TFPIE 2016 talk

Proust:
A Nano Proof Assistant

Prabhakar Ragde
University of Waterloo

Marcel Proust

Problem:

What are proof assistants actually doing?

```
(* Coq code *)

Theorem plus_right_id:
  forall n, n + 0 = n.
Proof.
  induction n.
  - simpl. reflexivity.
  - simpl. rewrite IHn.
    reflexivity.
Qed.
```

```
plus_right_id =
fun n : nat =>
 nat_ind
   (fun n0 : nat => n0 + 0 = n0) eq_refl
   (fun (n0 : nat) (IHn : n0 + 0 = n0) =>
     eq_ind_r (fun n1 : nat => S n1 = S n0)
              eq_refl IHn)
   n
: forall n : nat, n + 0 = n
```

Curry-Howard correspondence

Propositions are types.

Proofs are programs.

(Not enough.)

To demystify:
Write a typechecker.

(For dependent types.)

Another problem:

How to teach logic in CS undergraduate core?

Inconsistent content.
Confining.
Boring.
Irrelevant.
Impractical.

Can Curry-Howard help?

Develop Proust as a typechecker **and** proof assistant.

Students write proofs as code to be checked by Proust.

Students see development
of Proust code
(there isn’t much)
and write some of it.

First:

Propositional logic

BHK interpretation (1920's):

A proof of $T \to W$ is a construction transforming a proof of T into a proof of W.

How to represent proof terms?

Simply-typed lambda calculus?

Too much notational overhead.

Untyped proof terms.

Bidirectional type system.

Checking and inference modes.

Syntax-driven algorithm.

Natural deduction (Gentzen 1935).

Introduction & elimination rules.

$\longrightarrow$ introduction is λ.

$\longrightarrow$ elimination is function application.

$$\vdash \lambda x.\lambda y.y\ x : A \to (A \to B) \to B$$

So far: no notion of computation.

Proust is written in Racket.

User writes S-expression syntax.

```
expr = (λ x => expr)
     | (expr expr)
     | (expr : type)
     | x

type = (type -> type)
     | X
```

Easily parsed into AST.

```
(struct Lam (var body))
(struct App (rator rand))
(struct Ann (expr type))

(struct Arrow (domain range))
```

```
; parse-expr : sexp -> Expr

(define (parse-expr s)
  (match s
    [`(λ ,(? symbol? x) => ,e) (Lam x (parse-expr e))]
    [`(,e1 ,e2) (App (parse-expr e1) (parse-expr e2))]
    [(? symbol? x) x]
    [`(,e : ,t) (Ann (parse-expr e) (parse-type t))]
    [else (error 'parse "bad syntax: ~a" s)]))
```

Students can extend the parser and pretty-printer as needed.

The Proust code is a straightforward implementation of the checking/inference rules.

```
; type-check : Context Expr Type -> boolean

(define (type-check ctx expr type)
  (match expr
    [(Lam x t)
       (match type
         [(Arrow tt tw)
            (type-check (cons `(,x ,tt) ctx) t tw)]
         [else (cannot-check ctx expr type)])]
    [else (if (equal? (type-infer ctx expr) type)
                      true
                      (cannot-check ctx expr type))]))
```

```
; type-infer : Context Expr -> Type

(define (type-infer ctx expr)
  (match expr
    [(Lam _ _) (cannot-infer ctx expr)]
    [(Ann e t) (type-check ctx e t) t]
    [(App f a)
       (define tf (type-infer ctx f))
        (match tf
          [(Arrow tt tw) #:when (type-check ctx a tt) tw]
          [else (cannot-infer ctx expr)])]
    [(? symbol? x)
       (cond
         [(assoc x ctx) => second]
         [else (cannot-infer ctx expr)])]))
```

```
(define (check-proof p)
  (type-infer empty (parse-expr p)) true)

(check-expect
  (check-proof
    '((λ x => (λ y => (y x))) : (A -> ((A -> B) -> B))))
  true)
```

Where’s the interaction?

Use Racket's REPL.

Add `?` for hole/goal
(incomplete part of proof).

State variables:

- Current expression
- Goal counter
- Goal table (number to type)

`set-task!` initializes state variables.

```
(set-task! '(? : (A -> ((A -> B) -> B))))
```

`(refine n s)` refines goal `n` with expression `s`.

```
(refine 0 '(λ x => ?))
```

Goal 1 now `(((A -> B) -> B))`,
context types `x` as `A`.

```
> (set-task! '(? : (A -> ((A -> B) -> B))))
Task is now (?0 : (A -> ((A -> B) -> B)))
> (refine 0 '(λ x => ?))
Task is now ((λ x => ?1) : (A -> ((A -> B) -> B)))
> (refine 1 '(λ y => ?))
Task is now
((λ x => (λ y => ?2)) : (A -> ((A -> B) -> B)))
> (refine 2 '(y ?))
Task is now
((λ x => (λ x => (y ?3))) : (A -> ((A -> B) -> B)))
> (refine 3 'x)
Task is now
((λ x => (λ x => (y x))) : (A -> ((A -> B) -> B)))
```

We reason about the rules for the other logical connectives.

Students can add the code.

A proof of $T \wedge W$ is a proof of T and a proof of W.

$\wedge$-introduction is pairing,
$\wedge$-eliminations are projections.

A proof of $T \vee W$ is either
a proof of T or a proof of W.

$\vee$-introductions are injections,
$\vee$-elimination is case expression.

There is no proof of $\bot$.

From $\bot$, one can prove anything.

$\neg T$ is $T \to \bot$.

This is natural deduction for intuitionistic propositional logic.

It doesn’t have proof by contradiction or the law of the excluded middle.

Next:

Predicate logic

Goal:

To be able to say and prove
"For all natural numbers n,
$n + 0 = n$."

We rewind to just implication
without holes, and refactor to
mingle terms and types.

```
expr = (λ x => expr)
     | (expr expr)
     | (expr : expr)
     | x
     | (expr -> expr)
     | X
```

`type-infer` now checks well-formedness of types.

We need:

- Quantifiers $\forall$, $\exists$
- Things to quantify over
- Computation

First-order predicate logic gives us only the first one.

Higher-order logic is
more natural and expressive.

It is what Coq and Agda use.

We add $\forall$, but want to quantify over more than a single unspecified domain.

$\forall X\ W$ becomes $\forall(X : T) \to W$.

A proof of $\forall(X : T) \to W$ is a construction which permits us to transform $x : T$ into a proof of $W[X \mapsto x]$.

This is a generalized lambda. (Dependent type.)

$\rightarrow$ is a special case of $\forall$.

λ introduces $\forall$.

Using a $\forall$ is function application.

We add *Type* as a constant.

(Can quantify over propositions, write polymorphic functions.)

We choose $\mathit{Type} : \mathit{Type}$.

We need:

- Substitution
- α-conversion
- α-equivalence

We can add a simple definition mechanism.

We also need:

- Reduction to WHNF
- Full β-reduction (for definitional equality of types)

These give us computation, and more compact proofs.

```
expr = (λ x => expr)
     | (expr expr)
     | (expr : expr)
     | x
     | (expr -> expr)
     | (∀ (x : expr) -> expr)
     | Type
```

This minimal language
is surprisingly expressive.

We can encode:

- Booleans
- Pairs
- Natural numbers
- All logical connectives

Or, as before, we add syntax,
and reason about the rules.

(Simpler, more efficient,
and students can add code.)

Equality introduction:
`(eq-refl T)` proves `(T = W)`
when `T` and `W` are
definitionally equivalent.

How do we use or eliminate
an equality?

The induction principle
for equality:

$$\forall(A : \mathit{Type})(x : A)(P : A \to \mathit{Type})$$
$$\to (P\ x)$$
$$\to \forall(y : A) \to (x = y) \to (P\ y)$$

```
(define (type-infer ctx expr)
  (match expr
    ...
    [(Eq-elim x P px y peq)
       (define A (type-infer ctx x))
       (type-check ctx P (Arrow '_ A (Type)))
       (type-check ctx px (weak-reduce (App P x)))
       (type-check ctx y A)
       (type-check ctx peq (Teq x y))
       (weak-reduce (App P y))]
    ...))
```

We do something similar for natural numbers.

```
expr = ...
     | Z
     | (S expr)
     | Nat
     | (nat-ind ...)
```

Induction principle:

$$\forall(P : Nat \to Type)$$
$$\to (P\ Z)$$
$$\to (\forall(k : Nat) \to ((P\ k) \to (P\ (S\ k))))$$
$$\to (\forall(n : Nat) \to (P\ n))$$

If P ignores its argument,
we have a recursor.

```
(def 'nat-rec
  '((λ C => (λ zc => (λ sc => (λ n =>
      (nat-ind (λ _ => C) zc (λ _ => sc) n)))))
    : (∀ (C : Type) -> (C -> (C -> C) -> Nat -> C))))

(def 'plus
  '((λ n => (nat-rec (Nat -> Nat) (λ m => m)
                     (λ pm => (λ x => (S (pm x)))) n))
    : (Nat -> Nat -> Nat)))

(def 'plus-zero-left
  '((λ n => (eq-refl n))
    : (∀ (n : Nat) -> ((plus Z n) = n))))
```

```
(def 'plus-zero-right
  '((λ n =>
      (nat-ind (λ m => ((plus m Z) = m)) (eq-refl Z)
               (λ k => (λ p =>
                 (eq-elim
                   (plus k Z)
                   (λ w => ((S (plus k Z)) = (S w)))
                   (eq-refl (S (plus k Z))) k p)))
               n))
      : (∀ (n : Nat) -> ((plus n Z) = n))))
```

We can continue to add features.

But it is probably time to switch to a real proof assistant such as Coq or Agda.

These add:

- holes/goals (properly treated);
- dependent pattern matching;
- user-defined inductive datatypes;
- tactics to construct proofs.

Summary:

Proust is a family of small Racket programs that act as rudimentary proof assistants.

They demystify larger proof assistants, and can be used to teach logic to CS students.

Thanks to:
Stephanie Weirich
Lennart Augustsson
Andrej Bauer
Altenkirch, Danielsson, Löh, Oury
Löh, McBride, Swierstra
Philip Wadler
and many others