HoPL Talk on :

The Implimentation of Dependent Type Proof Assistants

- Ankit Kumar

Proof assistants are software that allow a user to define mathematical structures, and reason about them using machine checkable theorems and proofs.

Do we need Proof Assistants?

* Published results can be faulty

  e.g. An error in the first purported proof of the 4-color theorem [Kempe-1879] was eventually pointed out a decade later [Heawood, 1890]

* A book by [Lecat, 1935] gave 130 pages of errors made by major mathematicians up to 1900

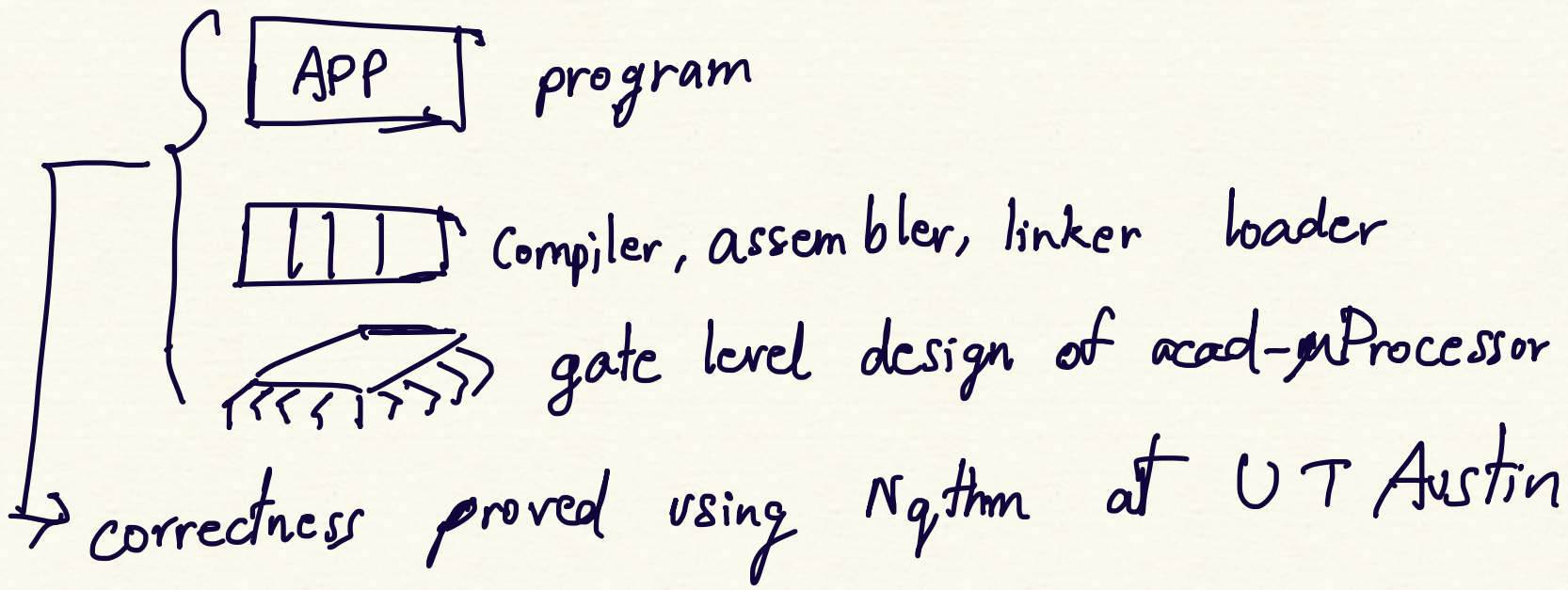
* Explosion of the Ariane 5 rocket in 1996 due to software error, costing $500 Mil

Proof assistants have helped us:

* Even though the 4-color theorem was proved in 1976 by Kenneth Appel and Wolfgang Haken, it was computer assisted and thus, infeasible for a human to check by hand. Gonthier proved it again, using Coq in 2005.

* Xavier Leroy led the CompCert project to produce a verified C compiler back-end robust enough to use with real embedded software.

* The CLI stack:



CURRY-HOWARD ISOMORPHISM

OTHER METHODS

1970

1980

1990

2000

2010

AUTOMATH – de Bruijn

proof checker

LCF – Milner

Scott's Logic of Computable functions.

- theorem : thm
- constructor as Inf. rules
- strongly typed host

NUPRL – R.L. Constable

ISABELLE

MIKE GORDON

HOL88

HOL90

HOL-LIGHT

META-PRL

HOLZERO

MATITA

REDPRL

Coq – Coquand Huet

Griffin

EFS

Elf Pfenning

Twelf

Pfenning

Schürmann

PVS

Classical, typed HOL

Shankar

LEAN

Moura...

Boyer-Moore

N_q Thm

quantifier free

FOL=

or

Primitive Rec Arithmetic

(term rewriting)

ACL2

Voronkov, Hoder

vampire

(resolution)

(resolution) McCune

FOL

otter

|

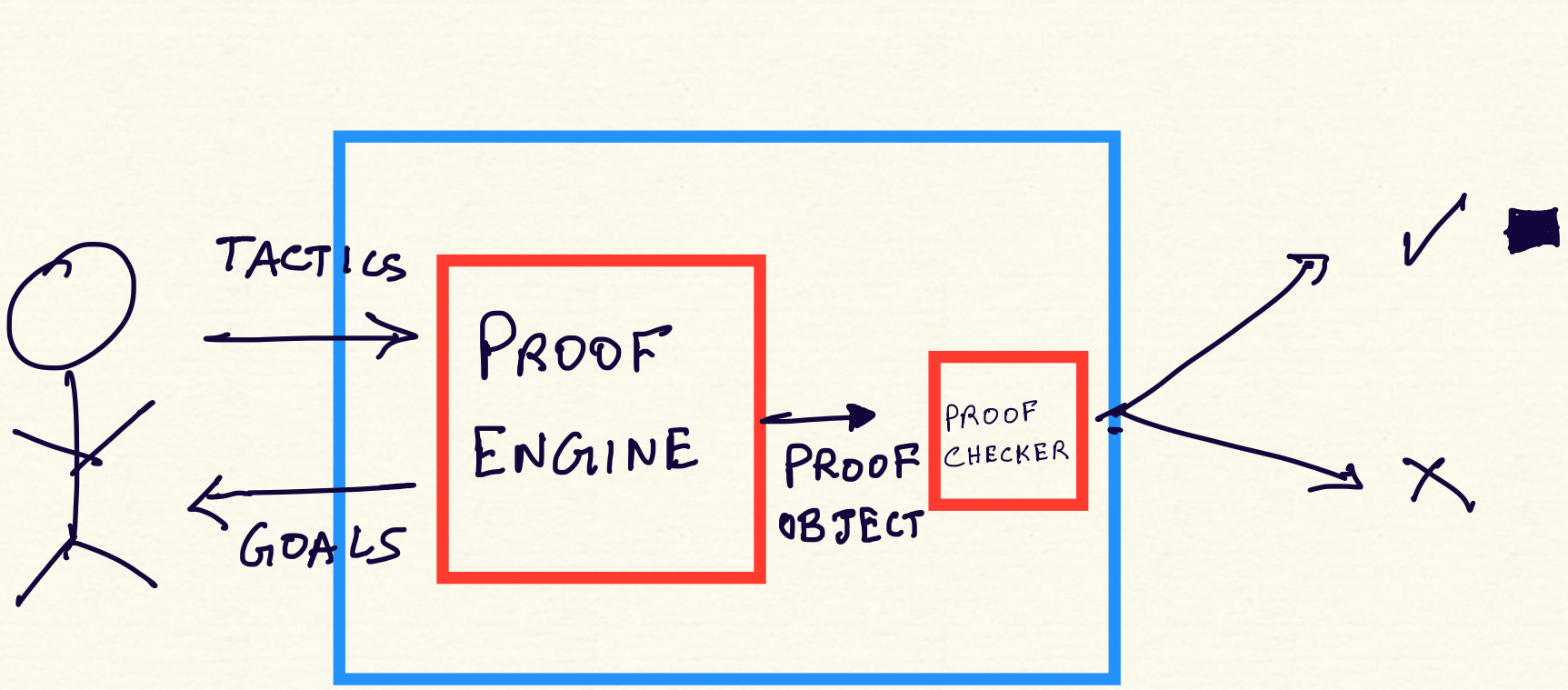
proving

ACL2s

USE CURRY-HOWARD ISOMORPHISM (Lucy's talk)

term : type

proof : proposition



proof development in a type theory based proof assistant

– diagram adapted from "Proof Assistants: History, Ideas & Future – H GEUVERS"

Thm: $\forall f : \mathbb{N} \to \mathbb{B}ool,$

$\exists i, j : \mathbb{N},\ i \neq j \wedge f(i) = f(j)$

Proof: 2 variants

i) Classical

Let $T_s(f) := \forall i, \exists j,\ i < j \wedge f(j) = T$

$F_s(f) := \forall i, \exists j,\ i < j \wedge f(j) = F$

Lemma: $\forall f : \mathbb{N} \to \mathbb{B}ool,$

$T_s(f) \vee F_s(f)$

pick either one!

pick i and j s.t. $f(i) = f(j) = T$

pick i and j s.t. $f(i) = f(j) = F$

ii) Constructive:

Consider $f(0)$, $f(1)$, $f(2)$

if $f(0) == f(1)$

then $i = 0, j = 1$

else if $f(0) == f(2)$

then $i = 0, j = 2$

else

$i = 1, j = 2$ ∎

We have a proof, and a way to pick i and j!

MLTT is a constructive type theory

Both NuPRL and Coq are based on type theories derived from MLTT.

However, there is a fundamental difference between them based on their treatment of

propositions-as-types.

This affects design decisions and implementation, as we will see in the rest of this talk.

Recap:

true and provable are used interchangeably.

Types — Constructive Meaning (Brouwer)

$\bot$ — never true

$A \& B$ — iff A is true and B is true

$A \vee B$ — iff A is true or B is true, and we say which one.

$A \Rightarrow B$ — iff we can effectively transform any proof of A into a proof of B

$\neg A$ — iff $A \Rightarrow \bot$

$\exists x : A . B$ — iff we can construct an element $a : A$ and a proof of $B[a/x]$

$\forall x : A . B$ — iff we have an effective method to prove $B[a/x]$ for any $a : A$

NuPRL - Constable et al 1986

PROOF
REFINEMENT
LOGIC

- Implements Computational Type Theory, a variation of Martin Löfs Intuitionistic Type Theory
- Proofs (Programs) are constructed by interactive refinement, using rules

Propositions - as - types

MLTT
syntactic
(theory of formal proof)

$$E_{m,n} = \begin{cases} \{0\} & \text{if } m=n \\ \emptyset & \text{if } m \neq n \end{cases}$$

$0 : m=n$

CTT
semantic
(theory of Truth)

axiom : $m=n$

intensional

Types are equal by definition

extensional

Types are equal by proof

Direct Computation Rules — Computation System support

$(\lambda n.0)\, a = 0 \in \mathbb{N}$ for any type of a

Extensions

- subset types $\{x : A \mid B(x)\}$
- quotient types $2 = 4 \in \mathbb{Z}_2$
- W-types (wf trees)
- recursive types

Computation System

CTT terms

Canonical	Non-canonical
n i nil	-i a+b
axiom a=b∈A	int_eq(m,n,s,t)
A→B {A | B}	decide(a; x.s; y.t)
...	

Redex:

int_eq(m,n,s,t) ⟶ s if m is n else t

-j ⟶ negation of j

a+b ⟶ sum of a and b

decide(inl(a); x.s; y.t) ⟶ s[a/x]

...

TYPES:

$$T = S \text{ if under } \exists T'.\, T \to T' \wedge T' \equiv S$$

symm, trn.

TERMS:

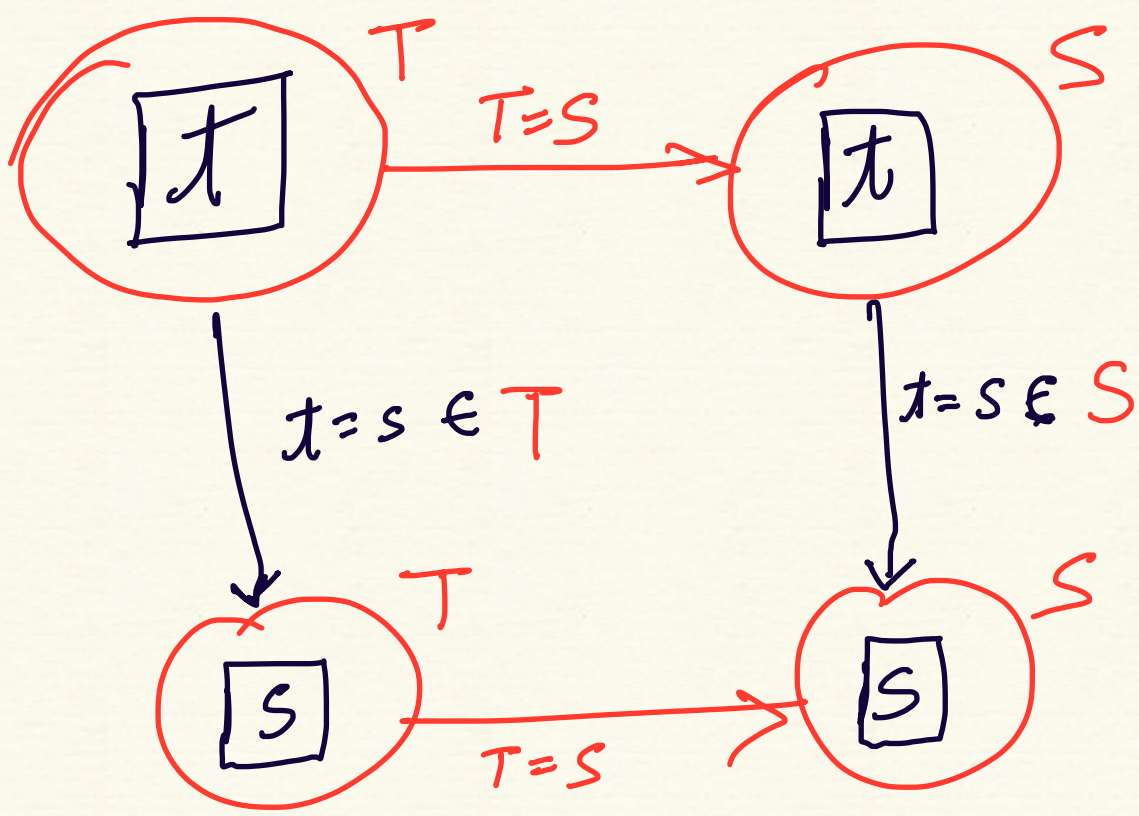
$$t = s \text{ if under } \exists t'.\, t \to t' \wedge t' \equiv s \in T$$

Symm, trn.

T type iff $T = T$ if $t \equiv s \in T$ } inhabited types

$t = s \in T$ if $T = S \wedge t \equiv s \in S$ } extensional equality

$t \in T$ iff $t = t \in T$



type checking is undecidable, carried out using proof tactics

Inductive definition of equality

$T \cong T'$ iff

$T, T' \to$ void

∨ $T, T' \to$ atom

$T, T' \to$ int

or $\exists A, A', a, a', b, b'.\ T \to (a = b \text{ in } A)$

$T' \to (a' = b' \text{ in } A')$

$\wedge$ $A \cong A'$

$a \cong a' \in A$

$b \cong b' \in A$

$t \cong t' \in T$ if $T \cong T'$ & $t = t' \in T'$

$t = t' \in \text{atom}$ iff $\exists i.\ t', t \to i$

$t \cong t' \in \text{int}$ iff $\exists n.\ t, t' \to n$

$t \in (a = b \text{ in } A)$ iff $t \to \text{axiom}$ & $a = b \in A$

Judgements : units of assertion

Let $x_1 \dots x_n$ be variables

$T_1 \dots T_n$ be types s.t. x_i is free in T_j if $i<j$

Then $x_1 : T_1 \dots x_n : T_n \;@\; t_1 \dots t_n$ holds

if every $t_i \in T_i \left[\frac{t_1 \dots t_n}{x_1 \dots x_n} \right]$

SEQUENT

$x_1 : T_1 \dots x_n : T_n \gg S \;[\text{ext } s]$

is true at $t_1 \dots t_n$ iff

$x_1 : T_1 \dots x_n : T_n \;@\; t_1 \dots t_n$ holds

$$\wedge\; \forall\, t_1' \dots t_n'.\; \Big[\; S[t_i'/x_i] = S[t_i/x_i]$$

$$\wedge\; s[t_i'/x_i] = s[t_i/x_i] \in S[t_i/x_i] \;\Big]$$

extensional equality

Judgement form:

Hyps ... ≫ S ext t

extracted term

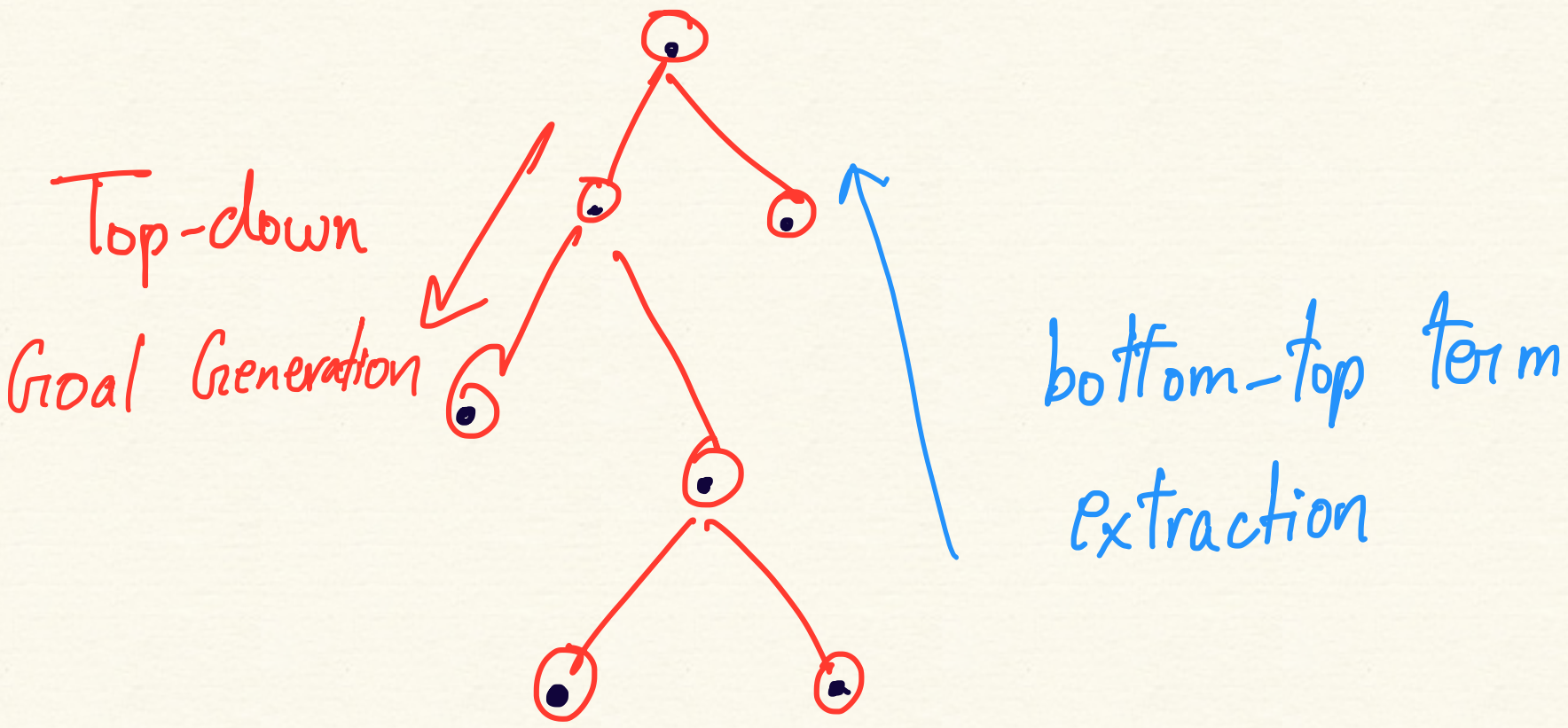
Refinement Rules

$$H \vdash T \ [\text{ext } t] \text{ by rule}$$

$$H_1 \vdash T_1 \ [\text{ext } t_1]$$

$$\vdots$$

$$H_k \vdash T_k \ [\text{ext } t_k]$$



for example:

$H \gg x : A \to B$ ext $\lambda y.\, b$ by intro at U_i [new y]

$y : A \gg B[y/x]$ ext b

$\gg A$ in U_i

$H \gg$ int_eq $(a; b; t; t')$ in $\top$ by intro

$\gg a$ in int

$\gg b$ in int

$a = b$ in int $\gg t$ in T

$(a = b$ in int$) \to$ void $\gg t'$ in T

Thm: All $f: int \to \{0,1\}$. Some $i,j : int$.

$i < j \;\&\; f(i) = f(j)$

PROOF: By intro

1. $f: int \to \{0,1\} \gg$

Some $i,j : int.\ i<j \;\&\; f(i) = f(j)$

By seq $f(1)=0 \mid f(1)=1$

[ext \f. e1]

$f: int \to \{0,1\},\ f(1)=0$ in int $\gg$

Some $i,j : int.\ i<j \;\&\; f(i) = f(j)$

By seq $f(2)=0 \mid f(2)=1$

[ext int_eq (f(1); 0; e2; e3)]

$f: int \to \{0,1\},\ f(1)=0$ in int, $f(2)=0$ in int $\gg$

Some $i,j : int.\ i<j \;\&\; f(i) = f(j)$

[ext int_eq (f(2); 0; e4; e5)]

By intro 1,2

$f: int \to \{0,1\},\ f(1)=0$ in int, $f(2)=0$ in int $\gg$

$0 < 1 \;\&\; f(0) = f(1)$

[ext <1; <2; e>>]

[ext axiom]

2. $(int \to bool)$ in U_1

```
\f. int_eq (f(1); 0;
      int_eq (f(2); 0;
        <1,<2, axiom>>;
          int_eq (f(3); 0;
            <1,<3, axiom>>;
             <2,<3, axiom>>));

    int_eq(f(2); 0;
      int_eq (f(3); 0;
        <2,<3, axiom>>;
          int_eq (f(4); 0;
            <2,<4, axiom>>;
            <3,<4, axiom>>));
    <1,<2, axiom>>))
```

- extracted proof term extracted from

"Finding computational Content in Classical Proofs" - Constable, Murthy

Griffin in '89 extended Curry-Howard isomorphism to scheme, which contains control construct call/cc that allows accesn to current continuation. This relates classical proofs to typed programs.

- Based on Griffin's work Chet and Constable outline a general method of extracting algorithms from classical proofs of Π_2^0 sentences
- Implemented in a fragment of NuPRL's type theory.

NuPRL Cons:

Extracted programs tend to be inefficient.

In order to extract efficient code, need to write complex proofs.

Coq

- Implementation of Calculus of Constructions, launched by Coquand and Huet in '84.

- Inductive types added later by Paulin-Mohring

CoC

Constructions: well typed expressions of typed lambda-calculus

core language has 5 formation rules:

$*$	Universe of all types
$[x:M]N$	PROD
$(\lambda x:N)\,M$	LAM
$(M\ N)$	APP
x	VAR

MLTT	CoC
* Predicative $[A : U_0] A : U_1$	Impredicative $[A : *] A : *$ Propositions live in *

abstraction allowed over

proof : proposition : context

objects

$[x_1 : M_1] [x_2 : M_2] \ldots [x_n : M_n] *$

eg. $P : [x : nat] *$

* Abbr. for Prod:

$\{P \mid [x:nat] *\}\, N$

→ a uniary predicate P over nat in N

* $\{X \mid *\}\, N$ as $\forall x.N$

* $[x:A]\, B$ as $A \to B$ if x doesn't occur in B.

* $([x:A] M_x\ x)$ as let $x = X$ in M_x

$\longrightarrow := \lambda A.\, \lambda B.\, [x:A]\, B : \forall A.\, \forall B.\, *$

Now if $t : ((\to A)\, B) \cong [x:A]\, B$

and $u : A$

then $t\ u\ x$

Need conversion rules.

$\cong$ defined as the smallest congruence over propositions and contexts containing β-conversion defined inductively:

$$\frac{\Gamma \vdash M : N}{\Gamma \vdash M \cong M} \qquad \frac{\Gamma \vdash M \cong N \quad \Gamma \vdash N \cong P}{\Gamma \vdash M \cong P}$$

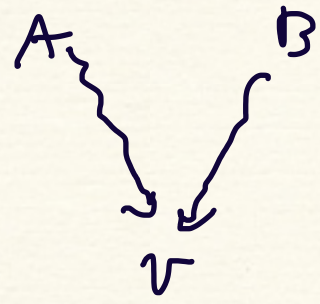
$$\frac{\Gamma \vdash P_1 \cong P_2 \quad \Gamma [x : P_1] \vdash M_1 \cong M_2 \quad \Gamma [x : P_1] \vdash M_1 : N_1}{\Gamma \vdash (\lambda x : P_1)\, M_1 \cong (\lambda x : P_2)\, M_2}$$

$$\frac{\Gamma \vdash (M\,N) : P \quad \Gamma \vdash M \cong M_1 \quad \Gamma \vdash N \cong N_1}{\Gamma \vdash (M\,N) \cong (M_1\,N_1)}$$

$$\frac{\Gamma [x : P] \vdash M : P \qquad \Gamma \vdash N : P}{\Gamma \vdash ((\lambda x : P)\, M\ N) \cong [N/x]M}$$

$$\frac{\Gamma \vdash M : P \quad \Gamma \vdash P \cong Q}{\Gamma \vdash M : Q}$$

1) β-reduction is confluent



2) strong normalization holds.

Theorem: Given Γ and M, it is decidable whether $\Gamma \vdash M : N$. Further if N exists, it can be effectively computed.

Theorem: If $\vdash M : N$ and N is an object, then by removing types from M, we get a pure strongly normalizing λ-term.

The Constructive Engine

Proof object is created by interaction of engine with current construction and env.

$$\frac{\Gamma \vdash M : \mathsf{Prop}}{\Gamma; x : M \vdash}$$

$$\frac{\Gamma \vdash M : \mathsf{Type}(i)}{\Gamma; x : M \vdash}$$

Introducing new Hypotheses.

$$\frac{\Gamma \vdash M : T}{\Gamma; x := M : T \vdash}$$

new definition.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}(1)}$$

Prop intro

$$\frac{\Gamma \vdash M : (x:B)C \quad \Gamma \vdash N : A \quad \Gamma \vdash A \cong B}{\Gamma \vdash (M\,N) : [N/x]\,C}$$

App Intro

$$\frac{\Gamma;\ x = M : T \vdash N : A}{\Gamma \vdash [x/M]N : [x/M]A}$$

Discharging Definition.

Intensional equality

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \mathrm{Prop} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B}$$

Type Conversion

These rules are invoked by higher level commands and tactics

The Mathematical Vernacular

2 sub languages

expressions

names,

Prop, Type (n),

(M N), $(x : M)\, N_x$ (forall),

$[x : M]\, N_x$ (function),

let $x = M$ in N_x

commands

Hypothesis $x : T$

Axiom $x : A$

Variable $x : T$

Theorem x P Proof M

checks that M is a proof of P and defines it under x.

Theorem example: $\forall (f: \mathbb{N} \to \mathrm{Bool}),$

$\exists i, j, \ (f\ i) = (f\ j).$

PROOF: intros

```
destruct (f 0) eqn: H1.
destruct (f 1) eqn: H2.
 symmetry in H2.
rewrite H2 in H1.
exists 0, 1.
 auto; easy.
 destruct (f 2) eqn: H3.
 symmetry in H3.
 rewrite H3 in H1
 exists 0, 2.
 auto; easy.
  .
  .
  .
  }
```

```
example :=
  fun f : nat → bool ⇒
let b := f 0 in
 let H1 : f 0 = b :=
     eq_refl in
(if b as b0
 return (f 0 = b0 →
     exists i,j : nat, fi = fj)
then
 fun H2 : f 0 = true ⇒
  let b0 := f 1 in
  let H3 : f 1 = b0 :=
    eq_refl in
      .
      .
      .
```

CTT (NuPRL)	CoC (Coq)
1. Completely Predicative	Set, Type are Predicative Prop is impredicative
2. Proof objects are computational programs. Can't construct object in $\perp$	Proof objects are symbolic. eg. Let $t : \forall x:Prop.\ x$ ($\perp$) $t\ p = p$ type- Reducto-Ad-Absurdum
3. Extensional type theory → good for Univalent Foundations	Intensional Type Theory
4. So, type-checking undecidable	Hence decidable type checking
5. Full power of Y-combinator, Turing Complete	Terms are strongly normalizing