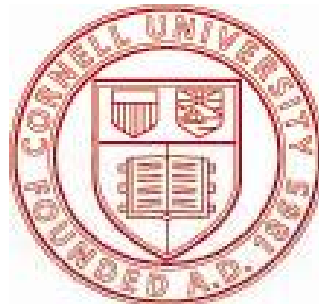
Series Title: Proofs as Processes

Robert L. Constable

Cornell University
Department of Computer
Science



What is this lecture series about?

1. Present two new results:

   -- a bit of new theory: completeness of iFOL,

   -- a bit of new theory and practice: synthesized protocols, e.g. “proofs as processes” and the logic of events.

2. Look toward what might come next in type theory.

Reflecting a bit on History

We will briefly note some related history as befits the international Turing celebrations.

For example, I'll mention the role of Turing and Church in type theory. For the constructive aspects, we need to mention Brouwer's work.

Here we see three major figures, one from computer science, one from logic, and one from mathematics creating the genome of constructive type theory.

Amazing Times

Consider the broad international uptake of Coq!

Four Color Theorem of Gonthier

MSR investing in Gonthier’s unit proving Feit-Thompson

Fundamental Theorem of Algebra

The CompCert CLight compiler of Leroy

Fields medalist Voevodsky’s Homotopy Theory

Software Foundations textbook by Pierce *et al*

DARPA heavily using proof assistants, mainly Coq

Several key industrial applications (Java card, Air Bus,…)

Many thousands of Coq users – over 10K used it?

And more …

Imagine what Coq programmers know

Expert programmers dig deeply into their languages. Imagine what expert Coq programmers learn, the kind of logic and mathematics that is part of what they routinely use.

mathematical types, including inductive types

formal logic with higher-order quantifiers

proof rules and their realizers (extracts)

formal semantics

classical versus constructive logic

all in a unified theory

Imagine what programmers learn

In the United States, the Common Core Mathematics curriculum teaches algorithms, before middle school.

School students now learn sets in mathematics and types and computable functions in programming.

What will happen when they learn Coq-like programming in school as well, as they might?

Unified Theory Idea is Plausible

Coq, Nuprl, and MetaPRL are closely related siblings, they share the theory of inductive types based on Nax Mendler's LICS papers, also for co-inductive types.

They share the ITT predicative universe hierarchy and realizability semantics.

They share the LCF tactic mechanism from 1979.

They even share code written by Chetan Murthy.

Unified Computing Theory Idea

The constructive type theories of Coq and Nuprl taken together span a large part of computing theory with Coq especially strong in programming languages (PL) and advanced mathematics and Nuprl strong in distributed protocols and constructive domain theory (partial types).

We'll see some of the protocol work in the "third half of the lecture series" as they say on Car Talk here in America, and I'll do one example on partial types.

What Turing foresaw

In the April issue of *Science*, Andrew Hodges said this about Turing:

His universal machine would compute with formulas as well as numbers.

Hodges says: ``It (the universal machine) put logic, not arithmetic, in the driving seat.''

What else Turing foresaw

Turing and his PhD student Robin Gandy worked on type theory, and Turing wrote

Practical Forms of Type Theory, JSL, 1948.

His idea was that type theory was the natural language for “working mathematics” and that Church’s simple type theory was a good model – HOL users agree.

What OPLSS contributes

Constructive type theories and their proof assistants are the joint creation of computer science (informatics), logic, and mathematics – ITT, CTT, CIC are examples.

OPLSS is one of the schools that brings the players from these disciplines together, nurtures work in type theory – its basic concepts and important applications.

I believe that the PL community will continue to play a nurturing role and understands that the subject is far from finished and that it plays a unifying role in computing theory as well as in programming languages.

What Remains to be Done?

Consider one of the most basic concepts in constructive type theory -- realizability semantics for logic -- under a variety of different names:

propositions as types, proofs as terms (PAT)

proofs-as-programs (programs-from-proofs)

evidence semantics

Curry-Howard isomorphism

Brouwer/Heyting/Kolmogorov (BHK) semantics

A Poll

I'd like to think that propositions-as-types realizability is one of those ideas familiar to many OPLSS participants and that it will eventually be known by all computer scientists – the way sets, open sets, continuous functions, vector spaces, etc. are known to all mathematicians.

How many people here could teach this idea to a computer science 2nd year student?

Brief Review: Propositional Evidence

Suppose that we have evidence types for the atomic propositions, A,B,C, …. Here is how evidence is defined for compound propositions.

[A & B] == [A] × [B]	Cartesian product
[A v B] == [A] + [B]	disjoint union
[A => B] == [A] → [B]	function space
[¬A] == [A] → void	functions to empty type

Brief Review: Propositional Evidence

Suppose that we have evidence types for the atomic propositions **A,B,C**,... Here is how to define evidence for compound formulas using types (or sets).

$$[\![\boldsymbol{A} \,\&\, \boldsymbol{B}]\!] == [\![\boldsymbol{A}]\!] \,\mathrm{x}\, [\![\boldsymbol{B}]\!]$$

$$[\![\boldsymbol{A} \vee \boldsymbol{B}]\!] == [\![\boldsymbol{A}]\!] \oplus [\![\boldsymbol{B}]\!]$$

$$[\![\boldsymbol{A} \Rightarrow \boldsymbol{B}]\!] == [\![\boldsymbol{A}]\!] \rightarrow [\![\boldsymbol{B}]\!]$$

$$[\![\neg \boldsymbol{A}]\!] == [\![\boldsymbol{A}]\!] \rightarrow \phi$$

Evidence for Quantified Formulas

[Ex.B(x)]   == x:D × [B(x)] dependent product

[Allx.B(x)] == x:D → [B(x)] dependent functions

The existential quantifier is also regarded as a ∑ type, a disjoint sum of a family of evidence types.

Evidence for Quantified Statements

$$[\![\exists \boldsymbol{x} : \boldsymbol{A}.\boldsymbol{B}_{\boldsymbol{x}}]\!] == \boldsymbol{x} : [\![\boldsymbol{A}]\!] \times [\![\boldsymbol{B}_{\boldsymbol{x}}]\!]$$

$$[\![\forall \boldsymbol{x} : \boldsymbol{A}.\boldsymbol{B}_{\boldsymbol{x}}]\!] == \boldsymbol{x} : [\![\boldsymbol{A}]\!] \rightarrow [\![\boldsymbol{B}_{\boldsymbol{x}}]\!]$$

The existential quantifier is also regarded as a ∑ type, a disjoint sum of a family of evidence types.

Simple Example

Consider the meaning of this proposition in the realizability semantics:

$$(A\&B \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))$$

Its realizer in CTT is this untyped applied λ-term

$\lambda f. \lambda x. \lambda y. f(\langle x,y \rangle)$ or

$\lambda f,x,y. f(\langle x,y \rangle)$

Simple Example

Consider the meaning of this proposition in the realizability semantics:

$$(A \& B \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))$$

Here is the typing judgment *a la* CTT

$$f: (A \& B) \Rightarrow C,\ x{:}A,\ y{:}B \mid\!- f(\langle x,y \rangle)\ \varepsilon\ C$$

This is the specification for the Currying task.

Un-Currying Task

$(A \Rightarrow (B \Rightarrow C)) \Rightarrow (A\&B \Rightarrow C)$

the untyped realizer is

$\lambda f. \lambda p. f(p._1)(p._2)$

$p._1$ and $p._2$ project from the pair,

e.g. $p._1 \, \varepsilon \, A$ and $p._2 \, \varepsilon \, B$

the typing judgment is

$f: (A \Rightarrow (B \Rightarrow C)), p: A\&B \mid\!- f(p._1)(p._2) \, \varepsilon \, C$

Here is a challenge

Our colleagues finally understand that constructive logic is natural and easy to understand, directly tied to specifying programming problems and all that.

They learn some logic, get excited and want to know the consistency and completeness theorems for this idea in pure first-order logic – FOL.

We give them proof rules and consistency, then …

We give them back Kripke semantics -- and possibly a classical completeness proof as well!! And then we give them excuses.

Lecture Plan

1. Look at a constructive completeness proof for intuitionistic first-order logic (iFOL) with respect to its intended realizability semantics.

2. Inject discussion of type theory design issues and possible future directions, and follow up in subsequent lectures as appropriate and suited your interests.

Narrative about Completeness

First-Order Logic (FOL) is a central notion in logic (math, philosophy, computer science, linguistics). It's taught in basically all logic books.

Gödel's completeness theorem ties together proof and validity -- central to classical FOL study liking proof theory and model theory.

There was no constructive theorem of this kind for intuitionistic first-order logic (iFOL) using the intended realizability semantics (BHK). Completeness only for Beth and Kripke semantics.

Gödel's PhD Thesis was Completeness of FOL

On July 6, 1929 Gödel's PhD dissertation was approved by his advisor Hans Hahn. It was titled: *On the completeness of the calculus of logic.*

In 1930 it was published as: *The completeness of the axioms of the functional calculus of logic.* He used the axioms of *Principia Mathematica.*

Another Poll

How many of you have seen a proof of this theorem?

How many of you could teach the proof?

I think the simplest and clearest proof is from Smullyan's *First-Order Logic*, Springer 1968, Dover 1993, using tableaux, pp 57-61.

Gödel’s Completeness Theorem

This theorem says that there is a set of inference rules and axioms such that every formula of FOL that is true in every model (valid) is provable using these rules.

These are the complete set of rules.

This theorem sets us up for Gödel’s more famous incompleteness theorem for arithmetic.

Classical Models and Truth

There are many ways to prove this theorem, we don't teach Gödel's original proof since there are now much better ones. Also we now sometimes refer the models as Tarksi structures. These structures have a domain of discourse D, assumed to be non-empty, and for every relation R^n of the logic, we assign a propositional function R^n: D → Bool. So we get $\langle D, R_1, R_2, \ldots, R_m \rangle$ as a structure or model.

Each R_i has an arity n_i, $R_i: D^{ni} \rightarrow$ Bool.

Classical Method of Proof

The usual classical proof assumes that a formula G is valid (true in all structures), and then a systematic method of searching for a proof is devised based on the proof rules.

We then show that if the systematic procedure fails to find a proof, it will generate a structure M in which the formula is not true, contradicting its validity. Running on forever creates such a structure as well by König's Lemma.

Need for a new proof method

This classical proof method for FOL fails for iFOL, and we need to change the formulation.

First, the domain D must be a type. Second, the relations are not maps into Bool, they must be maps into Prop, the propositions on D. Third, the semantics is not based on Tarski but on Brouwer, on realizability. Fourth, we must be able to find the proof, not show that the proof finding procedure must fail, we must know that it will halt.

Narrative Theme continued

The intended semantics is closely tied to proofs, the computational content of proofs provides the realizers.

Results by Gödel, Kreisel, etc. since Beth 1947 suggest that there is no constructively valid completeness theorem with respect to BHK semantics and intuitionistic validity. I will mention these results next.

Narrative Theme continued

The negative results are for intuitionistic validity, say as given in Troelstra and van Dalen (TvD88). There is a vagueness and lack of “canonicality” there, but some results:

Kreisel based on Gödel showed that given Church’s Thesis (CT) the valid formulas of iFOL are not r.e. Moreover, completeness implies Markov’s Principle.

Intuitionistic Validity

The validity semantics carries over the classical notion of a model and uses BHK semantics for the logical operators on propositions and types for the domain of discourse, e.g. D is a type in the first universe, $Type_1$.

We say that formula G is valid iff it is realizable in all models of the domain and the atomic propositions. So modern constructive type theories like CIC, ITT, and CTT make this semantics precise. That is a big step since 1988.

Narrative Theme continued

CTT implemented by Nuprl also provides a closely related notion, uniform validity, by using intersection types and polymorphic terms.

That concept is not in CIC nor ITT. (It could easily be added to ITT82.)

Narrative Theme continued

The punch line is that uniform validity provides a strong completeness result, and the right one, since it captures precisely the observation that iFOL proofs provide uniform realizers, also called polymorphic realizers, as their computational content.

Here is how I plan to provide the details for this narrative.

Proof Outline Plan

1. Mention briefly Constructive Type Theory (CTT02), concepts since 2002 needed for the completeness theorem.

2. Sketch a formal semantics for intuitionistic first-order logic (iFOL) and minimal first-order logic (mFOL).

3. State and illustrate the new completeness theorem, proved with Mark Bickford in 2011.

Some Unifying Basic Concepts from Type Theory

Computability, e.g. partial computable functions in all types (Turing complete)

Polymorphism and intersection types

Uniform validity and constructive completeness

Realizers for Inference Rules

Rules for Implication

Implication Introduction (=>R)

ū:H |- A => B by λ(x.slot)

ū:H, x:A |- B by b(x) -----------^

The realizer or proof term is λū,x.b(x)

Sample Refinement Proof

|- A=>(B=>A) by λ(x.__)

x:A |- (B=>A) by λ(y.__) ---^

x:A, y:B |- A by x -------^   (axiom)

The untyped realizer is: λx. λy. x. Notice that this is polymorphic, i.e. uniform in A, B.

Grounded in computation

Constructive type theory derives its meaning from an underlying computation system, usually defined by reduction rules in the style of Plotkin's structured operational semantics.

For example, untyped lambda terms, λ(x.b), are considered irreducible values, and applications ap(f;a) are evaluated by reducing f to a function value, then applying the function value to the argument a (either by "name" or "value").

Untyped Reduction Relation

The ITT and CTT approach to computation mediates a long standing debate in computer science between those in the “no types camp” who believe that computation should be untyped, as in the untyped lambda calculus (Lisp, Scheme), and those in the “types camp” who believed it should be typed (Algol68, Java, F#).

In CTT and ITT the basic reduction relation defining computability is untyped, but reasoning is typed, this is fundamental to completeness.

Open-ended Computation System

In CTT, ITT, and CIC the computation system is left open ended, it is not possible to prove unsolvability by enumerating all possible computational forms. In particular, Church's Thesis (CT) is not postulated.

In building the Nuprl proof assistant for CTT84, we anticipated extending the computational model to concurrent systems and to physical devices. Thus CT was not the appropriate notion.

A Computational Meta-theory

Edinburgh LCF showed the power of having a computational meta-theory based on a clean functional programming model as the basis for automating reasoning. Agda, HOL, Coq, Nuprl, and MetaPRL are all descendents of the LCF proof assistant architecture.

Key to Milner's LCF is the idea that programs are polymorphic and the compiler attempts to infer the type in context given some base types such as integers and lists. The type checker was a small proof assistant.

An Aside on Proof Structure

The structure of proofs as tactic-trees adopted in Nuprl's implementation of CTT arose from a convergence of concerns:

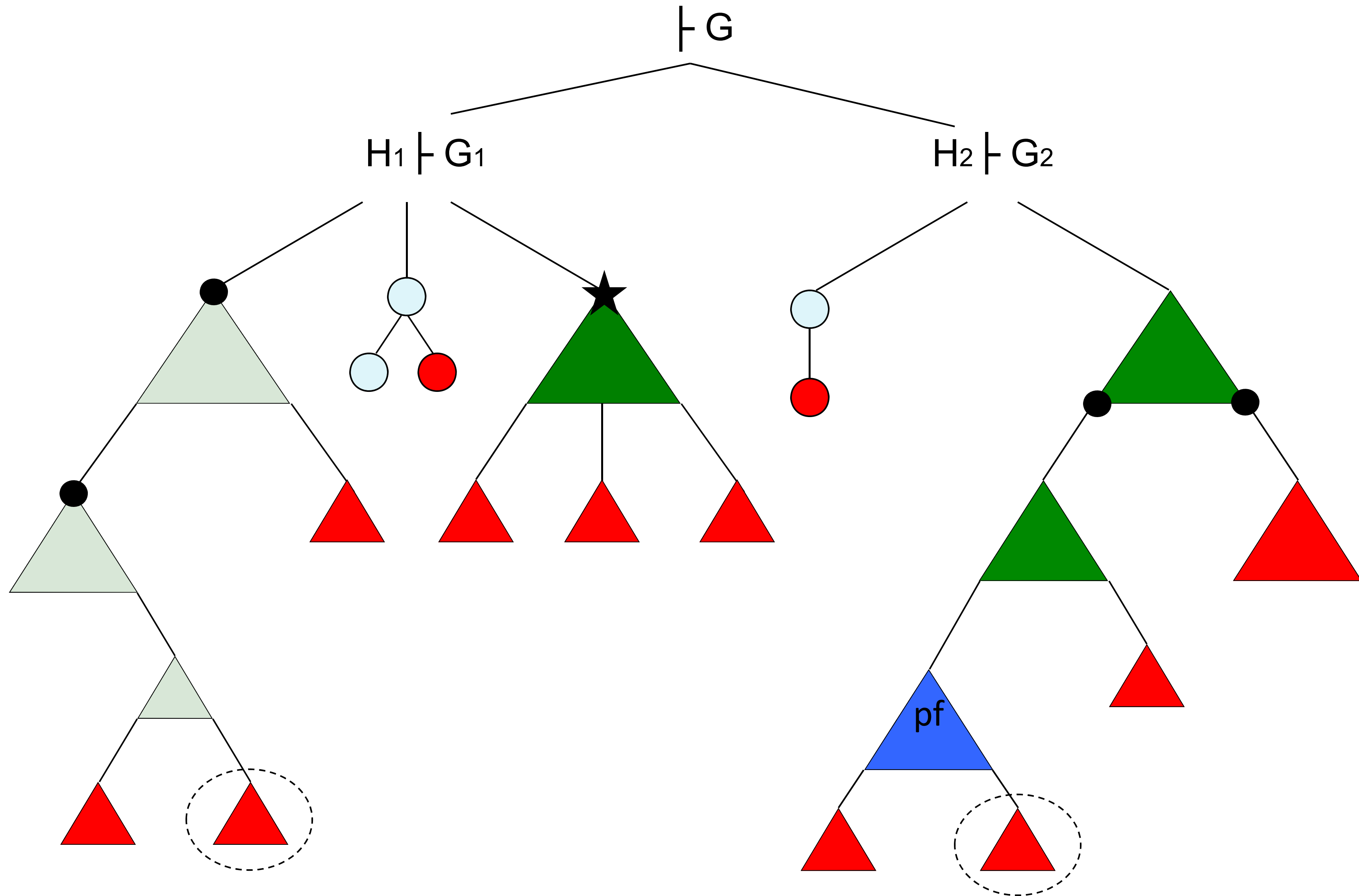
programming by refinement (PRL)

top down problem solving (AI)

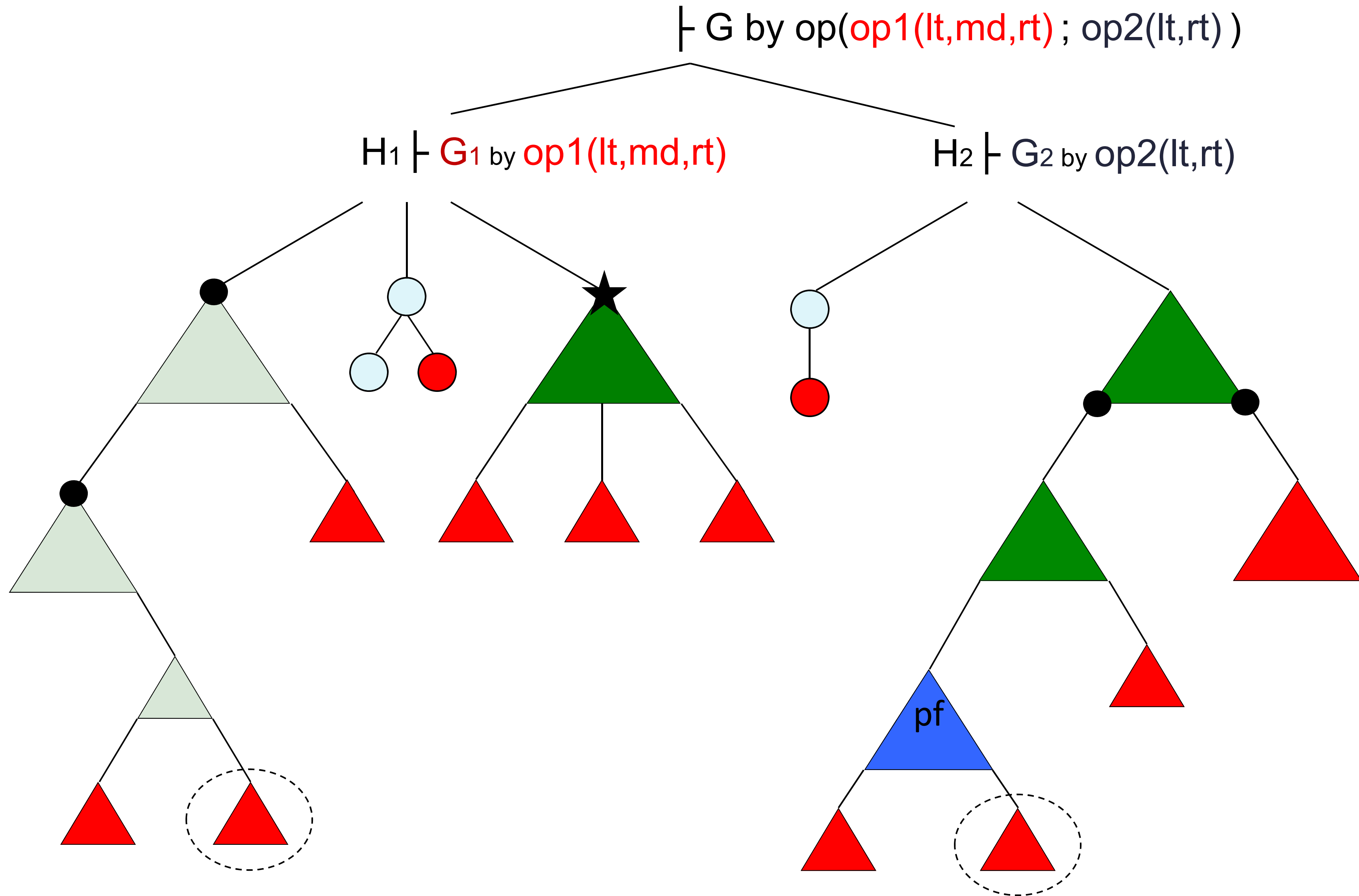
LCF tactics and tacticals

(tableaux style proofs)

A Picture of Proof Trees



Proof Structure with Extracts



Analyzing Proof Structure

key insights ●
clever step ★
filled in by machine
humans ignore
humans need
experts needed
routine
learners need
obvious
Trivial
well known
minor variant of pf

Some Unifying Basic Concepts

Polymorphism and computational semantics for records, algebraic structures, and objects.

Uniform validity and constructive completeness.

Computability of partial functions, domains.

Influence of ML style polymorphism

The power of the ML organization of tactic-based inference is well known.

I want to draw attention to the way in which polymorphism also advanced propositions-as-types as a computational semantics, illustrating the unity and synergy among the new concepts.

Type Polymorphism

The term $\lambda x.x$ is the identity function on all types, i.e. $\lambda x.x \; \varepsilon \; A \to A$ for any type A.

The term $\lambda x.\lambda y.\ pair(x;y))$ denotes a pairing function of type (A -> (B -> (A × B)) for any types A and B. We also used $\lambda x.\lambda y.\ \langle x,y \rangle$ earlier.

We say that such terms are polymorphic.

Note, the functions computed by these terms are "too big" to exist in Set Theories.

Intersection Types

The binary intersection type was used as a logical connective for propositional logic by Coppo and Dezani in 1978.

$$A \bigcap B$$

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.

Intersection Types over a Family

We can also intersect a family of types B_a. Let

$$\bigcap_{a \in A} B_a$$

denote the type of elements that belong to B_a for every a in Type. This intersection operator acts as a uniform quantifier over A.

$$\bigcap_{A:Type} A \Rightarrow A$$ contains exactly λx.x.

Aside on Dependent Records

In LICS 2003 A.Kopylov introduced dependent intersection types that allowed him to define.

Dependent Records

$\{x_1:A_1;\ \ x_2:A_1(x_1);\ x_3:A(x_1,x_2);\ \ldots\ ;x_n:A_n(x_1,\ldots,x_{n-1})\}$

functions r from Labels $\{x_1,\ldots,x_n\}$ to values

where $r(x_1)\varepsilon\ A_1$, $r(x_2)\varepsilon A_2(r(x_1))$, etc.

This mechanism is widely used in all Nuprl applications, and Kopylov PhD 2004 gave us a theory of objects.

Cumulative Impact

Our work on intersection and union types revealed the benefits of polymorphism and its explanatory power in understanding algebraic structure, classes, and objects. This led us to make these new type constructors essential to CTT after 2003.

We began to systematically exploit these ideas and apply them heavily in our practical work and in looking at logical questions. I made an observation in a 2010 class that fascinated me. Here it is.

Realizers from Proofs

We could see clearly from proof systems for iFOL, that all of the realizers are uniformly valid in the sense that they belong to the type of the axiom regardless of the domain D and the atomic propositions P_i and relations $R(x_1,\ldots,x_n)$.

We will look at this fact for refinement style presentation of the inference rules and explain the realizers built from these rules.

Refinement Style Rules

Our rules resemble tableaux proofs in that they are top down. From a proof goal, we generate subgoals.

The goals and subgoals are sequents:

$$u_1{:}A_1, \ldots, u_n{:}A_n \;|\text{-}\; G$$

more succinctly $\bar{u}{:}H \;|\text{-}\; G$

Realizers for Inference Rules

Axioms

$u_1{:}A_1, \ldots, u_n{:}A_n \mid\!- A_i$ by u_i

Generally the A_i are propositions, but we will also have declarations d:D that d is in the domain of discourse. They will justify the goal d in D.

Realizers for Inference Rules

Rules for Implication

Implication Introduction (=>R)

ū:H |- A => B by λ(x.slot)

ū:H, x:A |- B by b(x) -----------^

The realizer or proof term is λū,λx.b(x)

Sample Refinement Proof

|- A=>(B=>A) by λ(x.__)

x:A |- (B=>A) by λ(y.__) ---^

x:A, y:B |- A by x -------^ (axiom)

The realizer is: λx. λy. x. Notice that this is polymorphic, i.e. uniform in A, B.

Realizers for Inference Rules

Rules for Implication

Implication Elimination(=>L)

$\bar{u}_1$: H, x: A => B, $\bar{u}_2$:H’ |- G by ap(x; slot)

$\bar{u}_1$: H, x: A => B, $\bar{u}_2$:H’ |- A by a ---------^

$\bar{u}_1$: H, x: A => B, y:B, $\bar{u}_2$:H’ |- G by g(y) ------^

realizer $\lambda \bar{u}_1, x, \bar{u}_2 . g(ap(x;a)/y)$

Realizers for Inference Rules

Rules for &

Introduction (&R)

ū: H |-- A & B by < slot-a, slot-b>

ū: H |- A by a(ū) for slot-a --^

ū: H |- B by b(ū) for slot-b -------^

The realizer is λū. <a(ū),b(ū)> where a,b might depend on the variables in ū, say $u_1,\ldots,u_n$.

Realizers for Inference Rules

Rules for &

Elimination (&L)

$\bar{u}_1$: H, x: A & B, $\bar{u}_2$:H' |- G by spread(x; u,v. slot)

$\bar{u}_1$: H, u:A, v:B,$\bar{u}_2$:H' |- G by g(u,v) for slot ----^

realizer $\lambda\bar{u}_1$,u,v,$\bar{u}_2$.spread(x;u,v.g(u,v))

On Notation

realizer $\lambda \bar{u}_1, u, v, \bar{u}_2.spread(x; u, v.g(u,v))$

The spread notation is from ITT82 and works well as a proof term, telling us how to name the hypotheses using labels “u” and “v”.

Given a pair p, u = p.1 and v = p.2. We can also say “let p = u,v in g”, an ML like notation.

Realizers for Inference Rules

Rules for Existential Quantifier

Exists Intro (ExistsR)

ū:H |- Ex.G(x) by <d, slot>

---------------ˇ

ū:H |- G(d) by g(d) ---------^

ū:H |- d in D

realizer λū.<d,g(d)>

Sample Proof: Currying

f: (A&B) => C |- A => (B => C) by λ(x. __)

f: (A&B)=>C, x:A |- (B => C) by λ(y. __)

f: (A&B)=>C, x:A, y:B |- C by ap(f;__)

f: (A&B)=>C, x:A, y:B |- A&B by <x,y> --^

f: (A&B)=>C, x:A, y:B, v:C |- C by v = ap(f; <x,y>)

λf. λx λy. ap(f;<x,y>)

λf λx,y. f(<x,y>)

Realizers for Inference Rules

Rules for Existential Quantifier

Exists Elimination (ExistsL)

$\bar{u}_1$:H, x: Ex.R(x), $\bar{u}_2$:H' |- G by spread(x; u,v. slot)

$\bar{u}_1$:H, u:D, v:R(u),$\bar{u}_2$:H' |- G by g(u,v) ------------^

$\lambda \bar{u}_1$,u,v,$\bar{u}_2$.spread(x;u,v.g(u,v))

Realizers for Inference Rules

Rules for Universal Quantifier

Universal Intro (All-R)

ū:H |- All x.G(x) by λ(x.slot)

ū:H, x:D |- G(x) by g(x) ------^

λū λx.g(x)

Realizers for Inference Rules

Rules for Universal Quantifier

Universal Elim (All-L)

$\bar{u}_1$:H, x: All x.R(x), $\bar{u}_2$:H' |- G by ap(x;d)

$\bar{u}_1$:H, x: All x.R(x), y:R(d),$\bar{u}_2$:H' |- G by g(y)

$\bar{u}_1$:H, x: All x.R(x), $\bar{u}_2$:H' |- d in D

realizer $\lambda\bar{u}_1,x,\bar{u}_2.g(ap(x;d)/y)$, i.e. substitute ap(x;d) for y in g(y).

Realizers for Inference Rules

Rule for False

False Elim

$\bar{u}_1$: H, x: False, $\bar{u}_2$:H' |- G by any(x)

$\lambda \bar{u}_1, x, \bar{u}_2.\text{any}(x)$

This rule distinguishes iFOL from mFOL.

Realizers for Inference Rules

Classical Rule

Excluded Middle

$H \mid\!- P \vee \neg P$ by magic(P)

This realizer makes sense in a classical account of oracle computability, given an oracle for P. This realizer is not polymorphic.

Polymorphic Realizers

Notice that all of the realizers except for magic(P) are polymorphic. Thus a notion of uniform validity will rule out Excluded Middle.

More Examples

Here is another simple realizer.

(Ex.P(x) & Allx.(P(x) => Q(x))) => Ey.Q(y) by

λh.spread(h;e,f. spread(e;d,p.<d,f(d)(p)>)).

Exercise: Build the proof from this realizer.

Exercises

Find the realizers for these formulas. In the next lecture we will build a proof from them using the proof procedure for completeness.

3. (~(P & Q) & (P v ~P) & (Q v ~Q)) => ~P v ~Q.

4. Ex.(P(x) => C) => ((Allx.P(x)) => C) where C has no free occurrences of x.

My colleague, Liron Cohen, and Robbert Krebbers both students in the school will help you understand the exercises.

End of First Lecture

Lecture 2 Plan

Questions about Lecture 1?

We will look more closely at the completeness theorem, its formulation and proof. I will use the solution to exercise three as a motivator.

How many people got the solution?

If there is time, I will discuss other uses of the intersection type in CTT, the type theory of Nuprl e.g. its use in defining dependent records.

See additional exercises on the last slides.

Consistency Theorem

Notice that from our analysis of proofs and their associated extracted realizers, we can see by a simple induction on the proof structure that iFOL is consistent, every provable formula is (uniformly) valid.

We will also see that our completeness theorem provides an alternative proof of cut elimination.

Constructive Model Theory

We can see these definitions and the subsequent completeness theorems as an example of constructive model theory.

Formulating a Completeness Theorem

We have this key observation, all the realizers from the standard intuitionistic rules are polymorphic.

To get an “if and only if result”, we want to make some claim like “given a polymorphic realizer we can find a proof,”e.g. build one.

Formulating a Theorem continued

Can we make this idea precise? Does it make sense?

I knew it made sense from a result of Läuchli in 1970 where he tried to define some notion of “uniform truth” by talking about permutations. But we need a stronger notion.

Formulating a Theorem continued

We need to say, no matter what domain D we pick and no matter what atomic relations R_i, the realizer is the same.

Because of the work of Coppo and Dezani and the work of Reynolds and Pierce, we knew how to say this. We say that the evidence belongs to the intersection of all the evidence we can find for any D and R_i. This was a new idea.

Completeness Theorems

Theorem: A formula G of mFOL is provable iff it is uniformly valid.

Corollary: A formula G of iFOL is provable iff it is uniformly valid.

Constable and Bickford 2010: Intuitionistic completeness of first-order logic, arXiv 2011.

Intersection Types

The binary intersection type was used as a logical connective for propositional logic by Coppo and Dezani in 1978.

$$A \bigcap B$$

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.

Intersection Types over a Family

We can also intersect a family of types B_a. Let

$$\bigcap_{a \in A} B_a$$

denote the type of elements that belong to B_a for every a in Type. This intersection operator acts as a uniform quantifier over A.

$$\bigcap_{A:Type} A => A$$ contains exactly λx.x.

We have a conjecture

A formula G of iFOL is provable if and only if it is uniformly true with respect to any domain D and atomic relations R_i.

Let's try it only some simple examples and see if it holds up.

Try it on the exercise

What is evidence for

(~(P & Q) & (P v ~P) & (Q v ~Q)) => ~P v ~Q ?

λ h. spread(h; na,dp,dq. decide(dp;
p. ….;
np. Inl(np)))

What to do in the case p. … ? Should we split on nq?

Trying on exercise

Is there something easier than one more case split on Q? YES, how about this evidence for ~Q?

```
p.inr(λq. na(<p,q>)
```

This is almost right, but it only proves False, we need to prove ~Q. But from False anything! So we use

```
p.inr(λq. any(na(<p,q>)))
```

Can we build a proof?

We set up the problem this way, we have a formula G and a realizer evd and we want to construct a proof

$\models$ G, evd

In the case of $\models G_1 \Rightarrow G_2$, evd

we know that evd must reduce to λh.g. our evidence has this form, that is what it means in type theory for evd to be evidence. So we can start building the proof. This gives us a new evidence structure.

Evidence structures

h:(~(P & Q) & (P v ~P) & (Q v ~Q)) **|=** ~P v ~Q ,

spread(h; na,dp,dq. decide(dp;
p.inr(λq. any(na(<p,q>)));
np. Inl(np)))

Continuing

This tells us to create a new evidence structure:

na:~(P&Q), dp:(Pv~P),dq:(Qv~Q) |= ~P v ~Q, decide(dp;

p.inr(λq. any(na(<p,q>)));

np. inl(np)))

Continuing

na:~(P&Q), p:P,dq:(Qv~Q) **|=** ~P v ~Q,
inr(λq. any(na(<p,q>)))

na:~(P&Q), p:P,dq:(Qv~Q) **|=** ~Q,
λq. any(na(<p,q>))

na:~(P&Q), p:P,dq:(Qv~Q), q:Q **|=** ~Q, ap(na;<p,q>)

Looks promising so far

This example worked like a charm, but it was pretty simple. You can try it on all the other exercises. They are also easy, but I end this lecture with a harder challenge.

A Nasty Case

What about this very simple case?

|= (False => A), λf.any(f)

It also works like a charm! But so does this!

|= (False => A), λf.17

This is disconcerting because once we have False as a hypothesis, we loose all evidence.

Need Minimal Logic

If we use minimal logic, then we can avoid the case of vacuous hypotheses, but then the theorem is less interesting.

Harvey Friedman to the rescue! Harvey showed how to embed iFOL into mFOL using his famous A-translation. (Tim Griffin and Chet Murthy made this famous in the 1990's solving an open problem using Nuprl.)

Completeness for iFOL

Using Friedman's embedding of iFOL into mFOL we can get completeness for iFOL.

Given a formula G of iFOL, its embedding into mFOL, is obtained by replacing all occurrences of False by A, call this G^A.

If G is uniformly valid in iFOL, then G^A is uniformly valid in mFOL, hence provable. By replacing A by False it is provable in iFOL.

Cases for Minimal Logic

Consider how the right hand side, construction rules work in evidence structures.

$$H \models G, evd$$

G_1 & G_2 <g_1,g_2>

G_1 v G_2 inl(g_1), inr(g_2)

G_1 => G_2 $\lambda x.g_2(x)$

Allx.G $\lambda x.g(x)$

These cases are easy

$H \models G_1, g_1$

$H \models G_2, g_2$

$H \models G_i, inl(g_1)$

The complexity of the problem decreases in these cases, and we can keep decomposing the evidence term until we reach the leaves.

A hard case

H, f: A => B, H’ **|=** G, ap(f;a)

How do we justify the claim

H, f:A =>B,H’ **|=** A, a ?

Finitary models

We justify this implication elimination case by looking at a subset of the models that we call finitary. On these models for A we can ensure that if f is given an argument not in A, then it will diverge. Here we depend on having partial types, which we denote as $\bar{A}$.

We say that a belongs to $\bar{A}$ if it belongs to A if it converges.

Constant functions

It is also important in the case of functions f from A => B that we make sure we return the same result if we call the function f twice on the same argument. We do this by noting that we can always assign constant functions to these witnesses for A => B because once we have any element of B, that suffices.

Universal witnesses

In the case of evidence functions for universally quantified formulas f: Allx.G(x), we need to remember the applications of the function and store its graph on domain elements. The functions need to be equal only on identical elements, d.

The really hard case

We have only been thinking about evidence that is built from “logical bits”, but we need to imagine that evd is given to us as any term in CTT that we know is evidence. It might use number theory or graph theory to show that G is realizable. How can we cope with any possible kind of evidence?

Hard Exercise

Show that the following type theory term is evidence for
(((A&B) => (A&B)) => (C v D)) => (C => E) => E

λh,f,g. let n = gcd(10,5) in

let m = last(factorization(111)) in

let $I_1 = (\lambda x.x)^n$ in

let $I_2 = (\lambda p.<p.1,p.2>)^m$ in

(if isl(h(I_1)) then f else g)

let x = h(I_2) in if isl(x) then outl(x) else outr(x))

The intuitionistic bit

Instead of using König's Lemma, our proof uses an intuitionistic version called the Fan Theorem. Kleene showed that this theorem is not consistent with Church's Thesis.

The theorem says that in any finitely branching tree generated by a constructive process, if every branch terminates, then there is a uniform bound on the length of any path.

Main Narrative Continued

These theoretical results make constructive type theory more appealing and "complete". A practical test of such theories is that we can use them as practical programming languages that support correct-by-construction programming.

We will consider these issues in the last lecture on proofs as processes.

Partial Types

We will now examine some types used in the proof that distinguish Nuprl's CTT from Coq's CIC and reflect on the future of type theory. We also see a significant difference from the set theory approach to these ideas.

Computability in All Types

Here is how Computational Type Theory (CTT) defines recursive functions. Consider the 3x+1 function with natural number inputs.

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

Alternative Syntax

f = function(x. if x=0 then 1
else if even(x) then f(x/2)
else f(3x+1))

Using Lambda Notation

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

Here is a related term with function input f

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

The recursive function is computed using this term.

Defining Recursive Functions in CTT

```
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 = λ(x.F(f,x)) is another expression of it, and the CTT definition is:

$$fix(\lambda(f.\ \lambda(x.\ F(f,x))))$$

which reduces in one step to:

$$\lambda(x.F(fix(\lambda(f.\ \lambda(x.\ F(f,x)))),x))$$

by substituting the fix term for f in λ(x.F(f,x)) .

Non-terminating Computations

CTT defines all partial recursive functions, and partial elements of types hence non-terminating ones such as this

$$fix(\lambda(x.x))$$

which in one reduction step reduces to itself! So CTT is a Turing complete programming language.

This system of computation is a simple functional programming language. In CTT it is essentially the programming language also used in the metatheory, ML.

Partial Functions

The concept of a partial function is another example of where type theory and set theory differ substantially and where CTT differs from CIC. It is an important difference because the halting problem and related concepts are fundamentally about whether computations converge, and in type theory this is the essence of partiality. For example, we do not know that the 3x+1 function belongs to the type N -> N.

Partial Functions

We do however know that the 3x+1 function, call it f in this slide, is a partial function from numbers to numbers, thus for any n, f(n) is a number if it converges (halts).

In CTT we say that a value a belongs to the bar type $\bar{A}$ provided that it belongs to A if it converges. So f belongs to $A \to \bar{A}$ for $\bar{A}$ = N.

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

Unsolvable Problems

Suppose there is a function h that decides halting, e.g. h: $\bar{A} \rightarrow$ Bool. Define the following element of $\bar{A}$ for $\bar{a}$ in A:

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

where $\uparrow$ is a diverging element, say fix(λ(x.x)).

Note d is in by the typing rule for fix.

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

Generalized Halting Problem

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

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

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

by substituting the fix(…) for x in the body.

If hd(d) = *f*, then we see that d converges to ā, again a contradiction to the meaning of h.

Why is this result noteworthy?

First notice that the result applies to any purported halting function h. In classical mathematics, there surely is a noncomputable function to decide halting.

Moreover the standard way to present unsolvability constructively is to model Turing machines and prove that no Turing computable function can solve the halting problem. But this result says that no function can solve it.

Why is this result noteworthy?

Another reason to take note of this result is that it is so simple, about the simplest unsolvability result around -- no indexings, no reflection, simple realistic computing model.

But the main reason to take note is that this result contradicts classical mathematics and shows that CTT with bar types is not consistent with the law of excluded middle, as the rest of the theory is.

END OF LECTURE TWO

Extra Reading on Type Theory

In the next few slides I present other results in CTT type theory that are generally noteworthy for their impact on the implementation and for their wide use in Nuprl theories.

Records and structures

We will take a brief look at how we can define algebraic structures as records, illustrating other advantages of intersection types and polymorphism. This work forms the basis of an account of objects in CTT.

Subtyping and Polymorphism

There is a primitive subtyping relation in CTT.

$A \sqsubseteq B$ means that the elements of *A* are elements of *B* and a=b in *A* implies a=b in *B*. Here are some basic facts about subtyping:

$$\{x : Z \mid x > 0\} \sqsubseteq Z$$

$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A \times B \sqsubseteq A' \times B'$$

$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A + B \sqsubseteq A' + B'$$

$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A' \rightarrow B \sqsubseteq A \rightarrow B'$$

Applications of Polymorphism

Polymorphism allows an elegant formal treatment of record types, objects, and inheritance. We will look at records and provide a simple semantics for them and a simple binary operation to build them.

This account also allows us to define dependent records, a significant extension of the record concept.

Dependent Records

$\{x_1{:}A_1;\ x_2{:}A_1(x_1);x_3{:}A(x_1,x_2);\ldots;x_n{:}A_n(x_1,\ldots,x_{n-1})\}$

functions r from Labels $\{x_1,\ldots,x_n\}$ to values

where $r(x_1) \varepsilon A_1$, $r(x_2) \varepsilon A_2(r(x_1))$, etc.

Record Types and Inheritance

We can define algebraic structures as records. For example, a monoid on carrier S is a record type over S with two components, an associative operator and an identity:

Monoid = { op: S x S -> S; id: S}.

A group extends this record type on S by including an inverse operation.

Group = { op: S x S -> S; id: S; inv:S ->S}

A *Group* is a subtype of a *Monoid* as we show next.

$$Group \sqsubseteq Monoid$$

Groups and Monoids as Records

The basic idea is that the elements of a record type are functions from the field selectors names, e.g. {op,id,inv} to elements of types assigned to them by a mapping called a signature, Sig:{op,id,inv} -> Type. Here are the mappings for a Group over the carrier S.

Sig(op)= SxS->S, Sig(id)=S, Sig(inv)=S->S

Monoid and Group are these dependent function spaces, Group a subtype of Monoid.

$i: \{op,id,inv\} \rightarrow Sig(i) \sqsubseteq i:\{op,id\} \rightarrow Sig(i)$

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.

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

Elements of Co-inductive Types

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

St -> In -> St x 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)) Ξ corec(T.F(T)).

Future Directions

I think we will continue to see constructive type theory unify concepts from logic, mathematics, and computer science. Homotopy type theory is a current example as are game semantics.

We will see applied formal theories support important deployed systems. These theories will be as dynamic as the systems.

Future Directions

A more internal and specialized development will be the erosion of the barrier between mathematics and metamathematics – as Brouwer desired – “there is one mathematics.”

I already see Nuprl applied to itself to verify the optimizations to the evaluators for distributed protocols. These implemented theories and their proof assistants are autocatalytic.

Exercises

1. (P => Q) => ~Q => ~P

   where ~P means (P => False), “not P”.

   In minimal logic, we take a special constant for False, say Any. While (False => P) holds, in minimal logic, (Any => P) does not hold for arbitrary P. Show

2. (P => Q) => ((Q => Any) => (P => Any))

Exercises continued

3. (~(P & Q) & (P v ~P) & (Q v ~Q)) => ~P v ~Q.

4. Ex.(P(x) => C) => ((Allx.P(x)) => C) where C has no free occurrences of x.
5. Ey.~Q(y) => Allx.(P(x) => Q(x)) => ~Allx.P(x).

6. (Ey.Allx.P(x,y) => Eu.Allv.Q(u,v)) => (Ally.(Allx.P(x,y) => Eu.Allv.Q(u,v)).

Hard Exercise

Recall the example:

(((A&B) => (A&B)) => (C v D)) => (C => E) => E

Are these both correct realizers?

$\lambda h.\lambda f.\ \lambda g.\ decide(h(id_1);c.f(c);d.g(d))$

$\lambda h.\lambda f.\ \lambda g.\ ap(decide(h(id_1);c'.f;d'.g);ap(decide(h(id_2);c.c;d.d))$

where $id_1 = \lambda x.x$, id2 = $\lambda x.spread(x;x_1,x_2.<x_1,x_2>)$

For more information, articles, and examples of verified code see www.nurpl.org

THE END

Questions?

Also see www.nuprl.org