Proof Assistants and The Rise of Type Theory: Circa 1912 – 2012

Robert L. Constable
Cornell University
Mach 19, 2012
Lecture at Carnegie Mellon University

Lecture Plan

We'll look at the role of proof assistants in the rise of type theory from the publication of *Principia Mathematica* to the present day.

My plan is to show features of the Nuprl and MetaPRL proof assistants and connect them to major ideas in a century long development of type theory, especially constructive type theory.

The Basic Questions

Why is the use of interactive proof assistants on the rise? Consider this lot of them:

Agda, Coq, HOL, Isabelle HOL (HOL), MetaPRL, Minlog, Nuprl, PVS, Twelf, and others.

Why are they all based on type theory, not set theory? Why constructive? The red ones are constructive -- there is a red HOL dialect. (ACL2 is constructive, but not a type theory.)

What is a Proof Assistant?

They are interactive software systems that help users create proofs in a formalized mathematical theory.

These formal theories arose in logic over the last century. The first implementations of simple fragments were in the 1950s, Davis, Newell, Shaw, Simon, Gelernter in the US, Prawitz in Sweden.

Why are there formal theories?

This is a fascinating question to which I'll give a narrow answer tracing the "main line of logic", a technical answer.

The bigger picture is about AI and the over-arching goal of computer science to automate intellectual processes and build a strong symbiosis between people and machines. It is about remarkable intellectual accomplishment of CS. For fun, see *Darwin among the Machines* by G. Dyson

Logical Context

Consistency Questions and Logic

Euclid's 5th postulate

Computing with infinite series and infinitesimals reveals contradictions (19thC)

then infinite sets and the paradoxes

cause increasing concern.

These days, software errors are of great concern as are vulnerabilities in cyber warfare.

Logical Landmark One

Begriffsschrift 1879

Gottlob Frege

He invents first-order logic, predicate calculus -- a precise language for concept writing (Begriffsschrift).

-- A is sensible

|- A is provable

Frege Advances Leibniz's Vision

Another way to see this advance is that Frege did what Leibniz aspired to do already in the 17th century, create a basic logic for coding all knowledge. He anticipated Gödel numberings.

Many modern logicians are in the Leibniz genealogy, and we use "monads."

Logical Landmark Two
The Axiomatic Approach

First there was the relative consistency approach, e.g. non-Euclidean geometries.

In 1899 Hilbert used the Axiomatic Approach to Geometry, remove intuition. Peano Axioms in 1889, Hilbert 1900 axioms for the Reals.

1908	1908
Russell Types	Zermelo Sets

Logical Landmark Three

In 1910, Whitehead and Russell published Volume I of their three volume *Principia Mathematica*, a comprehensive logical foundation for mathematics. It was not completely formal, but Newell, Shaw and Simon drew their examples from it.

The logic was Russell's Type Theory.

Consider the MetaPRL Proof Assistant

MetaPRL like Isabelle and Twelf is a Logical Framework designed by Jason Hickey for his Cornell PhD 2001 and implemented in O'Caml by him and extended by other students in the PRL group, A. Nogin, A. Kopylov, and others in Russia.
Let's open the prover. What do we see?

MetaPRL

\- \-

\- \-

Type Theory Axioms — CZF Set Theory Axioms

PRL Group — Peter Aczel

MetaPRL Offers a Choice of Theories

Interestingly these choices are connected due to a fundamental result of Peter Aczel showing how to embedded CZF into Type Theory.

This embedding relies on the use of recursive types in constructive type theory. These types create great expressiveness for both Nuprl and Coq. These types came from Nax Mendler's 1987 Cornell PhD thesis.

Recursive Types

Recursive types are good data structures, e.g. can define lists of type A as

$$List(A) = Unit + (A \times List(A)).$$

We can define numbers as

$$Nat = Unit + Nat$$

The general form is

$$T = F(T) \text{ for } F:Type \rightarrow Type$$

for F a monotone function on Type, e.g.

$$X \text{ subt } Y \text{ implies } F(X) \text{ subt } F(Y)$$

CZF Sets form a Recursive Type

Sets are embedded into constructive type theory using the recursive type

$$\text{Sets} = B{:}\text{Type} \times (B \to \text{Sets})$$

The axioms of CZF are validated using the axioms of type theory, and every theorem can be interpreted as a result in type theory.

We can see sets as one kind of type, a very rich data type or a mathematical type.

Very Rich Type Theories

Very rich type theories are appealing because they facilitate the formalization of concepts. In this case, CZF is two axioms away from ZFC, a main stream foundation for mathematics.

On the other hand, a consistency guarantee is harder, say compared to ACL2 a modern proof assistant that does not use type theory but instead Recursive Arithmetic.

Other Nice Embeddings

MetaPRL can define theories like ACL2 and implement a famous result of Gödel that it is possible to translate Peano Arithmetic, PA, a classical first-order theory of numbers, into Heyting Arithmetic, HA, a constructive first-order theory of numbers close to ACL2.

Gödel thus showed that PA is consistent iff HA is consistent. This kind of result led him to think that Hilbert's program was doomed.

Other Key Types

The constructive type theory of Nuprl depends on other basic constructors such as quotient types to change the base equality on types, e.g. on **Z** introduce equality mod n, **Z**//mod(2), define Bag(T) as a List(T)//permutations.

Nuprl also hides computational content using set types {x:A| P(x)}, access to the proof of P(x) is not available; it was produced then hidden. Nuprl can save carrying around unnecessary information.

What else can we do with MetaPRL?

We can also formulate incompatible theories in logical frameworks. For example, in Nuprl, our theory of partial recursive functions using bar types is incompatible with classical mathematics whereas most of Nuprl is compatible. So MetaPRL can isolated those results, which are a constructive version of Scott's domain theory, related to Edinburgh LCF.

What else can we do with MetaPRL?

We can read formal mathematics and the “glossing of the theorems”, all the facts are there. Some proofs are extraordinarily clear, like crystals.

Let’s read one.

Also note:

We can automatically translate some theorems and proofs into natural language, a distinctly AI feature.

Math Library

PRL PROJECT

"Proof/Program Refinement Logic"

Math Library

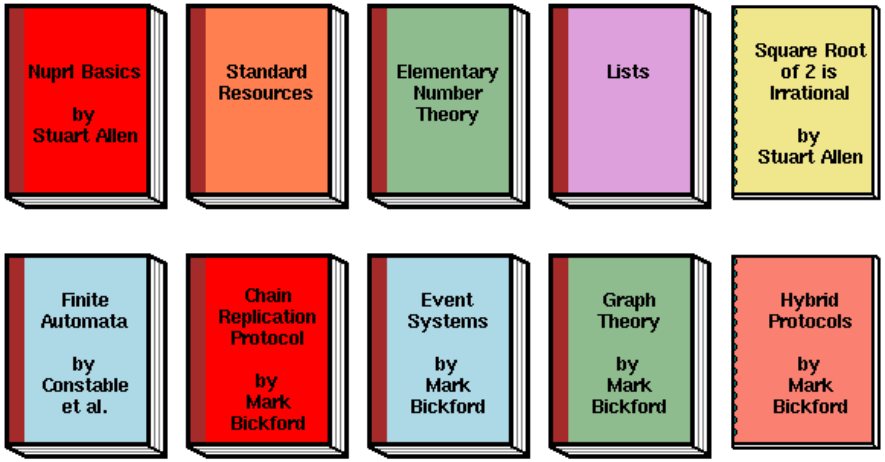
Selections from the PRL Math Library

This is a library of fully formalized definitions and proofs. Some of the material has been enhanced by the addition of informal explanations referring to it.

How to browse the library...

There is also a experimental automatic web projection project for the FDL content

The Library



Stamps

At: stamps sfa

$\vdash\ \forall i{:}\{8...\}.\ \exists m,n{:}\mathbb{N}.\ 3\cdot m+5\cdot n = i$

By: NSubsetInd Concl

Generated subgoals:

1 1. $i : \mathbb{Z}$ — 1 step
2. $0<i$
3. $8 = i$
$\vdash\ \exists m,n{:}\mathbb{N}.\ 3\cdot m+5\cdot n = i$

2 1. $i : \mathbb{Z}$ — 5 steps
2. $8<i$
3. $\exists m,n{:}\mathbb{N}.\ 3\cdot m+5\cdot n = i-1$
$\vdash\ \exists m,n{:}\mathbb{N}.\ 3\cdot m+5\cdot n = i$

About:

ℤ [natural] a+b a-b u = v ∈ A ∀x:A. B(x) ∃x:A. B(x)

Stamps 2

At: stamps sfa 2

---

1. $i : \mathbb{Z}$
2. $8 < i$
3. $\exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i - 1$

$\vdash\ \exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i$

By: ExistHD Hyp:-1

Generated subgoal:

1 3. $m : \mathbb{N}$ 4 steps
4. $n : \mathbb{N}$
5. $3 \cdot m + 5 \cdot n = i - 1$

$\vdash\ \exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i$

About:

ℤ [natural] a+b a-b a·b a<b u = v ∈ A ∃x:A. B(x)

Stamps 2 1

At: stamps sfa 2 1

---

1. $i : \mathbb{Z}$
2. $8 < i$
3. $m : \mathbb{N}$
4. $n : \mathbb{N}$
5. $3 \cdot m + 5 \cdot n = i - 1$

$\vdash \exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i$

By: Decide: $n = 0$

Generated subgoals:

1 6. $n = 0$ — 2 steps

$\vdash \exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i$

2 6. $\neg n = 0$ — 1 step

$\vdash \exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i$

About:

ℤ [natural] a+b a-b a·b a<b u = v ∈ A ∃x:A. B(x)

Stamps 2 1 2

At: stamps sfa 2 1 2

---

1. $i : \mathbb{Z}$
2. $8 < i$
3. $m : \mathbb{N}$
4. $n : \mathbb{N}$
5. $3 \cdot m + 5 \cdot n = i - 1$
6. $\neg n = 0$

$\vdash \exists m,n:\mathbb{N}.\ 3 \cdot m + 5 \cdot n = i$

By: Witness: $m+2 \mid n-1$

Generated subgoals:

None

About:

$\mathbb{Z}$ [natural] a+b a-b a·b a<b u = v ∈ A ∃x:A. B(x)

Looking Closely at a Proof

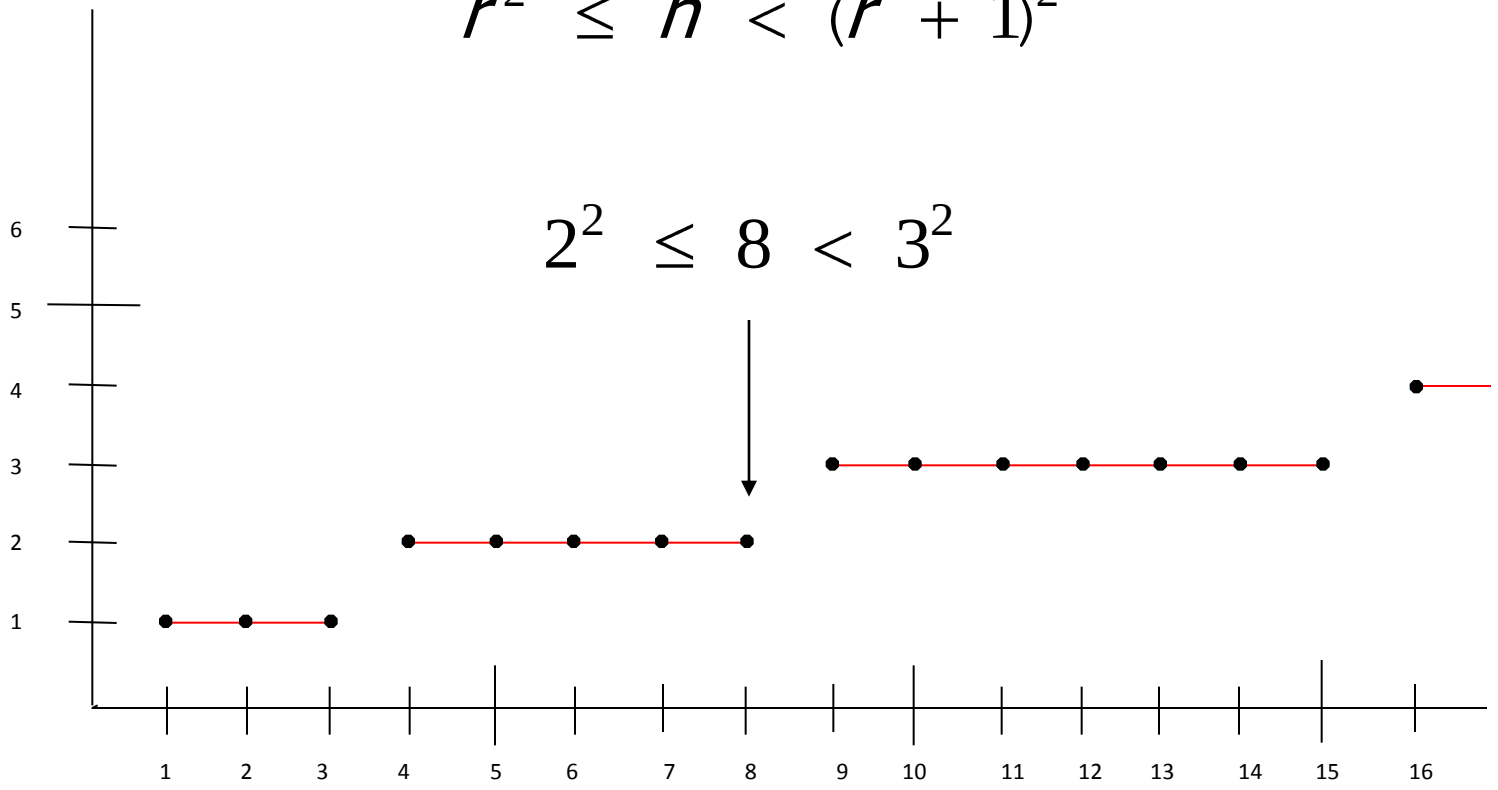
Let's look more closely at how a proof is represented in Nuprl and MetaPRL. We'll take a simple example with interesting computational content.

First we look at the normal textbook style proof, and then at the proof tree presentation, also called tableau style or refinement style proof. All these styles are very readable.

Integer Square Root

$$r^2 \leq n < (r+1)^2$$

$$2^2 \leq 8 < 3^2$$



Theorem $\forall n: \mathbb{N}.\ \exists r: \mathbb{N}.$ Root (r,n)

Proof by induction

Base n = 0 take r = 0, clearly Root (0,0)

Induction assume $\exists r: \mathbb{N}$.Root (r, n-1)

Choose r_0 **where** Root(r_0, n-1), i.e. $r_0^2 \leq n\text{-}1 < (r_0+1)^2$

$(r_0+1)^2 \leq n$ v $n<(r_0+1)^2$

case $(r_0+1)^2 \leq n$ **then** $r = (r_0+1)$

$(r_0+1)^2 \leq n < ((r_0+1)^2 < (r_0+2)^2)$

case $n < (r_0+1)^2$ **then** $r=r_0$ since $r_0 \leq n\text{-}1 < n$.

Qed

Proof of Root Theorem

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

BY allR

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

BY NatInd 1

. . . . .base case.....

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

BY existsR [0] THEN Auto

. . . . .induction case .....

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

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

BY Decide [$r + 1^2 \le i$] THEN Auto

Proof of Root Theorem (continued)

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

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

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

BY `existsR` $\lceil \underline{r} + 1 \rceil$ THEN `Auto'`

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

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

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

BY `existsR` $\lceil \underline{r} \rceil$ THEN `Auto`

A Recursive Program for Integer Square Roots

Here is a very clean functional program

$$
\begin{aligned}
r(n) := \ & \textbf{if } n = 0 \textbf{ then } 0 \\
& \textbf{else let } r_0 = r(n-1) \textbf{ in} \\
& \textbf{if } (r_0 + 1)^2 \leq n \textbf{ then } r_0 + 1 \\
& \qquad\qquad \textbf{else } r_0 \textbf{ fi} \\
& \textbf{fi}
\end{aligned}
$$

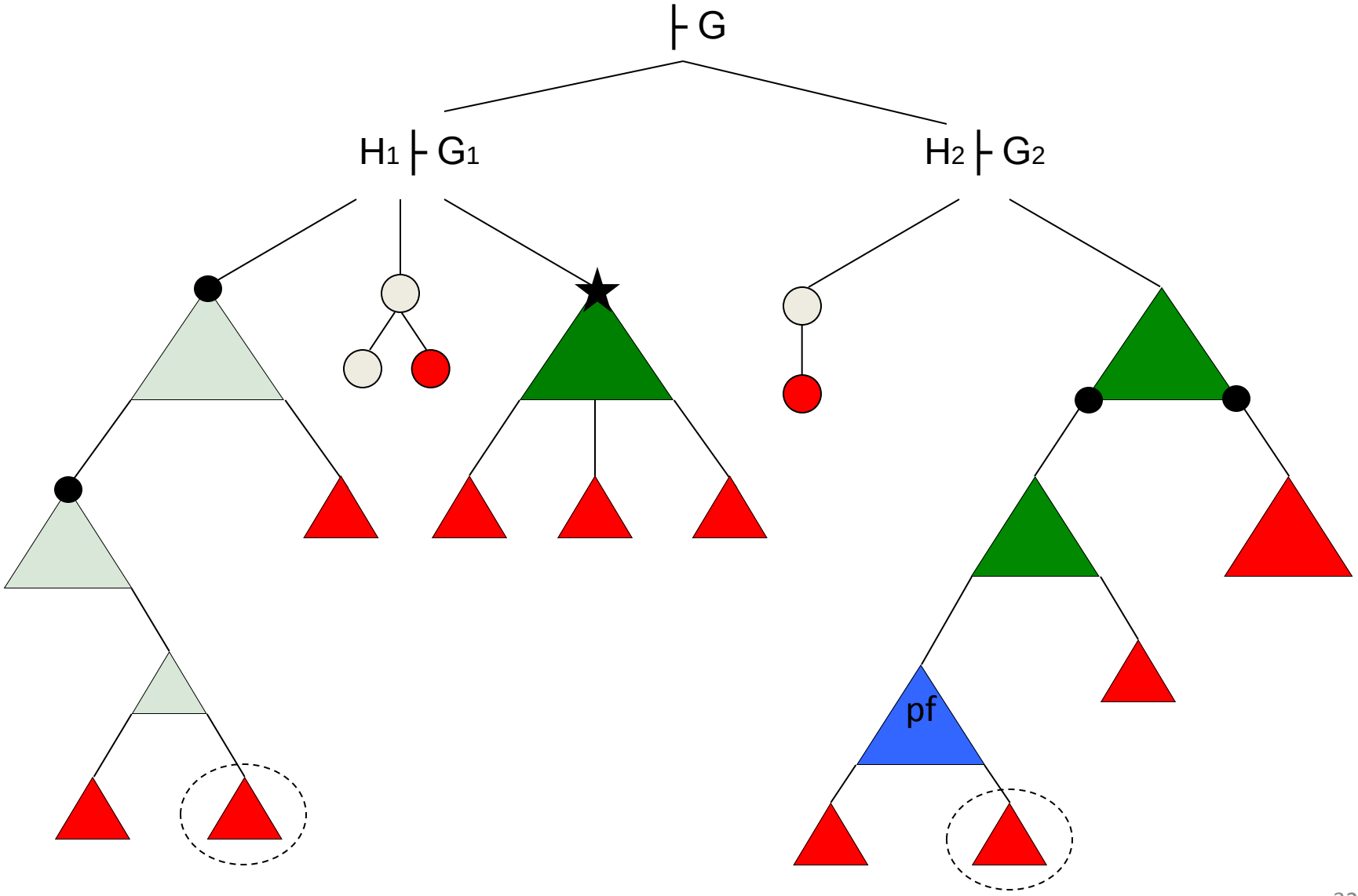
This program is close to a declarative mathematical description of roots.

Closer Look at a Proof

We see a refinement style proof here. The proof starts with the goal and works downward to generate sub goals by selecting an applicable rule.

The PRL Project, started with Joseph Bates and me in 1979 studies refinement proofs, the Program/Proof Refinement Logic (PRL).

A Picture of Proof Structure



A Big Bang: Creating the Formal Universe

We have just seen the epitome of a formal proof, and idea introduced by Hilbert progressively from 1904, 1925, 1934, then realized by computer scientists.

Formal axiomatic systems were the key to showing the consistency of mathematics according to Hilbert as long as one used what he called finitary mathematics. To study these systems and show they are consistent, cannot prove False. Primitive recursion is finitary, and PRA is a finitary logic, not very strong.

Integer Root - Proof

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

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

.....basecase.....
$\vdash \exists r:\mathbb{N}.\ \ r^2 \le 0 < (r+1)^2$
$\surd$ BY existsR ⌜0⌝ THEN Auto

.....upcase.....
$i:\mathbb{N}^+,\ \ r:\mathbb{N},\ \ r^2 \le i-1 < (r+1)^2$
$\vdash \exists r:\mathbb{N}.\ \ r^2 \le i < (r+1)^2$
BY Decide ⌜$(\underline{r}+1)^2 \le i$⌝ THEN Auto

.....Case 1.....
$i:\mathbb{N}^+,\ \ r:\mathbb{N},\ \ r^2 \le i-1 < (r+1)^2,\ \ (r+1)^2 \le i$
$\vdash \exists r:\mathbb{N}.\ \ r^2 \le i < (r+1)^2$
$\surd$ BY existsR ⌜$\underline{r}+1$⌝ THEN Auto'

.....Case 2.....
$i:\mathbb{N}^+,\ \ r:\mathbb{N},\ \ r^2 \le i-1 < (r+1)^2,\ \ \neg((r+1)^2 \le i)$
$\vdash \exists r:\mathbb{N}.\ \ r^2 \le i < (r+1)^2$
$\surd$ BY existsR ⌜$\underline{r}$⌝ THEN Auto

From the proof we can extract a program to compute the root

Since the proof is constructive, it defines a function from natural numbers N to N. We can see the ML code for the function.

The constructive provers can extract computable functions from such proofs, Coq gives Haskell and ML. Nuprl gives ML like programs.

Integer Root - Program

Algorithm with proof terms

```
let rec sqrt i
= if i=0 then <0,pf_0>
  else let <r,pf_{i-1}>  = sqrt (i-1)
       in
         if (r+1)^2≤n  then
<r+1,pf_i>
         else <r,pf_i' >
```

Algorithm with proof terms removed

```
let rec sqrt i
= if i=0 then 0
  else let r = sqrt (i-1)
       in
         if (r+1)^2≤n  then r+1
         else r
```

SML Program

```
fun sqrt n = if n=0 then 0
             else let val r = sqrt (n-1)
                  in
                    if n< (r+1)^2  then r
                    else r+1
                  end
```

How does constructivity guarantee a computable function?

What does constructivity mean? It might seem like a very long way from denying the law of excluded middle to extracting functions.

Denying the validity of P v ¬P is a consequence of constructive semantics, not a definition of intuitionism or constructivity.

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 $\mathscr{M}$.

[A & B**]** == **[**A**]** x **[**B**]**

[A v B**]** == **[**A**]** + **[**B**]**

[A => B**]** == **[**A**]** → **[**B**]**

[false**]** == ϕ (the empty type)

[¬A**]** == **[**A**]** → ϕ

Evidence for Quantified Propositions

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

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

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

L.E.J. Brouwer's Intuitionism

This semantics of evidence comes from Brouwer in 1907. He saw this semantics as a natural way to understand the meaning of all mathematical statements. He believed that mathematical meaning for the human mind was grounded in computational intuition. He believed this intuition can not be captured in language or in logics.

As a semantic method applied in logic by Heyting and Kolmogorov this is called the Brouwer/Heyting/Kolmogorov (BHK) semantics.

Philosophical Observations: The Formal Universe

Brouwer and Hilbert came well before Turing, and they could not see clearly the extent of their deep insights. They fought each other bitterly and did not realize that the union of their insights had created a new digital universe.

Hilbert -- take out all intuition, formalize

Brouwer – intuition is the foundation

Heyting and Kolmogorov did not realize that they had formulated the logical laws of this universe that Brouwer discovered. These laws are deep.

From the Profound to the Practical

- Constructive proofs give the user more bang for the effort, get a proof and a program known to be correct-by-construction. In a sense the proof is a program as well.
- Can this method generate more interesting programs with reasonable effort?
- Can we get efficient programs this way?
- Does this work for all data types, A -> B?

Extraction Comes from the Semantics

We see from the evidence semantics that the meaning of a true assertion provides the function we need. Systems like Nuprl and MetaPRL rely on a sound implementation so that provable statements have evidence. This evidence is sometimes called a realizer, a term introduced by Kleene when he related Brouwer's ideas to those of Church/Turing.

A more subtle example

We will consider another example that makes a deep point discovered by Cornell PhD, Douglas Howe. He showed how to draw on the power of the Church/Turning complete computation system of Nuprl.

These results make Nuprl and MetaPRL practical programming languages.

A Theorem that Roots Exist
(Can be Constructed)

Theorem $\forall n: \mathbb{N}.\ \exists r: \mathbb{N}.$ Root (r,n)

Pf by efficient induction

Base n = 0 let r = 0

Induction case assume $\exists r: \mathbb{N}.$Root (r, n/4)

Choose r_0 **where** $r_0^2 \le n/4 < (r_0 + 1)^2$

note $4 \cdot r_0^2 \le n < 4 \cdot (r_0 + 1)^2 = 4 \cdot r_0^2 + 8 \cdot r_0 + 4$

thus $2 \cdot r_0 \le$ root (n) $< 2 \cdot (r_0 + 1)$

if $(2 \cdot r_0 + 1)^2 \le n$ **then** r = 2×r + 1

since $(2 \cdot r_0)^2 = 4 \cdot r_0^2 + 8 \cdot r_0 + 4$

else r = 2×r since $(2 \cdot r_0)^2 \le n < (2 \cdot r_0 + 1)^2$

Qed

Efficient Root Program

The iterative code and the recursive program are both very inefficient. It is easy to make them faster. Note the recursive call root(n/4).

$$
\begin{aligned}
&\text{root}(n) := \ \textbf{if}\ \ n=0\ \ \textbf{then}\ \ 0\\
&\textbf{else}\ \ \textbf{let}\ \ r_0 = \text{root}\,(n/4)\ \textbf{in}\\
&\textbf{if}\ (2\cdot r_0+1)^2 \le n\\
&\qquad\qquad \textbf{then}\ 2\cdot r_0+1\\
&\qquad\qquad \textbf{else}\ 2\cdot r_0\ \textbf{fi}\\
&\qquad\qquad \textbf{fi}\\
&\qquad\qquad \text{since if}\ \ n \neq 0,\ n/4 < n
\end{aligned}
$$

This is an efficient recursive function, but why is it correct?

Fast Integer Root

```
∀n:ℕ.  ∃r:ℕ.  r²≤n<(r+1)²
BY  allR  THEN  Assert ⌜∀j:ℕ.  (n-j)²≤n ⇒ ∃r≥n-j. r²≤n<(r+1)²⌝
.....Assertion.....
   n:ℕ,  j:ℕ,  (n-j)²≤n
   ⊢ ∃r≥n-j. r²≤n<(r+1)²
   BY  NatInd 2
   .....basecase.....
      n:ℕ,  (n-0)²≤n
      ⊢ ∃r≥n-0. r²≤n<(r+1)²
   √ BY  existsR ⌜n⌝ THEN Auto'
   .....upcase.....
      n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n
      ⊢ ∃r≥n-j. r²≤n<(r+1)²
      BY  Decide ⌜n<(n-j+1)²⌝ THEN Auto
      .....Case 1.....
         n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n,
         n<(n-j+1)²
         ⊢ ∃r≥n-j. r²≤n<(r+1)²
      √ BY  existsR ⌜n-j⌝ THEN Auto'
      .....Case 2.....
         n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n
         ¬(n<(n-j+1)²)
         ⊢ ∃r≥n-j. r²≤n<(r+1)²
         BY  impL 3 THEN Auto
           n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n

           ¬(n<(n-j+1)²)
           ⊢ ∃r≥n-j. r²≤n<(r+1)²
        √  BY  existsR ⌜r⌝ THEN Auto'
.....Main.....
   n:ℕ,  ∀j:ℕ.  (n-j)²≤n ⇒ ∃r≥n-j. r²≤n<(r+1)²
   ⊢ ∃r:ℕ. r²≤n<(r+1)²
   BY  allL 2 ⌜n⌝ THEN Auto
      n:ℕ,  r:ℕ,  r≥n-n,  r²≤n<(r+1)²
      ⊢ ∃r:ℕ. r²≤n<(r+1)²
√     BY  existsR ⌜r⌝ THEN Auto
```

Fast Integer Root

$\forall n:\mathbb{N}.\ \ \exists r:\mathbb{N}.\ \ r^2 \le n < (r+1)^2$
BY allR THEN Assert ⌜$\forall j:\mathbb{N}.\ \ (n-j)^2 \le n \Rightarrow \exists r \ge n-j.\ r^2 \le n < (r+1)^2$⌝
.....Assertion.....
$n:\mathbb{N},\ \ j:\mathbb{N},\ \ (n-j)^2 \le n$
$\vdash \exists r \ge n-j.\ r^2 \le n < (r+1)^2$
BY NatInd 2
.....basecase.....
$n:\mathbb{N},\ \ (n-0)^2 \le n$
$\vdash \exists r \ge n-0.\ r^2 \le n < (r+1)^2$
√ BY existsR ⌜n⌝ THEN Auto'
.....upcase.....
$n:\mathbb{N},\ \ j:\mathbb{N}^+,\ \ (n-(j-1))^2 \le n \Rightarrow \exists r \ge n-(j-1).\ r^2 \le n < (r+1)^2,\ \ (n-j)^2 \le n$
$\vdash \exists r \ge n-j.\ r^2 \le n < (r+1)^2$
BY Decide ⌜$n < (n-j+1)^2$⌝ THEN Auto

Fast Integer Root

```
      .....Case 1.....
         n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1).r²≤n<(r+1)²,  (n-j)²≤n,
         n<(n-j+1)²
         ⊢ ∃r≥n-j.r²≤n<(r+1)²
      √ BY  existsR ⌜n-j⌝ THEN Auto'
      .....Case 2.....
         n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1).r²≤n<(r+1)²,  (n-j)²≤n
         ¬(n<(n-j+1)²)
         ⊢ ∃r≥n-j.r²≤n<(r+1)²
         BY  impL 3 THEN Auto
           n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1).r²≤n<(r+1)²,  (n-j)²≤n

           ¬(n<(n-j+1)²)
           ⊢ ∃r≥n-j.r²≤n<(r+1)²
        √  BY  existsR ⌜r⌝ THEN Auto'
.....Main.....
   n:ℕ,  ∀j:ℕ.  (n-j)²≤n ⇒ ∃r≥n-j.r²≤n<(r+1)²
   ⊢ ∃r:ℕ.r²≤n<(r+1)²
   BY  allL 2 ⌜n⌝ THEN Auto
      n:ℕ,  r:ℕ,  r≥n-n,  r²≤n<(r+1)²
      ⊢ ∃r:ℕ.r²≤n<(r+1)²
√     BY  existsR ⌜r⌝ THEN Auto
```

Fast Integer Root

```
.....Case 1.....
   n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n,
   n<(n-j+1)²
   ⊢ ∃r≥n-j. r²≤n<(r+1)²
√ BY  existsR ⌜n-j⌝ THEN Auto'
.....Case 2.....
   n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n
   ¬(n<(n-j+1)²)
   ⊢ ∃r≥n-j. r²≤n<(r+1)²
   BY  impL 3 THEN Auto
     n:ℕ,  j:ℕ+,  (n-(j-1))²≤n ⇒ ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n

     ¬(n<(n-j+1)²)
     ⊢ ∃r≥n-j. r²≤n<(r+1)²
 √  BY  existsR ⌜r⌝ THEN Auto'
```

How did we manage the fast program?

The method was to use efficient induction. How do we get the efficient induction proof rule? How do we know we can find the fast algorithms?

Essentially MetaPRL and Nuprl start with all possible programs, e.g. a Church/Turing complete language, an applied lambda calculus.

Correctness of the Recursive Program

The proof uses this "efficient induction principle." We can give a simple proof of the principle by ordinary induction.

$$P(0) \;\&\; \forall n{:}\,\mathbb{N}.(P(n/4) \Rightarrow P(n)) \Rightarrow \;\; \forall n{:}\,\mathbb{N}.P(n)$$

How do we derive efficient induction?

There are two ways to do this in computational type theory. Both rely on the key idea that we must prove Frege's well formedness judgments rather than simply type check them. We can use regular induction to prove the type of efficient induction.

The other way is to define efficient induction or any other fast algorithm using the **Y** combinator, a fixed point combinator.

Using Y seems impossible!

Type systems are known to give only subrecursive languages, can't get Turing completeness as would follow from the full untyped lambda calculus!

This is true about the Coq theory, CIC, and system, but not true about Computational Type Theory (CTT) and Nuprl/MetaPRL.

What's up, how does that work?

Howe's Trick

The other methods uses Doug Howe's discovery of the exact right computational equality for lazy evaluation. We can express this equality in computational type theory and use it to prove that untyped λ-terms are computationally equal to typed terms.

This kind of equality is essential in Nuprl and Coq.

Other Interesting Types

The Nuprl system has remained open, allowing new types and type constructors. Many of these have served us very well.

I will mention how we use intersection types since they came to us via CMU as did other features of Nuprl and MetaPRL as systems, e.g. the LF idea, defunctionalization in our evaluators, and domain theory (via partial types).

Intersection and Top Types

We can build records using a binary intersection of types, $A \bigcap B$

These are the elements in both types A and B with x=y in the intersection iff x=y in A & x=y in B.

Top is the type of all closed terms with the trivial equality, x=y for all x, y in Top. Note for any type A, we have $A \sqsubseteq Top$ and $A \bigcap Top = A$.

Building Records by Intersection

Record types can be built by intersecting singleton records as follows. Let

Id = {x,y,z,…} and Sig: Id -> Type where

Sig(i)= Top as the default. Then

$$x{:}A \sqcap y{:}B = \begin{cases} \{x{:}A\ ;\ y{:}B\} & \text{if } x \neq y \\ \{x{:}A \sqcap B\} & \text{if } x = y. \end{cases}$$

Axiomatizing Co-Inductive Types

In 1988 before we added intersection types to CTT, we axiomatized co-inductive types and implemented them in Nuprl as primitive.

Now with intersection types and the Top type, we can define them and introduce variants.

Defining Co-Recursive Types in CTT

Let F be a function from types to types such as F(T) = **N** x T or F(T) = St -> In -> St x T. Define objects of the co-recursive type corec(T. F(T)) as the intersection of the iterates of F applied to Top.

$$\bigcap_{n:\mathrm{N}} F^n(Top)$$

To build elements, we take the fixed point of a function f in the following type.

$$\bigcap_{T:Type} T \to F(T)$$

Elements of Co-Inductive Types

For example to build elements of the co-recursive type for the function F(T) given by

$$St \rightarrow In \rightarrow St \times T$$

we use fix(λ(t.λ(s,i.<update(s,i),t>))).

It is easy to show by induction that this belongs to the co-recursive type. If the function F is continuous, the type is a fixed point of F, $F(corec(T.F(T))) \Xi corec(T.F(T))$.

Fundamental Unity Emerges

Type theory plays a unifying foundational role in computer science comparable to the role of set theory in mathematics. Just as set theory unified many basic concepts, type theory unifies other concepts in ways that set theory cannot because it does not integrate a rich computation system beyond what comes from intuitionistic first-order logic.

Summary of Main Points

1. Expressive (rec-types, types as objects, quotients, squashing (hiding information), subtyping, intersection, Top, comp equal)
2. Constructive semantics (props-as-types)
3. Grounded in Church/Turing complete programming language
4. Computational equality on terms is key
5. Open-ended system including computational base (Church's thesis not needed)
6. Grounded in primitive proofs (LCF tactics)

THE END

A Motivating Observation

The demonstration that we can usefully implement pure and applied mathematical reasoning is one of the major intellectual discoveries of computer science, a complement to the discovery that automating mundane mental tasks is very difficult.

Here is a short list of some of the many impressive examples of this very demanding work. There are many other worthy examples that I won't list in this brief addendum to the lecture.

Selected Notable Examples of Formalized Mathematics

- Four Color Theorem formalization – Gonthier
- Kepler Conjecture Work – Halles and HOL team
- Constructive Higman's Lemma – Murthy
- Prime Number Theorem – Harrison, Avigad
- Girard's Paradox -- Howe
- POPLMark Challenge – Coq,Twelf,HOL
- Paris driverless Metro line 14 – Abrial, B-tool
- Mizar's Journal of Formalized Mathematics

A short story about our recent work on protocol synthesis

We have been demonstrating how to synthesize correct by construction distributed protocols. One of the properties we must prove first is that the protocols do not block. When we do this in Nuprl we show effectively that they are non blocking. This led me to the following cute result in 2008.

Fault-Tolerant Consensus Protocols

Cloud computing and distributed file systems used by Google, IBM, Amazon, etc. achieve reliability by duplication and depend on consensus protocols that are fault-tolerant and almost never fail to keep the replicas the same. They use Leslie Lamport's Paxos protocol.

Computing theory says that fault tolerant consensus is unachievable, this is the famous FLP result of Fischer, Lynch, and Paterson from 1985.

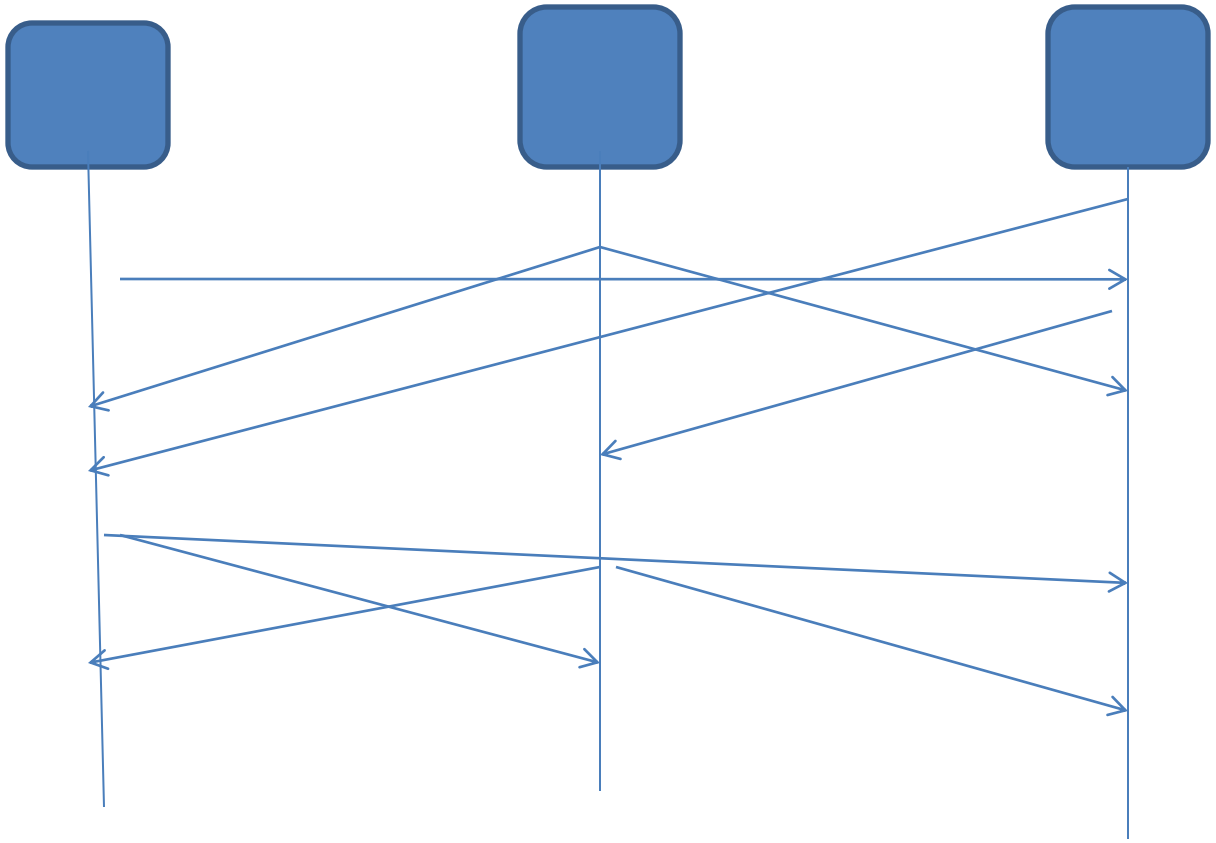
Consensus is a Good Example

In modern distributed systems, e.g. the Google file system, clouds, etc., reliability against faults (crashes, attacks) is achieved by replication.

Consensus is used to coordinate write actions to keep the replicas identical. It is a critical protocol in modern systems used by IBM, Google, Microsoft, Amazon, EMC, etc.

Requirements of Consensus Task

Use asynchronous message passing to decide on a value.



Logical Properties of Consensus

P1: If all inputs are unanimous with value v, then any decision must have value v.

All v:T. (If All e:E(Input). Input(e) = v then
All e:E(Decide). Decide(e) = v)

Input and Decide are event classes that effectively partition the events and assign values to them. The events are points in abstract space/time at which "information flows." More about this just below.

A Fundamental Theorem of about the Environment

The Fischer/Lynch/Paterson theorem (FLP85) about the computing environment says:

it is not possible to guarantee consensus among n processes when one of them might fail.

We have seen the possibility of this with the 2/3 Protocol which could waffle between choosing 0 or 1. The environment can act as an adversary to consensus by managing message delivery.

The Environment as Adversary

In the setting of synthesizing protocols, I have shown that the FLP result can be made constructive (CFLP). This means that there is an algorithm, env, which given a potential consensus protocol P and a proof pf that it is nonblocking can create message ordering and a computation based on it, env(P,pf), in which P runs forever, failing to achieve consensus.

Perfect Attacker

The algorithm env(P,pf) is the perfect "denial of service attacker" against any consensus protocol P that is sensible (won't block).

Note, 2/3 will block if it waits for n replies or if it refuses to change votes as rounds progress.

THE END AGAIN