

Design Issues for Implemented Type Theories

Robert Constable
Cornell University

A Logic of Computational Reality

I like this title better, but it will take the whole lecture to explain it.

Is this approach to the title constructively sound, or is it **impredicative**?

I used the title in the article for the lecture:

The Triumph of Types: Creating a Logical Reality for Computation on the OPLSS web site.

Footnote

The article is full of footnotes, so the lecture should have one as well. This is it.

To the Nuprl proof assistant, the title is fine as are proof goals that might not **make sense until we can prove them**.

| - Goal by proof

A completed proof shows that **Goal** is sensible. This lecture will show that the title is sensible. The cool example contrasts Nuprl and Coq. **We think like this**.

A Logic of Computational Reality

Intellectual Achievement of CS

The general public is impressed by the **technical achievements** of computer science, e.g. the Internet, the Web, PC's, smart phones, computer games, social networks, robotics, tools of computational science especially applied to health care, and so forth. Seems easy, hacking.

Computer scientists, logicians, mathematicians, physicists, and so forth also care about the **intellectual achievements** of the discipline. What are some them relevant to this summer school?

Intellectual Achievements – a few related to the summer school

- Mathematically precise notion of **computability**: functional, sequential, concurrent, distributed, Turing complete, subrecursive, ...
- Concept of a **programming language**, their design, implementation, semantics, logics,...
- Concept of **computational complexity**, theory of algorithms and data structures
- **Formalization** of mathematical knowledge
- **Automation of formal reasoning** and related intellectual processes

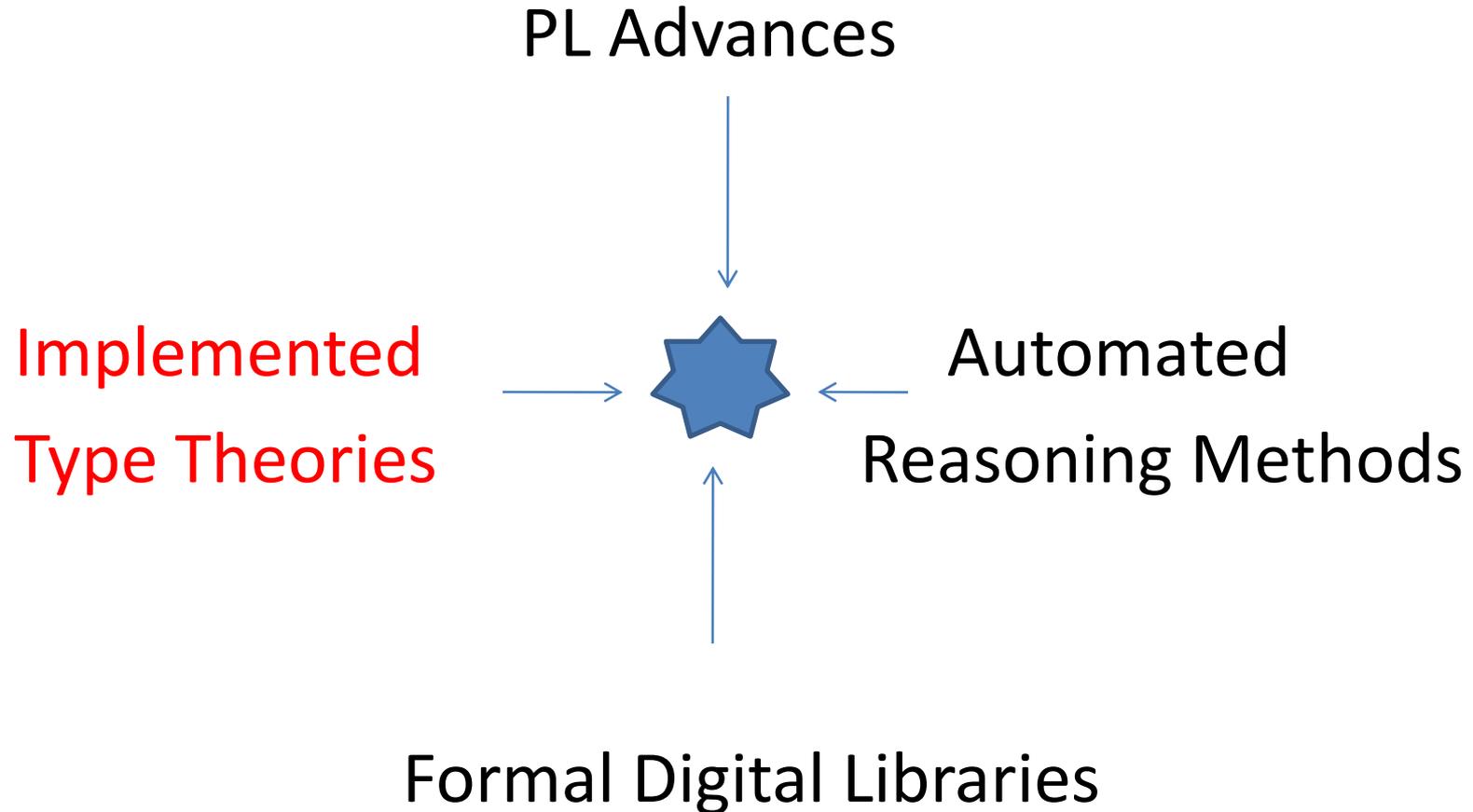
Automation Examples

- Formal Proof of Four Color Theorem – Gonthier
- Kepler Conjecture Formal Proof Attempt – Halles and HOL-Light team
- Prime Number Theorem – Harrison
- Solution of Girard paradox -- Howe
- Constructive Higman's Lemma – Murthy
- Kruskal's Theorem – Seisenberger
- POPLMark Challenge – Coq, Twelf, HOL
- Paris driverless Metro line 14 – Abrial with B-tool
- Mizar's Journal of Formalized Mathematics

What is next?

Advances in programming languages, implemented foundational theories (type theories), automated reasoning methods with machine learning, assembly of large formal digital libraries of computational mathematics and theoretical computer science, etc. are converging toward a **singularity in automating aspects of the programming process and problem solving process.** **Need to see it to believe it.** (Not so easy and a bit scary.)

Convergence to Singularity



Integrating Element

Implemented foundational theories will be a **key integrating element**, they will define the theoretical reality of the convergence, and it must match both the **computational reality and its logical rules**.

We can see a glimmer of integrating role if we look at **Logical Programming Environments** (LPE's).

Logical Programming Environments

Our **LPE** consists of subsystems, including:

- Native Logical Language (e.g. type theory)

- Proof System (LCF style interactive provers)

- Evaluation System (for programs)

- Verified Compilers (to external languages)

- Formal Digital Library (specs, code, proofs)

- Structure Editor (programs, tactics, proofs)

- Theory Modification Tools

Design Issues are Key

We know that for programming languages, the design issues are critical to success, and we can see that for LPE's there are more "moving parts" and a new dimension, the **logical reality of computation**.

I want to expose you to this **logical dimension** and show you how the approach I favor arose historically and what the options are.

The design of type theories

We will look at the structure of **Computational Type Theory** (CTT), the theory that provides the logical reality of computation in our LPE. It is closely related to the **Calculus of Inductive Constructions**, the logic of Coq and to Martin-Löf's early type theories, say **Intuitionistic Type Theory** (ITT).

Here is a brief comparison that will raise the concepts that we need to examine.

Points of Comparison

CTT

grounded in semantics (OPLSS10)
(partial equivalence relations)
implicitly typed (polymorphic)
extensional equality
predicative (universes)
Turing-complete and open
proof trees (refinement logic)
recursive type (Mendler)
library is a database
processes are primitive

CIC

grounded in proof theory
(strong normalization)
explicitly typed
intensional equality
impredicative
sub-Turing complete
proof scripts (sequents)
inductive types
library is file system
processes ?

Points of Comparison

CTT

grounded in semantics (OPLSS10)
(partial equivalence relations)
implicitly typed (polymorphic)
extensional equality
predicative (universes)
Turing-complete and open
proof trees (refinement logic)
recursive type (Mendler)
library is a database
processes are primitive

CIC

grounded in proof theory
(strong normalization)
explicitly typed
intensional equality
impredicative
sub-Turing complete
proof scripts (sequents)
inductive types
library is file system
processes ?

Lecture Plan

- I Introduction -- done
- II Historical Background (**my interpretation**)
Evolution of design issues
- III Integrating Role of Computer Science
Semantics of Evidence
- IV Issues in Formal Digital Libraries
Partial functions in constructive theories
Simple proof of Gödel's Incompleteness

Historical Context

This historical context will help us understand the **scale and scope of our enterprise.**

Rutherford notwithstanding, our effort is more than

stamp collecting

Aside

We have the analogues of the great theoretical divide in physics, but ours is unifiable.

Quantum Mechanics **Theory A** – Algorithms

Relativity **Theory B** --Type theory

Historical Backdrop

The research that led to modern type theories was conducted against the backdrop of a “crisis” in mathematics which caused logicians to look at ways to be more precise about basic concepts. Key players were:

Frege

Begriffsschrift (1879)

Russell & Whitehead

Principia Mathematica (PM) (1910)

Cantor

Set Theory (1874)

Zermelo (1908)

Historical Backdrop continued

Type Theory

Set Theory

PM 2nd Edition
1925

Zermelo/Fraenkel (*ZF*)
1922 (first-order)

Gödel thought of *ZF* as a simplification of *PM*.

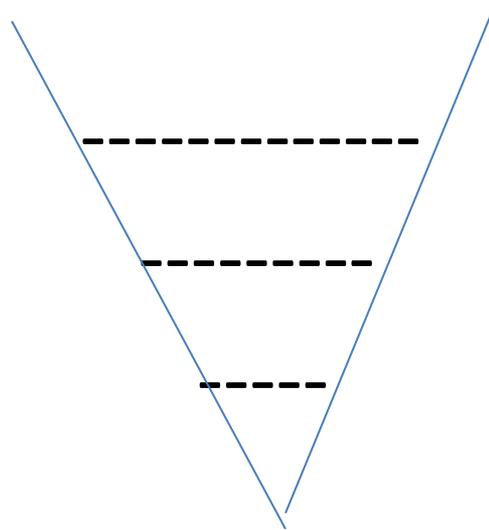
The Success of Set Theory

By the 1940's set theory was well established as a foundation for mathematics. Bourbaki's famous *Elements of Mathematics* is based on set theory and carries out the vision of *Principia*.

The famous **cumulative hierarchy** is a simple model to understand, and it is possible to easily code the basic ideas of nearly all branches of mathematics in set theory, some times using a ninth axiom, **choice**.

Encoding Mathematics into Sets

There is an elegant definition of the natural numbers, \mathbf{N} , as sets, e.g. \emptyset , $\{\emptyset\}$, $\{\emptyset, \{\emptyset\}\}$, This von Neuman encoding can be extended to the ordinals, providing the **spine** of the cumulative hierarchy.



Encoding Mathematics into Sets

Sets can encode numbers, relations, functions, algebraic structures, topological structures, Turing machines, etc.

To define **categories**, one needs a richer set theory than ZF. We introduce classes, and families, and other “large collections” as in the **Tarski-Grothendieck set theory** used in the Mizar proof checking system and its formal mathematics library (with about 50K items).

The Success of Set Theory

The success of set theory in resolving the foundational crisis and providing a simple elegant basis for mathematics is remarkable, a **great intellectual achievement** of the 20th century. It is hard to imagine a more elegant solution, yet it is not an adequate foundation for computer science and informatics. **Type theory** is a strong alternative, adequate for math and computer science. **Why is this so?**

Looking Back on this Period

When we look back on these results, say the combination of **first-order logic** and **ZF axioms**, we see a **remarkable success** in explaining the foundations of mathematics, reducing it to **eight axioms in first-order logic** and teaching set theory from grade school through college.

We'll see in this lecture why type theory is alive and well in computer science and being taught in secondary school as part of programming, but that story **requires a bit more history**.

Why is there a type theory alternative?

To answer this question, we need to go back to the critical year of 1907 and look at Brouwer's influence and his **intuitionistic mathematics**.

We will see sets (species) as one among many interesting types; many other types are very good for structuring computation, as computer scientists later discovered as well.

Along came Brouwer

In 1907 just as “traditional logic” seemed poised to solve the foundational crisis, with the predicate calculus (Frege) and either **types** (Russell) or **sets** (Zermelo), **L.E.J. Brouwer** advanced a wholly different solution, a radical solution.

Brouwer proposed grounding mathematics in computation and viewed logic as a special kind of very general computational mathematics.

Brouwer and Hilbert

The foundational work moved into its **second phase** by 1922 with *PM second edition* and first-order *ZF* set theory and a new element, Hilbert advances in 1927 his 1904 program of **formalization**. He distinguishes between **finitary** and **ideal** objects.

Hilbert proposed treating mathematical theories as pure **formal games**, the **object theory**, to be studied in a **meta-theory** used to justify ideal objects.

Brouwer attacks Hilbert's Formalism

Brouwer thought that the Hilbert Program was misguided and could not succeed. He was winning converts such as his former student Hermann Weyl who said:

“I now give up my own attempt and join Brouwer.” 1920

Brouwer seems radical

Some of Brouwer's positive results seemed very radical, his bar induction, Bar Theorem, Fan Theorem, Continuity Principle. They led to the theorem that **all functions on Reals are uniformly continuous!**

E. Bishop took a less radical approach in his **constructive mathematics**, and Kleene tamed Brouwer in his 1965 book **Foundations of Intuitionistic Mathematics** - major impact.

Lecture Plan

- I Introduction -- done
- II Historical Background (**my interpretation**)
Evolution of design issues
- III Integrating Role of Computer Science
Semantics of Evidence
- IV Issue with Formal Digital Libraries
Partial functions in constructive theories
Simple proof of Gödel's Incompleteness

Computer Science as Unifier

Computer scientists took bits from all three:

Ground meaning in computation – Brouwer

Formalize mathematics to automate it – Hilbert

Use types to organize the data – Russell

We will continue to refine and extend the convergence of these approaches.

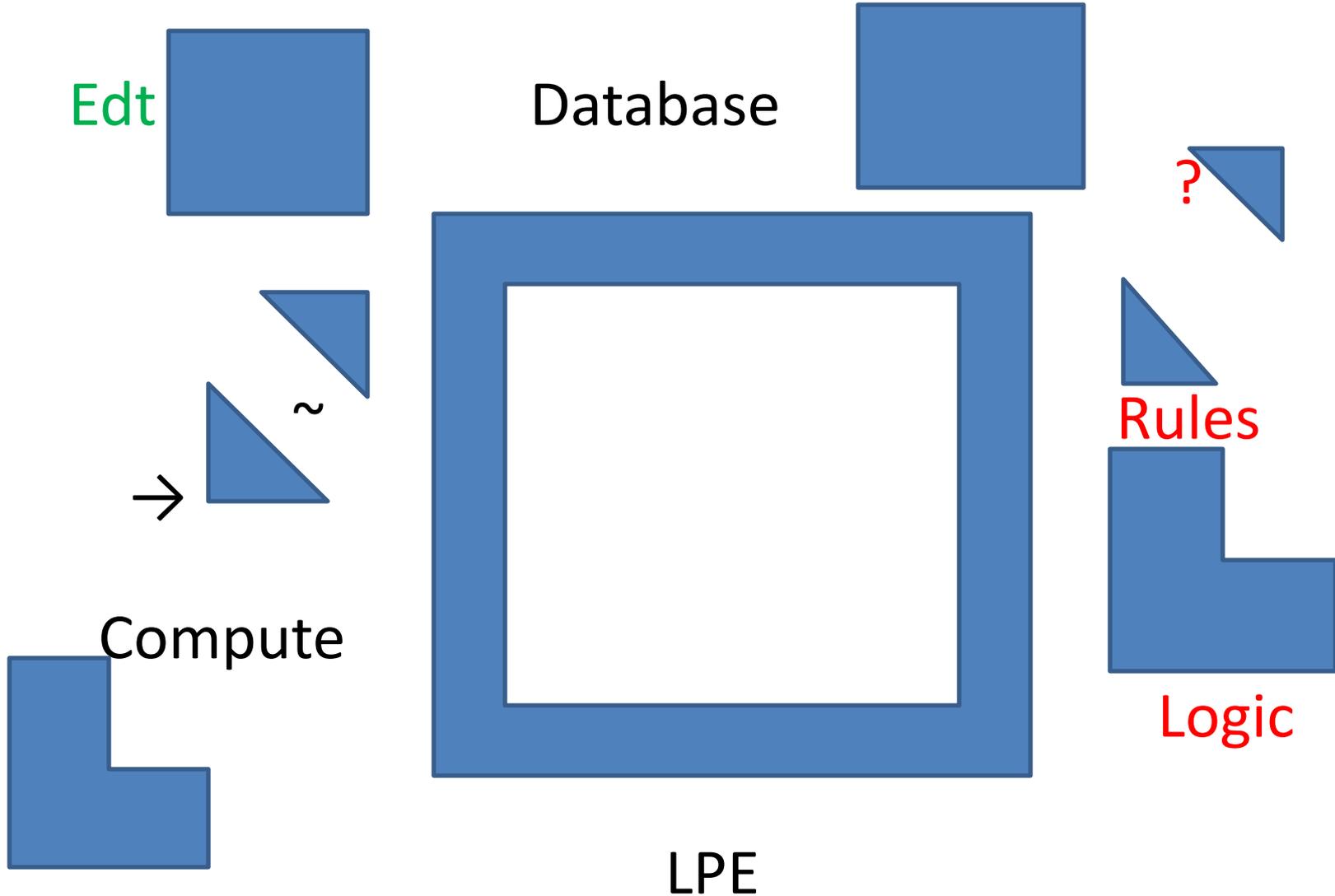
How do these themes fit together?

How did CS manage to bring these three themes together when the originators did not see the convergence?

Type theory emerged because of programming languages, sets were not the whole story of computational data. We'll look at this.

What is the last piece from Brouwer to fit into the modern view? Oddly it is one of his most fundamental ideas. What is it? Why was it last?

The Last Puzzle Piece



Programming Languages and Logics

From the critical **historical juncture circa 1907** there was a split in the genotype of mathematics: **sets** vs **types**.

We know about **sets** from mathematics courses and about **types** from programming courses.

Types also appear in basic CS theory courses such as Algorithms and Data Types.

Increasing Richness of Types in Programming Languages

Fortran started with simple type distinctions, fixed and floating point numbers distinguished by the alphabetical range of the variable names, e.g. k,l,m,n versus x,y,z.

From that meager beginning we have seen a **progressive enrichment of type systems** from Algol, Pascal, Algol 68, PL1, C++, Java, ML, F#, etc.

Standard PL Types

The types we now expect:

Product	$A \times B$	$\langle a, b \rangle$ ordered pairs
Record	$\{a_1:A_1; \dots; a_n:A_n\}$	maps a_i into A_i
Union	$A + B$	$\text{inl}(a), \text{inr}(b)$ inject
Function	$A \rightarrow B$	$\lambda(x.\text{exp})$ functions
Recursive	$\mu X.F(X)$	e.g. lists, trees, etc.

Type Inference Algorithms

The ML type inference algorithm of Robin Milner is an amazing **tool for thought**, widely used in programming courses. Given a function, e.g.

$\text{memb} = \lambda(x, \ell).$

if $\ell = \text{nil}$ **then** *false*

else if $x = \text{hd } \ell$ **then** *true*

else $\text{memb}(x, \text{tl } \ell)$)

ML infers the type of `memb` as: $\alpha \times \alpha \text{ list} \rightarrow \text{bool}$.

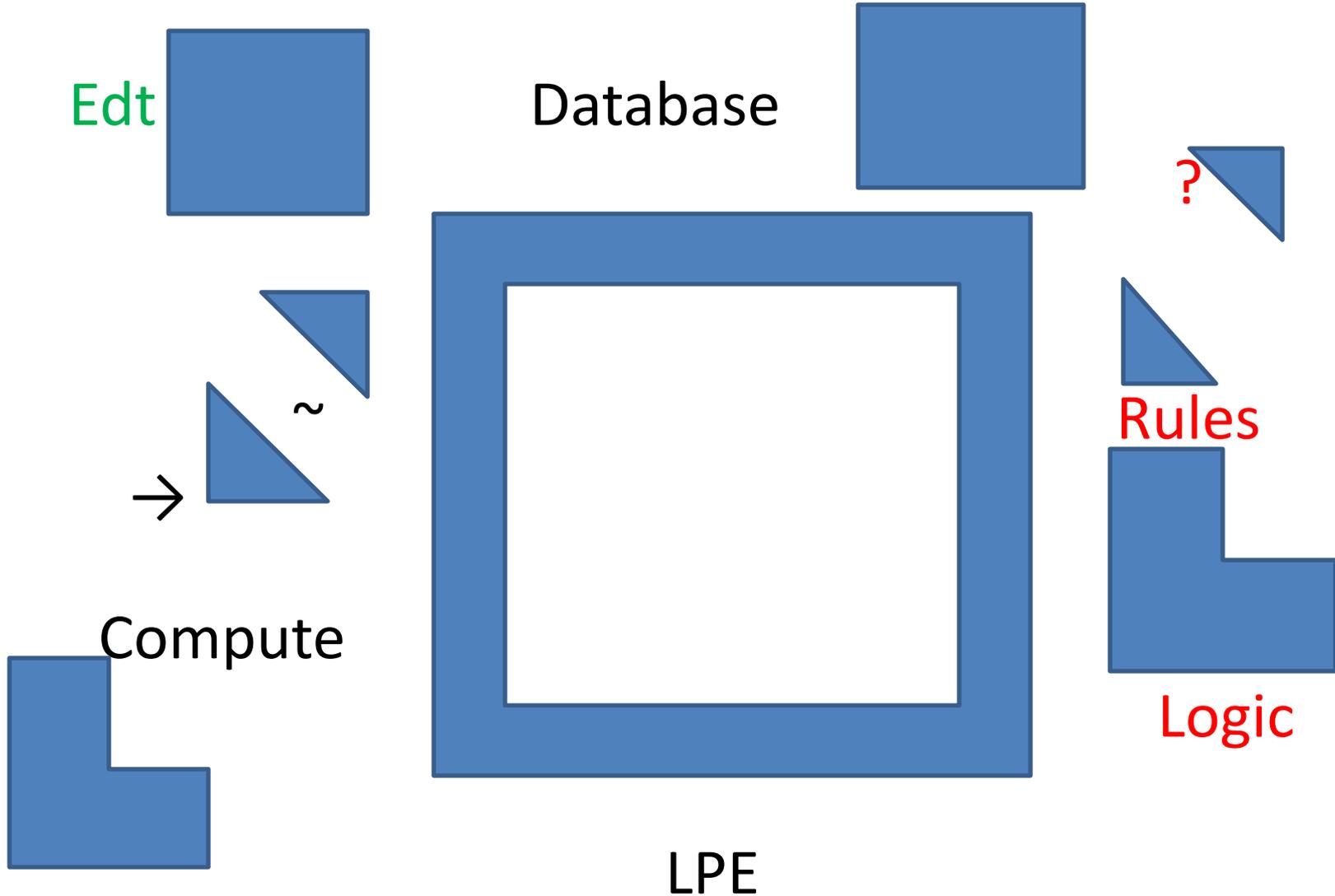
The type is the range of significance of the function.

Is there a limit to type enrichment?

Is there some limit to this process of type enrichment in programming languages?

I claim that it will continue at least until the programming languages have the types needed for constructive mathematics, but it will go beyond that, driven by the open ended nature of computation.

The Last Puzzle Piece



The Last Piece

The last piece to be accepted is Brouwer's most fundamental insight. Logicians call it the Brouwer/Heyting/Kolmogorov semantics.

Computer scientists, those great unifiers, give it a **terrible name**, the Curry-Howard Isomorphism. Sometimes we call it Propositions-as-Types, still not great. I called it proofs-as-programs, still not good. Scott called it constructive validity, good.

The key piece needs a good name

Brouwer's idea is profound and revolutionary. **Most people resist it strongly.** It needs a better name. One that I tried in 1985 seems plausible. I tried to tie the idea to classical logic as well and called it **evidence semantics.**

Let me try “evidence semantics” again here.

Lecture Plan

- I Introduction -- done
- II Historical Background (**my interpretation**)
Evolution of design issues
- III Integrating Role of Computer Science
Semantics of Evidence
- IV Issues in Formal Digital Libraries
Partial functions in constructive theories
Simple proof of Gödel's Incompleteness

Semantics of Evidence

The evidence semantics for a formula A in a model \mathcal{M} is the type of objects a that constitute evidence for A . These objects use elements of the universe $U_{\mathcal{M}}$ along with other elements.

$$[A] = \{a \mid a \text{ is evidence for } A \text{ in } \mathcal{M}\}$$

For example $[0=0]$ is “axiomatic” say, $\{true\}$

Evidence Semantics

The evidence for $0 < 1$ could be “axiomatic” or if we define $x < y$ to mean there is a z such that $x+z = y$ for

$z > 0$ then the evidence for $0 < 1$ is

$[0 < 1]_{\mathcal{M}} = \{ \langle z, \text{pf} \rangle \}$ where $z > 0$ and pf in $[x+z=y]_{\mathcal{M}}$

If A is false, then $[A]_{\mathcal{M}}$ is empty.

Propositional Evidence

Suppose that we have **evidence types** $[A]$ for the atomic propositions A . Here is how to construct evidence for compound formulas in a model \mathcal{M} .

$$[A \ \& \ B] \quad == \quad [A] \times [B]$$

$$[A \ \vee \ B] \quad == \quad [A] + [B]$$

$$[A \Rightarrow B] \quad == \quad [A] \rightarrow [B]$$

$$[\text{False}] \quad == \quad \phi \quad (\text{the empty set})$$

$$[\neg A] \quad == \quad [A] \rightarrow \phi$$

Evidence for Quantified Propositions

The evidence for quantified formulas requires **dependent types** over the universe $U_{\mathcal{M}}$ of the model \mathcal{M} :

$$[\text{All } x. B(x)] \quad == \quad x: U_{\mathcal{M}} \rightarrow [B(x)]$$

$$[\text{Exists } y. B(y)] \quad == \quad y: U_{\mathcal{M}} \times [B(y)]$$

Examples

What is $[A \Rightarrow A]$? $\{\lambda(x.x), \lambda(x. <17,x>.2), \dots\}$

What is $[A \ \& \ B \Rightarrow A]$?

$\{\lambda(x.\text{spread}(x; a, b. a)), \dots\}$

What is $[A \Rightarrow A \vee B]$?

$\{\lambda(x.\text{inl}(x)), \lambda(x.\text{inl}(\text{spread}(x, 17); \text{lft}, \text{rgt. lft})), \dots\}$

What is $[(0=0) \Rightarrow \text{True} \vee (0=0)]$?

$\{\lambda(x.\text{inl}(\text{true})), \lambda(x.\text{inr}(x)), \dots\}$

More Examples

For the domain $U_{\mathcal{M}}$ the natural numbers \mathbb{N} :

What is $[\exists x. (x > 0)]$?

$\{ \langle 1, \text{true} \rangle, \langle 2, \text{true} \rangle, \dots \}$

What is $[\forall x. \exists y. (x \leq y)]$?

$\{ \lambda(x. \langle x, \text{true} \rangle), \lambda(x. \langle x+1, \text{true} \rangle), \lambda(x. \langle x+2, \text{true} \rangle), \dots \}$

Common to School Lectures

The idea has already been used extensively in the school lectures under the other names such as **propositions as types**.

I will teach more about the idea in tomorrow's lectures and extend it further.

Evidence Semantics is Meaningful for Classical and Constructive Logics

For constructive (and intuitionistic) logics this evidence semantics is essentially the Brouwer, Heyting, Kolmogorov semantics (BHK) from 1907 to 1935.

We see that it is “applicable” to classical logics as well if we introduce an object that can be used as evidence for the law of excluded middle, $(P \vee \neg P)$. I call that element **magic(P)**.

Examples of magic

For the domain of **propositions** of level i , Prop_i

$[\text{All } P. (P \vee \neg P)]$ is $\{\lambda(P.\text{magic}(P)), \dots\}$

where **magic**(P) is an **oracle** computing either $\text{inl}(p)$ or $\text{inr}(p)$ for some evidence depending on P . We can stipulate that the evidence p is hidden.

Importance of Propositions as Types

If we adopt **an evidence semantics**, then proof terms denote evidence. They can be incorporated into the logic as terms, and if the logic is constructive, these **terms are part of a computation system**, so they can be evaluated.

This principle shows that in a sense, the dependent type system is a first limit or “fixed point” for programming languages and logics – classical or constructive.

Proofs as Programs

For a constructive logic, **all proof terms** can be considered as programs or data that reveal the implicit computational content and make it explicit.

The proof assistants based on computational type theory or other constructive theories can evaluate these proof terms and thus **extract the computational content**. For classical logic only some proof terms are computable.

Relationship to Truth Semantics

We can relate the semantics provided by treating propositions as types to the usual **truth semantics** made precise by Tarski.

Brouwer, Bishop and others would ask whether this is **meaningful**. You need to decide for yourself. It provides a **bridge**, perhaps a one way bridge to constructivity.

Standard Tarski Semantics

The standard semantics for a logical formula A is that A is true about a model \mathcal{M} if the truth value of A , $\mathbf{tr}(A) = true$ in \mathcal{M} . This is called the **Tarski semantics**.

For example, $\mathbf{tr}(A \& B) = true$ exactly when

$$\mathbf{tr}(A) = true \text{ and } \mathbf{tr}(B) = true$$

Also $\mathbf{tr}(A \rightarrow B)$ is $\mathbf{tr}(\neg A \vee B)$ as well as

$\mathbf{tr}(\text{All } x.B(x))$ is $\mathbf{tr}(B(a))$ all a in \mathcal{M} .

Footnote (Aside)

Some constructivists react badly to Tarski semantics, e.g. Girard, calling it meaningless, since it is defines **formal logical truth** in terms of logical truth.

But to others, this shows that the classical formal semantics is faithful, exactly right.

Relationship to Tarski Semantics

It is easy to relate evidence semantics to Tarski's semantics for first-order logic and show that a formula A is true in a model \mathcal{M} iff and only if there is evidence for A . Moreover, if pf is a proof of A , then we can define an evidence semantics for pf , say $[\text{pf}]$, such that:

$$\begin{aligned} \text{pf} \Vdash A & \text{ iff } [\text{pf}]_{\mathcal{M}} \varepsilon [A]_{\mathcal{M}} \\ \models_{\mathcal{M}} A & \text{ iff } a \varepsilon [A]_{\mathcal{M}} \end{aligned}$$

The Curry Howard Isomorphism

Many computer scientists like to call the “PAT” semantics the **Curry-Howard isomorphism**, even though it is not an isomorphism nor was it invented by either Curry or Howard.

The book on this topic, **Lectures on the Curry-Howard Isomorphism** by Sørensen and Urzyczyn, lists 23 contributors to the principle, the main ones being Brouwer, Heyting, Kolmogorov, Kleene, deBruijn, Curry, Howard, Girard, Scott, Martin-Löf, starting by 1907.

Propositions as Types

In my view this principle is one of the **key discoveries** of logic and computer science. It was created informally by Brouwer (influenced by Kant?) and germinated in *Principia Mathematica* which **brought logic and type theory together**.

This semantic principle has implications for philosophy and linguistics as well as informatics and logic.

Applications in Programming

We can use the constructive semantics to **program with proofs**. This can be done **efficiently**, providing documentation for correctness of the programs. The method is also called **correct by construction programming** (I think that terminology was used first by Cordell Green for his company Kestrel).

This application will be a focus of the Technical Lectures.

Formal Mathematics as a Programming Environment

This connection between proofs and programs suggests that an **implemented formal theory** of mathematics with **programs** and other **fundamental computer science concepts** can serve as a Programming Environment. I like the term **Logical Programming Environment (LPE)**.

Lecture Plan

- I Introduction -- done
- II Historical Background (**my interpretation**)
Evolution of design issues
- III Integrating Role of Computer Science
Semantics of Evidence
- IV Issue with Formal Digital Libraries
Partial functions in constructive theories
Simple proof of Gödel's Incompleteness

Formal Digital Libraries

The proof assistants are used to build large libraries of formalized mathematics and programming. The study of these libraries has become a research area called **Mathematical Knowledge Management (MKM)**

There are already many important results and problems in this young field, and the subject advances the goals of *PM*.

MKM Issues and Problems

Here are two important problems from MKM for which we have results.

1. How can provers based on different logics **share results**, e.g. constructive with classical?
2. How can multiple users modify a library without interfering with each other?

Sharing Formal Mathematics

As deBruijn's **Automath** project showed, it is possible to build proof terms for any logic, and this is a design principle, **proofs as terms**. Thus there is an evidence semantics for classical logic as well. Most of the proof terms of classical logic have computational meaning.

For non-constructive axioms such as $P \vee \neg P$, we use the term **magic(P)** as the proof term; it does not necessarily have computational meaning.

Gödel Translations

Gödel showed that it is possible to translate classical number theory, **Peano Arithmetic (PA)**, into intuitionistic number theory, **Heyting Arithmetic (HA)**, thereby establishing that if HA is consistent, so is PA. The two theories differ only in that PA obeys the law of excluded middle, i.e.

$$PA = HA + (P \vee \neg P)$$

Gödel Translations continued

For instance, the translation of $(P \vee \neg P)$ in HA is just $\neg(P \ \& \ \neg P)$, which is the same as

$$\neg\neg (P \vee \neg P).$$

In general, Gödel translates formula F of PA into $\neg\neg F$ and shows

$$(PA \vdash F) \text{ implies } (HA \vdash \neg\neg F)$$

Embedding Classical Theories into Constructive

Gödel's results show how to embed classical theories such as PA and ZF into their constructive analogues, by defining new logical operators

$P \mid Q \iff \neg(\neg P \ \& \ \neg Q)$ classical or

$\text{Ex.}P(x) \iff \neg(\text{All } x. \neg P(x))$ classical exists

Translating Among Theories

These Gödel translation results show how to relate classical and constructive theories.
Peano Arithmetic (PA) and ZF set theory factor as

$$\text{PA} = \text{HA} + (\text{P} \vee \neg\text{P})$$

$$\text{ZF} = \text{IZF} + (\text{P} \vee \neg\text{P})$$

$$\text{ZFC} = \text{IZF} + \text{Choice}$$

Caution about Gödel Translations

As the results of Curien and Herbelin show, we need to rethink the embedding results when we use them in constructive type theories because the **existential quantifier is stronger in type theory**. It is a projection, moreover in CTT and ITT it respects equality.

Some Theories Do not Factor

We will see an example of **constructive domain theory which does not factor this way**, and that fact raises interesting research questions about partial functions.

Exercise

Prove $\neg\neg (P \vee \neg P)$ and produce the evidence term in $[\neg\neg (P \vee \neg P)]$.

Recall that the elements of $(P \vee \neg P)$ have the form $\text{inl}(p)$ or $\text{inr}(\lambda(x.np))$, elements of a disjoint union type, noting that $[\neg P]$ is the function space from $[P]$ into the empty type.

Constructive Domain Theory

We consider an extension of HA to include partial recursive functions, say **Constructive Scott Arithmetic** (CSA) based on LCF over the type \mathbb{N} .

We take the basic type to be $\tilde{\mathbb{N}}$, those computable terms which have a natural number value iff they converge.

Computability in CSA

Here is how to define **general recursive functions**. Consider the **$3x+1$ function** with natural number inputs.

$f(x)$ = if $x=0$ then 1
 else if $\text{even}(x)$ then **$f(x/2)$**
 else **$f(3x+1)$**
 fi
fi

Using Lambda Notation

$f = \lambda(x. \text{if } x=0 \text{ then } 1$
 $\text{else if even}(x) \text{ then } f(x/2)$
 $\text{else } f(3x+1))$

Here is a related term with function input f

$\lambda(f. \lambda(x. \text{if } x=0 \text{ then } 1$
 $\text{else if even}(x) \text{ then } f(x/2)$
 $\text{else } f(3x+1)))$

The recursive function is computed using this term.

Defining General Recursive Functions

```
fix(λ(f. λ(x. if x=0 then 1
           else if even(x) then f(x/2)
           else f(3x+1)
           fi
           fi)))
```

Recursion in General

$f(x) = F(f,x)$ is a recursive definition, also
 $f = \lambda(x.F(f,x))$ is another expression of it, and the
CTT definition is:

$$\text{fix}(\lambda(f. \lambda(x. F(f,x)))$$

which reduces in one step to:

$$\lambda(x.F(\text{fix}(\lambda(f. \lambda(x. F(f,x))))),x))$$

by substituting the **fix term** for f in $\lambda(x.F(f,x))$.

Non-terminating Computations

CTT defines **all general recursive functions**,
hence non-terminating ones such as this

$$\text{fix}(\lambda(x.x))$$

which in one reduction step **reduces to itself!**

This system of computation is a simple
functional programming language.

Unsolvable Problems

It is remarkable that we can prove that there is no function in CTT that can solve the convergence problem for elements of basic bar types.

We can show this for non empty type \bar{A} with element \bar{a} that converges in A for basic types such as Z , N , $\text{list}(A)$, etc. We rely on the typing that if F maps \bar{A} to \bar{A} , then $\text{fix}(F)$ is in \bar{A} .

Unsolvable Problems

Suppose there is a function h that decides halting. Define the following element of \tilde{N} :

$$d = \text{fix}(\lambda(x. \text{if } h(x) \text{ then } \uparrow \text{ else } 0 \text{ fi}))$$

where \uparrow is a diverging term, say $\text{fix}(\lambda(x.x))$.

Now we ask for the value of $h(d)$ and find a contradiction as follows:

Generalized Halting Problem

Suppose that $h(d) = \text{true}$, then according to h , d converges, but according to its definition, the result is the diverging term \uparrow because by computing the fix term for one step, we reduce

$$d = \text{fix}(\lambda(x. \text{if } h(x) \text{ then } \uparrow \text{ else } 0 \text{ fi}))$$

to $d = \text{if } h(d) \text{ then } \uparrow \text{ else } 0 \text{ fi} .$

If $h(d) = \text{false}$, then d converges to 0.

Proving Undecidability

We can add the predicate $\text{Conv}(n)$ for any n in $\tilde{\mathbb{N}}$, asserting that the element n converges.

Suppose we could prove in CSA the following Convergence Theorem (CvT).

CvT: All $n:\tilde{\mathbb{N}}$. [$\text{Conv}(n) \vee \neg\text{Conv}(n)$].

Then we could **extract** a computable function

$h: \tilde{\mathbb{N}} \rightarrow \text{Bool}$. All $n:\tilde{\mathbb{N}}$. ($h(n)=\text{true}$ iff $\text{Conv}(n)$).

Incompatibility

Thus, we cannot prove CvT in CSA (because it is not true), indeed, we just **proved \neg Conv.**

If we try to add $(P \vee \neg P)$ to CSA, we do not obtain a sensible classical domain theory. Indeed we can then prove CvT. So the two theories are incompatible, inconsistent. They can not live in the same Formal Library.

Unsolvability and Incompatibility

Thus, we cannot prove CvT in CSA (because it is not true), indeed, we just **proved \neg CvT**.

In a classical version of CSA, say SA, we can prove CvT by the law of excluded middle. So we cannot factor

$$SA \neq CSA + (P \vee \neg P).$$

The two theories are incompatible and can not live together in the same library of results.

Other Foundational Issues

There are many questions in the **design space** for implemented formal theories, especially those that are implemented in proof assistants and used for programming.

Key Design Issues

In the paper for this lecture, I outline some of the most **critical design issues**. They are:

1. Predicativity, orders, universes: where is impredicativity safe? CIC theorists know cases
2. When to use extensional versus intensional equality? CTT is one of the only extensional type theories, what are its pluses and minuses?
3. Turing completeness of the computation system versus subrecursive computation systems versus open systems (Brouwerian), how to integrate?

Key Design Issues

4. How to develop a **theory of partial computable functions** that is useful in computing, e.g. for the semantics of programming languages as in Karl Crary's PhD thesis giving a semantics to ML building on Scott Smith's thesis on partial functions in CTT.

Another design goal might be to avoid contradicting Church's Thesis or depending on it.

Conclusion

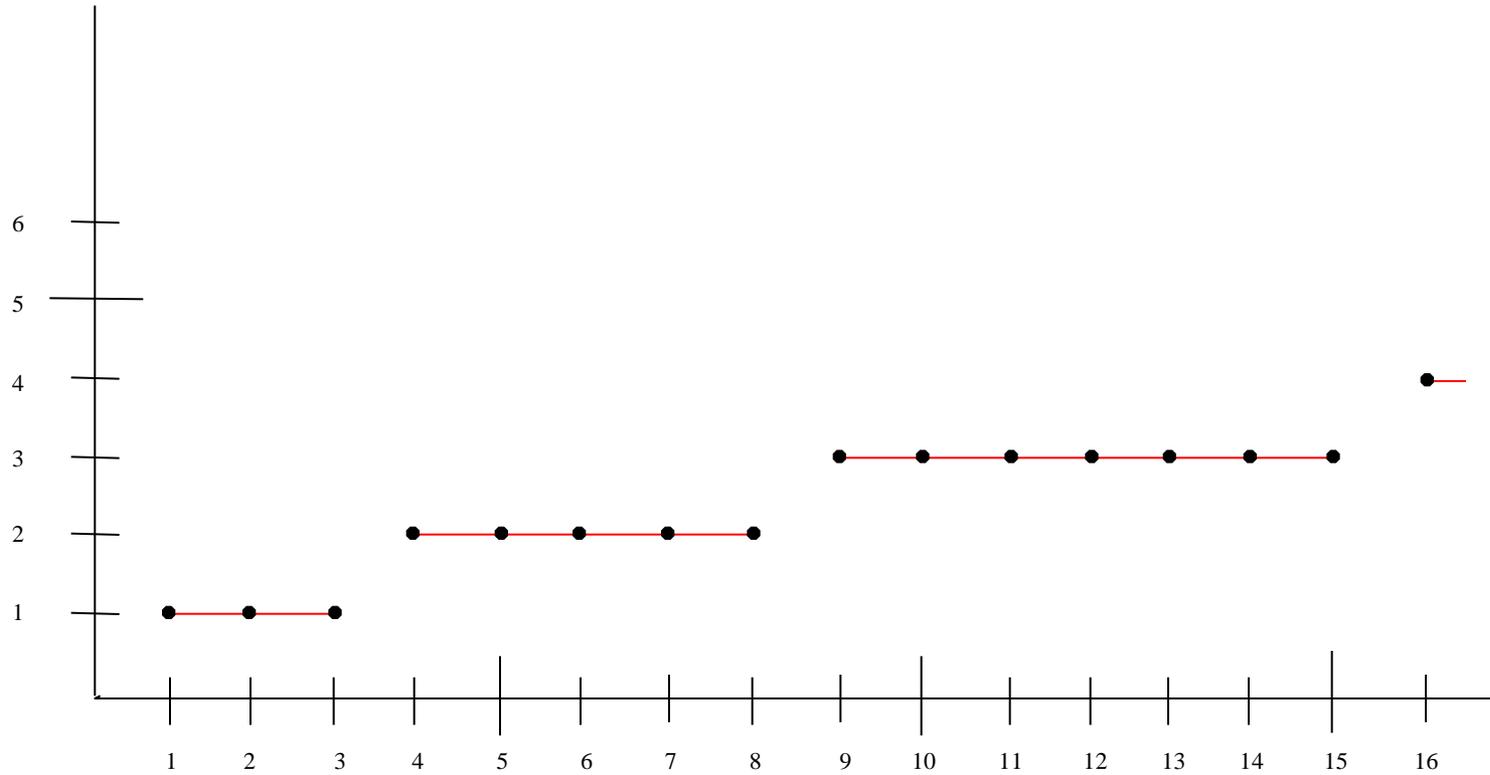
Principia Mathematica and *Brouwer* set off a chain reaction of investigations and discoveries in type theory leading to new areas of logic, informatics, and mathematics. I see discoveries accelerating, in part because the PL community has widely embraced type theory and formal methods, and the best compiler writers will make these implemented formal theories produce good code.

THE END

Appendix

1. An example of programming by extraction in Heyting Arithmetic over the Integers (\mathbf{Z}).
2. Examples of Refinement Logic Rules
3. Type theory as a foundation for mathematics, Voevodsky efforts.

Integer Square Root



Proof of Root Theorem

$\forall n : \mathbb{N}. \exists r : \mathbb{N}. r^2 \leq n < (r + 1)^2$

BY `allR`

$n : \mathbb{N}$

$\vdash \exists r : \mathbb{N}. r^2 \leq n < (r + 1)^2$

BY `NatInd 1`

... .induction case....

$\vdash \exists r : \mathbb{N}. r^2 \leq 0 < (r + 1)^2$

BY `existsR [0]` THEN `Auto`

... .induction case....

$i : \mathbb{N}^+, r : \mathbb{N}, r^2 \leq i - 1 < (r + 1)^2$

$\vdash \exists r : \mathbb{N}. r^2 \leq i < (r + 1)^2$

BY `Decide [r + 1^2 ≤ i]` THEN `Auto`

Proof of Root Theorem (cont.)

.....Case 1.....

$$i : \mathbb{N}^+, r : \mathbb{N}, r^2 \leq i - 1 < r + 1^2, r + 1^2 \leq i$$

$$\vdash \exists r : \mathbb{N}. r^2 \leq i < r + 1^2$$

BY `existsR [r + 1]` THEN `Auto'`

.....Case 2.....

$$i : \mathbb{N}^+, r : \mathbb{N}, r^2 \leq i - 1 < r + 1^2, \neg r + 1^2 \leq i$$

$$\vdash \exists r : \mathbb{N}. r^2 \leq i < r + 1^2$$

BY `existsR [r]` THEN `Auto`

The Root Program Extract

Here is the **extract term** for this proof in ML notation with **proof terms** (pf) included:

```
let rec sqrt i =  
  if i = 0 then < 0, pf0 >  
  else let < r, pfi-1 > = sqrt i - 1  
  in if r + 12 ≤ n then < r + 1, pfi >  
  else < r, pfi' >
```


Refining Universal Statements

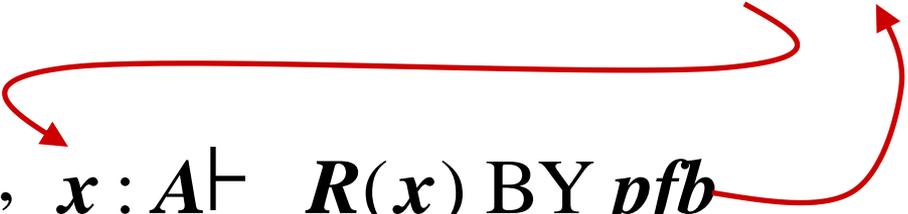
$\bar{H} \vdash \forall x : A. R(x)$ BY $\text{all}R(x. pfb)$

$\bar{H}, x : A \vdash R(x)$ BY pfb

Refining Universal Statements

$\bar{H} \vdash \forall x : A. R(x)$ BY $\text{all}R(x. pfb)$

$\bar{H}, x : A \vdash R(x)$ BY pfb



Constructive Semantics

Notice that the proof term corresponding to

$$\forall x : A. R \text{ is } \mathit{all}R(x.pf)$$

This should denote an element of $x : A \rightarrow R_x$

namely $\lambda(x.pf)$.

In the constructive case, this function should be computable. We get this result when the **evidence sets are types**.

Homotopy Interpretation of Constructive Type Theory

Field's medalist mathematician **Vladimir Voevodsky** is organizing meetings on the use of constructive type theory as a foundation for algebraic geometry, see Oberwolfach meeting Feb-March 2011.