsf{prop}{\to}\mathsf{prop}{\to}\mathsf{prop} \\
\mathsf{T} & : & \mathsf{prop}{\to}\mathbf{type} \\
\mathsf{imp\_intr} & : & \Pi A, B : \mathsf{prop}.\, (\mathsf{T}\, A \to \mathsf{T}\, B) \to \mathsf{T}(A \Rightarrow B) \\
\mathsf{imp\_el} & : & \Pi A, B : \mathsf{prop}.\, \mathsf{T}(A \Rightarrow B) \to \mathsf{T}\, A \to \mathsf{T}\, B.
\end{array}
$$

Automath as a Logical Framework

The user is responsible for the logical rules.

De Bruijn's version of the formulas-as-types principle:

Automath as a Logical Framework

The user is responsible for the logical rules.
De Bruijn's version of the formulas-as-types principle:

$$\Gamma \vdash_L \varphi \text{ iff } \Gamma_L, \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

where L is a logic, Γ_L is the context in which the constructions of the logic L have been declared.

Automath as a Logical Framework

The user is responsible for the logical rules.
De Bruijn's version of the formulas-as-types principle:

$$\Gamma \vdash_L \varphi \text{ iff } \Gamma_L, \overline{\Gamma} \vdash_{\text{type theory}} M : T(\varphi)$$

where L is a logic, Γ_L is the context in which the constructions of the logic L have been declared.
Choice and trade-off: Which logical constructions do you put in the type theory and which constructions do you declare axiomatically in the context?

Language theoretic studies

Metamathematics is about metatheory for logical systems (sequent calculus, natural deduction, . . .)

Language theoretic studies

Metamathematics is about metatheory for logical systems (sequent calculus, natural deduction, ...) but also about the

> *Metalanguage that you actually formally describe your formal systems in.*

Language theoretic studies

Metamathematics is about metatheory for logical systems (sequent calculus, natural deduction, ...) but also about the

> *Metalanguage that you actually formally describe your formal systems in.*

- How do you really do renaming of variables, capture avoiding substitution, instantiation of a quantifier, ...
- De Bruijn index representation: $\lambda\, 1\, (\lambda\, 1\, 2)$ denotes $\lambda x.x\, (\lambda y.y\, x)$.

Language theoretic studies

Metamathematics is about metatheory for logical systems (sequent calculus, natural deduction, ...) but also about the

> *Metalanguage that you actually formally describe your formal systems in.*

- How do you really do renaming of variables, capture avoiding substitution, instantiation of a quantifier, ...
- De Bruijn index representation: $\lambda\, 1\, (\lambda\, 1\, 2)$ denotes $\lambda x.x\, (\lambda y.y\, x)$.
- The "higher order" part of f.o.l. is in the logical framework (meta-language): $\forall_D : (D \to \text{prop}) \to \text{prop}$
  (This was already how Church did it in 1940.)

Extracting Programs from constructive proofs

A proof p of

$$\forall x : A\, \exists y : B\; R(x, y)$$

contains an algorithm

$$f : A \to B$$

and a proof q of $\forall x : A.R(x, f(x))$.
The specification $\forall x : A.\exists y : B.R(x, y)$, once realised (proven) produces a program that satisfies the spec.

Programming with constructive proofs

Example: sorting a list of natural numbers

$$\text{sort} : \text{List}_{\mathbb{N}} \to \text{List}_{\mathbb{N}}$$

Programming with constructive proofs

Example: sorting a list of natural numbers

$$\text{sort} : \text{List}_{\mathbb{N}} \rightarrow \text{List}_{\mathbb{N}}$$

More refined spec. (output is sorted):

$$\text{sort} : \text{List}_{\mathbb{N}} \rightarrow \exists y{:}\text{List}_{\mathbb{N}}(\text{Sorted}(y))$$

Programming with constructive proofs

Example: sorting a list of natural numbers

$$\text{sort} : \text{List}_{\mathbb{N}} \to \text{List}_{\mathbb{N}}$$

More refined spec. (output is sorted):

$$\text{sort} : \text{List}_{\mathbb{N}} \to \exists y{:}\text{List}_{\mathbb{N}}(\text{Sorted}(y))$$

Even more refined spec. (output is a permutation of the input):

$$\text{sort} : \forall x{:}\text{List}_{\mathbb{N}}\, \exists y{:}\text{List}_{\mathbb{N}}(\text{Sorted}(y) \wedge \text{Perm}(x, y))$$

The proof sort contains a sorting algorithm.

Programming with constructive proofs

Extracting the computational content from a proof.

$$\text{sort} : \forall x{:}\mathsf{List}_{\mathbb{N}}\, \exists y{:}\mathsf{List}_{\mathbb{N}} (\mathsf{Sorted}(y) \wedge \mathsf{Perm}(x, y))$$

Distinguishing data and proofs:

$$\text{sort} : \overbrace{\Pi x{:}\mathsf{List}_{\mathbb{N}}\, \Sigma y{:}\mathsf{List}_{\mathbb{N}}}^{\text{computation}} \underbrace{(\mathsf{Sorted}(y) \wedge \mathsf{Perm}(x, y))}_{\text{specification}}$$

Programming with constructive proofs

Extracting the computational content from a proof.

$$\text{sort} : \forall x{:}\text{List}_{\mathbb{N}}\, \exists y{:}\text{List}_{\mathbb{N}} (\text{Sorted}(y) \wedge \text{Perm}(x, y))$$

With data-proof distinction and program extraction:

$$\text{sort} : \Pi x{:}\text{List}_{\mathbb{N}}\, \Sigma y{:}\text{List}_{\mathbb{N}} \text{Sorted}(y) \wedge \text{Perm}(x, y))$$

$$\widehat{\text{sort}} : \text{List}_{\mathbb{N}} \to \text{List}_{\mathbb{N}}$$

$$\text{correct} : \forall x{:}\text{List}_{\mathbb{N}} (\text{Sorted}(\widehat{\text{sort}}(x)) \wedge \text{Perm}(x, \widehat{\text{sort}}(x)))$$

Formulas-as-Types, Proofs-as-Terms in Theorem Proving

$$\text{Proof checking} = \text{Type checking}$$

There is a "type check" algorithm TC:

$$\begin{aligned} \mathrm{TC}(p) &\mapsto A \text{ if } p : A \\ \mathrm{TC}(p) &\mapsto \text{fail if } p \text{ not typable} \end{aligned}$$

Proof search (Theorem Proving) =
interactive search (construction) of a term $p : A$.

Some Conclusions

Is type theory necessarily constructive?
"Constructive notion of proof $\neq$ notion of constructive proof"
(De Bruijn)

Some Conclusions

Is type theory necessarily constructive?
"Constructive notion of proof $\neq$ notion of constructive proof"
(De Bruijn)

Notion of constructive proof: Brouwer; content of axioms and rules

Constructive notion of proof: Hilbert; how to manipulate axioms and rules

Further refinements of Formulas-as-Types

Extend to classical logic

- $\forall x : \mathbb{N}\, \exists y : \mathbb{N}\, R(x, y)$ (with $R(x, y)$ atomic) is provable classically iff provable constructively
  - transform classical proof to constructive one
  - extract computational content from classical proof directly

Further refinements of Formulas-as-Types

Extend to classical logic

- $\forall x : \mathbb{N}\, \exists y : \mathbb{N}\, R(x, y)$ (with $R(x, y)$ atomic) is provable classically iff provable constructively
  - transform classical proof to constructive one
  - extract computational content from classical proof directly
- computational content of the double negation rule?
  cut-elimination is not confluent so: call-by-value vs. call-by-name
  CPS: "jumping out of a loop":

$$\mathit{mult}(l) := \mathit{if}\ \mathit{empty}(l)\ \mathit{then}\ 1\ \mathit{else}\ l[0] * \mathit{mult}\ (\mathit{tail}(l))$$

Further refinements of Formulas-as-Types

Extend to (classical) sequent calculus

- Replace sequents $\Gamma \vdash \Delta$ by $\Gamma \vdash A|\Delta$ and $\Gamma|A \vdash \Delta$.
- Proof terms can distinguish between forward and backward proofs. (Record the "proof process".)

Further refinements of Formulas-as-Types and Program Extraction

Extract programs from proofs in analysis.

- Exact real arithmetic
  - Not: determine output precision on the basis of input precision (interval arithmetic)
  - But: Let the requited output precision determiine the required input precision.