Proust: A Nano Proof Assistant

Prabhakar Ragde
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
plragde@uwaterloo.ca

Proust is a small Racket program offering rudimentary interactive assistance in the development of verified proofs for propositional and predicate logic. It is constructed in stages, some of which are done by students before using it to complete proof exercises, and in parallel with the study of its theoretical underpinnings, including elements of Martin-Löf type theory. The goal is twofold: to demystify some of the machinery behind full-featured proof assistants such as Coq and Agda, and to better integrate the study of formal logic with other core elements of an undergraduate computer science curriculum.

1 Introduction

Most computer science programs include some exposure to logic as a required subject. Often it is crammed into a single course on discrete mathematics, perhaps conflated with Boolean algebra and wedged between combinatorial counting and graph theory. At the University of Waterloo, where I teach, we are fortunate to have CS located in a Faculty of Mathematics, so our students take the same high-quality math courses (discrete and continuous) taken by math majors. Nonetheless, we have a required second-year CS course titled Logic and Computation (henceforth L&C).

L&C was a relatively recent addition to our curriculum (about fifteen years ago). While it is a prerequisite for the data structures and algorithms courses, that is more a matter of maturity rather than content, and consequently the content of L&C has tended to vary across offerings, from "math-style" (Hilbert proofs, emphasis on metamathematics) to "SE-style" (emphasis on program correctness). The efficacy of these approaches is open to question, especially since the amount of material available encourages broad, shallow treatments, and students may tend to come away with exposure to what an individual instructor thinks is important, rather than what might be most beneficial for future courses or careers.

It might make more sense to deal with some of these topics as they arise in other courses. For example, students in our first-year CS sequence learn structural recursion on natural numbers and lists, and this might be a place to prove properties of the code (starting with correctness) by induction. But these properties, as well as the subtasks involved in proofs by induction, involve manipulation of logical statements with for-all quantifiers, and students have difficulties with this. At least some of these difficulties stem from the fact that for-all is a binding construct whose use has consequences similar to the use of lambda. Proofs might better be deferred until after students have experience with higher-order functions.

This similarity between for-all and lambda suggests another approach. In the Curry-Howard correspondence, logical statements (propositions or formulas) correspond to types, and the proof of a for-all statement is a dependently-typed function (a generalized lambda). At first glance, the idea of using dependent types to introduce undergraduates to logic may seem misguided. But this approach makes use of the fact that CS students already have experience with a formal system: a programming language.

Submitted to:
TFPIE

Furthermore, our undergraduates have had a first course in functional programming, including both theoretical and concrete use of a substitution model (beta-reduction, though not called that or defined in full generality) to explain the execution of programs.

Consider the interpretation of an implication. The way we prove a statement of the form "If T, then V" is by assuming we have a proof of T, and using that assumption at several points in a proof of V. If we were then given an actual proof of T, we could substitute that for the assumption every place it occurs, and get a self-contained proof of V. The substitution process corresponds to the substitution of an argument value for a parameter name everywhere in the body of a function. Thus a proof of "If T, then V" corresponds to a function that consumes a proof of T and produces a proof of V, and the use of such an implication in a proof corresponds to function application.

To a nascent computer scientist, this is a powerful metaphor, especially when they learn that "If T, then V" is usually written in formal logic as $T \to V$, which is also the notation for the type of a function from T to V. The metaphor has more resonance than a representation of a proof as a tree or a DAG. This is the starting point of the several construction steps that result in Proust, a program that both assists students in building proofs (represented by functions) and checks their validity. The construction process is an integral part of the L&C course, with some programming tasks being homework exercises. I will describe the development of Proust, as connectives and quantifiers are added incrementally, in sections 2 and 3, followed by discussion in section 4 of wider curricular issues.

I have chosen to use the Racket programming language [9] to implement Proust, because that is the language used by our students in first year (plus all the reasons why we made that choice), and because S-expressions are particularly convenient as a data representation, but other functional languages could also be used. The design is a synthesis from many sources, notably the OPLSS 2014 lectures of Stephanie Weirich [11] and her work with collaborators on the Trellys project; blog posts by Lennart Augustsson [3] and Andrej Bauer [4]; papers on small dependently-typed systems for tutorial purposes (e.g. Altenkirch, Danielsson, Löh, and Oury [2]; Löh, McBride, and Swierstra [8]); monographs (e.g. Sorensen and Urzyczyn [10]); and the extensive literature on the languages Agda [1] and Coq [5].

2 Propositional Logic

Starting with the implication metaphor discussed in section 1, I derive proof rules for intuitionistic propositional logic (normally called inference rules, but that word will be useful elsewhere). The Proust program that checks proofs is a straightforward implementation of these rules. Along the way, though, I make some unconventional choices for pedagogical benefit.

From the previous discussion, we have types (logical statements, by the Curry-Howard correspondence) constructed from variables ($A, B, C \ldots$) and implication (infix $\to$). We also have functions (proof terms, by the correspondence) with parameters (also usually called variables, $x, y, z \ldots$), and function application. I will use lambda-calculus notation for terms (also called expressions) in the discussion here, but the Proust implementation adapts the syntax somewhat to ease the Racket implementation. In sequents (defined shortly) and proof rules, I will use T, V, W as type metavariables, and f, g, a, b, c, t, v, w as term metavariables. $t : T$ will be the assertion that term t has type T.

To check $\lambda x.t : T \to W$, we need to check $t : W$, but we need to do so with the knowledge that $x : T$, as x is likely to occur in t. This suggests maintaining a set of name-type bindings Γ (typically called a context), yielding a three-way relation $\Gamma \vdash t : W$ to be checked (this is a sequent). But checking a function application gives us some slight trouble. To check $\Gamma \vdash f\ a : W$, we need to check $\Gamma \vdash f : T \to W$ and $\Gamma \vdash a : T$. Where does T come from? We must infer it by looking at f.

This suggests a bidirectional approach [6]. I introduce the notation $\Gamma \vdash t \Leftarrow W$ to refer to the idea of type checking we started with, namely that term t has given type W in context Γ. But for type inference, I use the notation $\Gamma \vdash t \Rightarrow T$, indicating that in context Γ we are able to infer that t has type T. To infer the type of a term that is simply a variable, we look it up in the context. Denoting $\Gamma \cup \{x : T\}$ as $\Gamma, x : T$, here are the proof rules derived so far.

$$\frac{}{\Gamma, x : T \vdash x \Rightarrow T}\ (\textsc{Var}) \qquad \frac{\Gamma, x : T \vdash t \Leftarrow W}{\Gamma \vdash \lambda x.t \Leftarrow T \to W}\ (\to_E) \qquad \frac{\Gamma \vdash f \Rightarrow T \to W \qquad \Gamma \vdash a \Leftarrow T}{\Gamma \vdash f\ a \Rightarrow W}\ (\to_I)$$

Rules are given names with subscripts that indicate elimination (use) and introduction (creation), a pattern that will recur with other logical connectives. To typecheck a term for which we don't have a proof rule, we infer it and check that we get the same type. We cannot infer the type of a lambda, which means we also cannot check the type of an immediate application of a lambda. I discuss the curricular implications in section 4; for convenience, we introduce optional type annotation on terms. Here are the additional proof rules.

$$\frac{\Gamma \vdash t \Rightarrow T}{\Gamma \vdash t \Leftarrow T}\ (\textsc{Turn}) \qquad \frac{\Gamma \vdash t \Leftarrow T}{\Gamma \vdash (t : T) \Rightarrow T}\ (\textsc{Ann})$$

Note that we are avoiding both the overhead of mandatory type annotation on term variables (as in the simply-typed lambda calculus) and the need to discuss Hindley-Milner-style type inference. What we have gained by this approach is a set of syntax-directed proof rules which have a clear implementation as a pair of mutually-recursive procedures for checking and inferring.

Here is the initial grammar for types and terms, as the user will specify them when using Proust.

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

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

Note that the program does not enforce the lexical restrictions on choice of variables described above, or even the convention that parameters are lower-case while type variables are upper-case; any symbols may be used. The program parses user input into an AST built with Racket structures (records), using Racket's built-in pattern matching.

```
(struct Lam (var body))
(struct App (rator rand))

(struct Arrow (domain range))
```

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

(define (parse-expr s)
  (match s
    [`(λ ,(? symbol? x) => ,e)
       (Lam x (parse-expr e))]
    [`(,e0 ,e1)
       (App (parse-expr e0)
            (parse-expr e1))]
    [(? symbol? x) x]))
```

```
; parse-type : sexp -> Type

(define (parse-type t)
  (match t
    [`(,t1 -> ,t2)
        (Arrow (parse-type t1)
               (parse-type t2))]
    [(? symbol? X) X]
    [else (error ...)]))
```

I have elided the error message in `parse-type`. When the type and term languages are extended with new constructs in order to deal with additional logical connectives, students can implement the requisite parser extensions themselves. The code for type checking and inferring follows a similar pattern, guided by the grammar and the proof rules. Contexts are represented as association lists.

```
; type-check : Context Expr Type -> boolean
; produces true if expr has type t in context ctx (or error if not)

(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
; produces type of expr in context ctx (or error if can't)

(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)])]))

; a simple way to test: a proof term annotated with expected type
(define (check-proof p) (type-infer empty (parse-expr p)) true)
```

The `cannot-check/infer` procedures raise an error whose message includes pretty-printed representations of useful information (context, term, type); the pretty-printers use matching, and again can be written/extended by students.

We have a type checker for the implicational fragment of intuitionistic propositional logic. The size of the code base thus far: 4 lines for datatypes, 12 lines for parsing, 21 lines for checking/inference, one line for a test function, and not shown here, 16 lines for pretty-printing and 6 lines for error handling, for a total of 60 lines.

This program lets us check simple proofs such as

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

but there are also more interesting things we can check, such as the proof of the transitivity of implication, which is the function composition operator:

$$\lambda f.\lambda g.\lambda x.f\ (g\ x) : (B \to C) \to ((A \to B) \to (A \to C))$$

Because this is intuitionistic logic, there are some things we might expect to be able to prove at this point, but cannot, such as Peirce's Law: $((A \to B) \to A) \to A$. In section 4, I discuss how one might deal with this in the classroom.

I have not yet delivered on the promise of an interactive proof assistant. To this end we make use of DrRacket's REPL (read-evaluate-print loop), as well as its support for hash tables. We introduce the notion of holes (or goals), which are unfinished parts of the proof. A hole is a term represented by `?` in the term language and by a `Hole` structure in the AST with a field for a number. A hole typechecks with any provided type. In Proust, we define state variables for the current expression, goal table (a hash table mapping goal number to required type), and a counter to number new holes, handled as simply as possible.

```
(define current-expr #f)
(define goal-table (make-hash))
(define hole-ctr 0)
(define (use-hole-ctr) (begin0 hole-ctr (set! hole-ctr (add1 hole-ctr))))
(define (print-task) (printf "~a\n" (pretty-print-expr current-expr)))
```

The `set-task!` procedure initializes state variables for a new task (described as an S-expression). A typical application is `(set-task! '(? : T))`, where `T` is a logical statement to be proved.

```
(define (set-task! s)
  (set! goal-table (make-hash))
  (set! hole-ctr 0)
  (define e (parse-expr s))
  (match e
    [(Ann _ _) (set! current-expr e) (type-infer empty e)]
    [else (error "task must be of form (term : type)")])
  (printf "Task is now\n")
  (print-task))
```

The `refine` procedure refines goal n with expression s, but makes sure it typechecks first. If it does, it removes goal n and rewrites the current expression to replace that goal with s, using `replace-goal-with`, which does straightforward structural recursion on the AST (the goal being at a single leaf).

```
(define (refine n s)
  (match-define (list t ctx)
    (hash-ref goal-table n (lambda () (error 'refine "no goal numbered ~a" n))))
  (define e (parse-expr s))
  (type-check ctx e t)
  (hash-remove! goal-table n)
  (set! current-expr (replace-goal-with n e current-expr))
  (printf "Task is now\n" (format "~a goal~a" ngoals (if (= ngoals 1) "" "s")))
  (print-task))
```

With still under a hundred lines of code, we now have the capability of interactions like the following, which proves the "transitivity of implication" result described above. The > character is the REPL prompt. The interaction is crude, but genuinely helpful (more so with simple commands to pretty-print information about the goal and its context), and foreshadows similar ideas in interactions with Agda and Coq.

```
> (set-task! '(? : ((B -> C) -> ((A -> B) -> (A -> C)))))
Task is now (?0 : ((B -> C) -> ((A -> B) -> (A -> C))))
> (refine 0 '(λ f => ?))
Task is now ((λ f => ?1) : ((B -> C) -> ((A -> B) -> (A -> C))))
> (refine 1 '(λ g => ?))
Task is now ((λ f => (λ g => ?2)) : ((B -> C) -> ((A -> B) -> (A -> C))))
> (refine 2 '(λ x => ?))
Task is now ((λ f => (λ g => (λ x => ?3))) : ((B -> C) -> ((A -> B) -> (A -> C))))
> (refine 3 '(f ?))
Task is now
((λ f => (λ g => (λ x => (f ?4)))) : ((B -> C) -> ((A -> B) -> (A -> C))))
> (refine 4 '(g ?))
Task is now
((λ f => (λ g => (λ x => (f (g ?5))))) : ((B -> C) -> ((A -> B) -> (A -> C))))
> (refine 5 'x)
Task is now
((λ f => (λ g => (λ x => (f (g x))))) : ((B -> C) -> ((A -> B) -> (A -> C))))
```

The strength of the metaphor persists for the connectives $\wedge$ and $\vee$. A proof of $T \wedge W$ is a proof of T and a proof of W, so the proof term is a two-field record. We could call it `cons`, but it makes proofs more readable if we call the constructor (of the proof term that uses the introduction rule) `∧-intro`, and call the accessor functions (used in proof terms that use the two elimination rules) `∧-elim0` and `∧-elim1`.

$$\frac{\Gamma \vdash t \Leftarrow T \quad \Gamma \vdash w \Leftarrow W}{\Gamma \vdash \wedge_{intro}\ t\ w \Leftarrow T \wedge W}\ (\wedge_I) \qquad \frac{\Gamma \vdash v \Leftarrow T \wedge W}{\Gamma \vdash \wedge_{elim0}\ v \Leftarrow T}\ (\wedge_{E0}) \qquad \frac{\Gamma \vdash v \Leftarrow T \wedge W}{\Gamma \vdash \wedge_{elim1}\ v \Leftarrow W}\ (\wedge_{E1})$$

$$\frac{\Gamma \vdash t \Rightarrow T \quad \Gamma \vdash w \Rightarrow W}{\Gamma \vdash \wedge_{intro}\ t\ w \Rightarrow T \wedge W}\ (\wedge_I) \qquad \frac{\Gamma \vdash v \Rightarrow T \wedge W}{\Gamma \vdash \wedge_{elim0}\ v \Rightarrow T}\ (\wedge_{E0}) \qquad \frac{\Gamma \vdash v \Rightarrow T \wedge W}{\Gamma \vdash \wedge_{elim1}\ v \Rightarrow W}\ (\wedge_{E1})$$

A proof of $T \vee W$ is either a proof of T or a proof of W, so there are two introduction rules and two constructors, `∨-intro0` and `∨-intro1`. To use a proof of $T \vee W$, we need a way to use it if it is a proof of T, and if it is a proof of W. If we want to produce a proof of V, we must have a proof of $T \to V$ we can apply, and a proof of $W \to V$. The deconstructor `∨-elim` is a variation on a case expression. (It is normally made a new binding form to avoid mixing connectives in rules, but we already have a perfectly good binding form, namely lambda.)

$$\frac{\Gamma \vdash t \Leftarrow T}{\Gamma \vdash \vee_{intro0}\ t \Leftarrow T \vee W}\ (\vee_{I0}) \qquad \frac{\Gamma \vdash w \Leftarrow W}{\Gamma \vdash \vee_{intro1}\ w \Leftarrow T \vee W}\ (\vee_{I1}) \qquad \frac{\Gamma \vdash v \Leftarrow T \vee W \quad \Gamma \vdash f \Leftarrow T \to V \quad \Gamma \vdash g \Leftarrow W \to V}{\Gamma \vdash \vee_{elim}\ v\ f\ g \Leftarrow V}\ (\vee_E)$$

Finally, we have negation, whose treatment remains intuitive but may seem incomplete. The logical constant $\bot$ denotes an absurdity or contradiction (we use this for both the term and its type). $\bot$ has no constructor, but its eliminator `⊥-elim` can have any type. We define $\neg T$ to be $T \to \bot$, and use this as a desugaring rule in the parser.

$$\frac{\Gamma \vdash t \Leftarrow \bot}{\Gamma \vdash \bot_{elim}\ t \Leftarrow T}\ (\bot_E) \qquad \frac{}{\Gamma \vdash \bot \Leftarrow \bot}\ (\bot) \qquad \frac{}{\Gamma \vdash \bot \Rightarrow \bot}\ (\bot)$$

There are now more statements we cannot prove: $A \vee \neg A$, $\neg\neg A \to A$, and some of deMorgan's laws. But there are plenty of things we can prove, and this is a good point to assign some proof exercises – once students implement the necessary changes to the parser, pretty-printer, type checking, and type inference functions, because these are also reasonable programming exercises. Each connective requires a very small number of additional lines of code.

While we are writing proof terms that are functions in a language tantalizingly close to the one we are programming in, there is no notion of computation in that language so far. We draw conclusions about the AST, but we do not interpret or otherwise execute the proof-programs. In effect, we have only an evocative tree representation using S-expressions. This will change in the next section.

3 Predicate Logic

A conventional treatment introduces predicate logic by adding relational symbols and first-order quantification to propositional logic, possibly followed by axioms for equality and arithmetic. But just as students benefit from the use of higher-order functions in programming, they can benefit from the increased expressivity of higher-order logic. Our goal in this section, and in the development of Proust, is to reach the point where we can state and prove a suitable translation of "For all natural numbers n, $plus(n,0) = n$", where *plus* is a user-implemented function using structural induction on the first argument (necessitating a proof by induction).

In the previous section, we maintained a grammatical distinction between terms and types. But this example illustrates the need to eliminate this distinction. We rewind to the first program of section 3, handling the implicational fragment of propositional logic without holes, and merge the grammar rules.

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

The grammar is too permissive. We enforce well-formedness in code, adding a `Type` structure which will be the type inferred for a term which satisfies the previous grammar rule for types. We will soon add *Type* to the term language, but not yet, as we need to refactor the program so that the structure of our functions mirrors that of the merged grammar. Having done that, we are ready to consider quantification. As before, we will reason about what the proof rules should be, and what proof terms should look like.

We will write a for-all statement as $\forall(x : T) \to W$, where T and W are terms. How do we use such a statement in an informal proof? We instantiate x with a specific value t of type T, substituting t for x everywhere that x occurs in W. Using the notation $W[x \mapsto t]$ for this substitution, and leaving the proof terms unspecified for the moment, the rule looks like this:

$$\frac{\Gamma \vdash ? \Rightarrow \forall(x : T) \to W \qquad \Gamma \vdash t \Leftarrow T}{\Gamma \vdash ? \Leftarrow W[x \mapsto t]} \ (\forall_E)$$

If we consider the special case where W does not use x, the substitution has no effect, and the rule looks like implication elimination. This suggests that for-all is a generalization of implication, that the introduction proof term should be a generalization of lambda, and that the elimination proof term should be function application. We can keep implication in the term language for convenience, and desugar it to an equivalent for-all. This version of for-all is a dependent product type, often written Π in the literature (as in the title of [2]).

$$\frac{\Gamma, x : T \vdash t \Leftarrow W}{\Gamma \vdash \lambda x.t \Leftarrow \forall(x : T) \to W} \ (\forall_I) \qquad \frac{\Gamma \vdash f \Rightarrow \forall(x : T) \to W \qquad \Gamma \vdash t \Leftarrow T}{\Gamma \vdash f\ t \Leftarrow W[x \mapsto t]} \ (\forall_E)$$

Once we add *Type* as a constant to the language of terms, we can, for example, write the polymorphic identity function as $\lambda x.x : \forall(y : \mathit{Type}) \to (y \to y)$. We remove propositional variables from the term language; we only have variables that are parameters. But if *Type* is a value, what is its type? Here I make a decision that is contrary to our goal of a reliable proof assistant: *Type* has type *Type*. This was done by Martin-Löf in the earliest version of his type theory, but it was shown by Girard [7] to result in an inconsistent logic. The proof is complicated and not a simple, direct construction as in Russell's paradox, so it is unlikely to be an issue for students. Inconsistency can be avoided by a hierarchy of types ($\mathit{Type}_0 : \mathit{Type}_1$, etc.), and this is what is done in Coq and Agda, though Coq hides the details from the user until they become relevant. Managing this hierarchy is not difficult, but it is tedious, and some of the tutorial materials discussed in section 1 choose to avoid it in the same fashion as I have ([2],[8],[11]), while others tackle it ([4]). Avoidance seems best in a context where we are moving towards use of full-featured assistants.

Implementing the above rules requires implementing substitution, with attention to variable reuse and variable capture. There are places in the code where the Racket predicate `equal?` is used to compare types for structural equality (for example, in the implementation of the "turn" rule). But the polymorphic identity function mentioned above could equally well be typed as $\forall(z : \mathit{Type}) \to (z \to z)$. The name of the variable should not matter. In other words, we need alpha-equivalence. These topics are typically covered in a full treatment of the lambda calculus, but students in L&C will not have seen them.

As our proof terms are going to get more complicated, we add an association list of global definitions, managed by functions such as `def`, which typechecks an annotated term and adds it to the list with a symbolic name. But substituting such a definition for the use of a name will result in expressions that require simplification through reduction. We have already coded the basic substitution mechanism. Reduction to weak normal head form suffices in some places, but strong reduction (full beta-reduction) is needed in others. Definitional equality is the alpha-equivalence of two strongly-reduced expressions.

All of this takes considerably more care to present in the classroom than I have taken here. A proper presentation takes students through the reasons for each additional requirement, by pointing out expressions that we would expect to typecheck but do not in an incomplete implementation. The code base is still under two hundred lines (an additional 10 lines for alpha equivalence, 30 for substitution, 30 for reduction and equivalence, 20 for support for definitions).

The reward for all this work is a surprisingly expressive language, considering its size. To start with, we can implement Church encodings. The Boolean type is implemented by its eliminator, namely if-then-else, which makes it easy to implement other Boolean functions.

```
(def 'bool '((∀ (x : Type) -> (x -> (x -> x))) : Type))
(def 'true '((λ x => (λ y => (λ z => y))) : bool))
(def 'false '((λ x => (λ y => (λ z => z))) : bool))
(def 'band  '((λ x => (λ y => ((x bool) (((y bool) true) false) false)))
     : (bool -> (bool -> bool))))
```

We can implement logical AND ($\wedge$), and show that it commutes. Logical OR ($\vee$) is also possible.

```
(def 'and '((λ p => (λ q => (∀ (c : Type) -> ((p -> (q -> c)) -> c))))
             : (Type -> (Type -> Type))))
(def 'conj '((λ p => (λ q => (λ x => (λ y => (λ c => (λ f => ((f x) y)))))))
             : (∀ (p : Type) -> (∀ (q : Type) -> (p -> (q -> ((and p) q)))))))
(def 'proj1 '((λ p => (λ q => (λ a => ((a p) (λ x => (λ y => x))))))
              : (∀ (p : Type) -> (∀ (q : Type) -> (((and p) q) -> p)))))
(def 'proj2 '((λ p => (λ q => (λ a => ((a q) (λ x => (λ y => y))))))
              : (∀ (p : Type) -> (∀ (q : Type) -> (((and p) q) -> q)))))
(def 'and-commutes
  '((λ p => (λ q => (λ a => ((((conj q) p) (((proj2 p) q) a)) (((proj1 p) q) a)))))
     : (∀ (p : Type) -> (∀ (q : Type) -> (((and p) q) -> ((and q) p))))))
```

Arithmetic can be implemented using Church numerals.

```
(def 'nat '((∀ (x : Type) -> (x -> ((x -> x) -> x))) : Type))
(def 'z '((λ x => (λ zf => (λ sf => zf))) : nat))
(def 's '((λ n => (λ x => (λ zf => (λ sf => (sf (((n x) zf) sf)))))) : (nat -> nat)))
(def 'one '((s z) : nat))
(def 'two '((s (s z)) : nat))
(def 'plus '((λ x => (λ y => (((x nat) y) s))) : (nat -> (nat -> nat))))
```

We can check that `((plus one) one)` and `two` are definitionally equivalent, but we have no way of expressing this in our term language. The next step is to add equality. From this point on, algorithmic descriptions are shorter and clearer than proof rules.

The proof term `(eq-refl t)` will have type `(t = t)`. The type `(t = w)` will have a proof term exactly when `t` and `w` are definitionally equal. The elimination form `eq-elim` implements the principle that "equals may be substituted for equals". It is applied to five things: a term `t` of type `T`, a "property" `P` that has type `(T -> Type)`, a term `pt` of type `(P t)` (that is, a proof that `t` has property `P`), a term `w` of type `T`, and a term `peq` of type `(t = w)`. The application of `eq-elim` has type `(P w)` (that is, it proves that `w` has property `P`), and the reduction rule produces the result of reducing `pt`. All this takes more space to describe in English than does the implementation. Here are some proofs.

```
(def 'one-eq-one '((eq-refl one) : (one = one)))
(def 'one-plus-one-is-two '((eq-refl two) : (((plus one) one) = two)))
(def 'eq-symm
  '((λ x => (λ y => (λ p => (eq-elim x (λ w => (w = x)) (eq-refl x) y p))))
    : (∀ (x : Type) -> (∀ (y : Type) -> ((x = y) -> (y = x))))))
(def 'eq-trans
  '((λ x => (λ y => (λ z => (λ p => (λ q => (eq-elim y (λ w => (x = w)) p z q))))))
    : (∀ (x : Type) -> (∀ (y : Type) -> (∀ (z : Type) ->
                          ((x = y) -> ((y = z) -> (x = z))))))))
```

The Church numerals are inefficient and awkward. Furthermore, even if we add back $\bot$ (which we should do, to facilitate expressing logical negation), we can state but cannot prove $\neg(0 = 1)$. To address these issues, we implement Peano numbers, with type `Nat`, zero `Z` and successor function `S`. What should the elimination form be? Computationally, we use natural numbers via structural recursion, with a `Z` case and a function to be applied to the predecessor in the `S` case. We could implement this with a recursor `nat-rec`, whose type would be $\forall(T : Type) \to T \to (T \to T) \to Nat \to T$. The reduction rules would reduce `(nat-rec ZCase SCase 0)` to `ZCase` and would reduce `(nat-rec ZCase SCase)S n))` to `(SCase (nat-rec ZCase SCase n))`.

But we can generalize T to be dependent, having it be a property P indexed by a Nat. The type becomes $\forall(P : Nat \to Type) \to (P\ Z) \to (\forall(k : Nat) \to (P\ k \to P\ (S\ k))) \to (\forall(k : Nat) \to P\ k)$. This is familiar: it is the induction principle for natural numbers, which we can call `nat-ind`. If P is the constant function that produces T, we recover the recursor, and the reduction rules for `nat-ind` are the obvious generalization of those for `nat-rec`.

```
(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))))
```

That last definition demonstrates the capability we promised at the start of this section, and we can also prove $\neg(0 = 1)$, or more generally, $\forall(n : Nat) \rightarrow \neg(0 = S\ n)$. To fulfil the promise of the title of the section, we need to add the existential quantifier. We use the notation $\exists(x : T) \rightarrow W$ for the dependent sum type often called Σ in the literature (again, see the title of [2]). The introduction form `∃-intro` tuples a type T, a "witness" a of type T, and a value pa of type $W[x \mapsto a]$ (that is, a proof that a has property W). The elimination form `∃-elim` is applied to a type V, a function f of type $\forall(x : T) \rightarrow (W \rightarrow V)$, and a value b of type $\exists(x : T) \rightarrow W$. The witness a embedded in b will be used as the argument to f to produce a value of type V, which is the type of the application of the elimination form. The associated reduction rule reduces `(∃-elim V f (∃-intro T a pa))` to `(f a pa)`.

As was the situation in the previous section, we have implemented intuitionistic logic, and so there are some theorems of classical logic that we cannot prove. For example, we can prove the theorem $\neg(\exists(x : A) \rightarrow B) \rightarrow (\forall(x : A) \rightarrow \neg B)$, but $\neg(\forall(x : A) \rightarrow B) \rightarrow (\exists(x : A) \rightarrow \neg B)$ eludes us. By adding the law of the excluded middle, suitably quantified, as an antecedent, we can prove the latter.

Why stop here? The code base is still under three hundred lines, and there are some easy additions that make Proust more friendly. The encodings of $\wedge$ and $\vee$ are not inefficient, but they are awkward to use, and it is simple to add them as we did in section 2. We could add lists as a datatype without much difficulty. Racket supports various UI improvements. But many technical improvements significantly complicate matters: a proper treatment of holes in the context of dependent types; dependent pattern matching to simplify the use of eliminators; a general mechanism for user-defined recursive datatypes; a tactics language to avoid large explicit proof terms.

This is a natural transition point from Proust. Students are ready to move to a system like Agda or Coq. Agda uses explicit proof terms (eased by pattern matching and a nice Emacs interface); Coq hides proof terms, though they can be exposed and even written explicitly as needed. Proust's treatment of holes, quantifiers, typechecking, and induction principles are in line with these more sophisticated systems. We have fulfilled our twin goals of demystification of full-featured proof assistants and properly motivated/situated exposure to logic for computer science students.

4 Curriculum

The previous two sections focussed on the development of Proust as it might be presented to students, but a course using it will intersperse attention to related topics. A nonstandard treatment of a subject must take care to demonstrate connections to more traditional approaches, and that will be a major theme in this section.

The main point of the syntactic notion of proof is to be able to draw conclusions about the semantic notion of truth. It is standard to write $\Gamma \vdash T$ if there is a proof of T using Γ (in our notation, if there exists t such that $\Gamma \vdash t : T$). Students will be familiar with Boolean values and functions from programming, and have a tendency to confuse proof and truth, so it is best to introduce early the formal definition of a valuation for variables, the value of a formula with respect to a valuation, and the notation $\Gamma \models T$ if every valuation that makes the formulas in Γ true also makes T true.

One can then discuss the notion of soundness ($\Gamma \vdash T$ always implies $\Gamma \models T$) and completeness (the converse). For a CS student, soundness is the more useful property; completeness is interesting more from a metamathematical viewpoint. Our logics are sound, but they are not complete. It is worth mentioning the existence of alternate models for which our logics are complete (Heyting algebras, Kripke models), and perhaps demonstrating the use of these for unprovability results.

My experience of requiring students to write natural deduction proofs on paper is that they are frustrated by the inflexibility of the formal system relative to their previous experiences with informal proofs and what they know of Boolean algebra. While a buggy program may also frustrate them, they are more inclined to see it as something worth fixing. With Proust, an incorrect proof is a buggy program, and this combined with immediate feedback on correctness puts proof construction into familiar territory. Students should still see a couple of proofs presented as trees built from rules in the standard fashion, and the explicit correspondence with proof terms.

In the bidirectional system we use, an immediate application of a lambda cannot be typechecked; proofs must be normal forms. But students may know from earlier courses that immediate application of lambda is the easiest way to get local definitions (as in the Racket macro implementation of the `let` construct). Proofs in propositional logic are usually not complicated enough to require local definitions, and if necessary, either the definition mechanism of section 3 can be implemented earlier, or a specific local binding construct can be added to the term language.

The treatment of propositional logic in section 2, apart from the use of Proust, is close to a traditional approach using natural deduction. This is not the case for predicate logic as described in section 3, and students will benefit from a sketch of first-order logic, notably the rules and axioms needed to add equality and arithmetic, and the corresponding completeness and incompleteness results.

It is not clear how much time there will be, in a one-semester treatment, for students to spend much time with Agda or Coq. At the least, a demonstration should be included at the end, and perhaps a short one at the beginning to increase motivation.

Finally, students will benefit from an understanding of the historical development of this material. It is best incorporated in short vignettes through the term, rather than crammed into a history lecture full of names and dates.

The first offering of this version of L&C will be a special enriched section that I will offer in the Fall 2016 term, with the possibility of a regular section in Spring 2017. The material could be offered at the introductory graduate level, at an increased pace that permits the inclusion of additional material (System F, Gödel's System T) or more exposure to Agda and/or Coq. The section on predicate logic could be adapted as an introductory or parallel tutorial for a graduate course that makes heavy use of these proof assistants. The advantage of Proust over the tutorials cited earlier is primarily due to the low overhead

of Racket and an S-expression representation of proofs and logical statements, the ease of minor uses of mutation (counters, hash tables), and the advantages of the DrRacket IDE. It is possible that use of a dynamically-typed language to implement typechecking may avoid student confusion between levels of abstraction.

5 Acknowledgments

I would like to thank Stephanie Weirich for her prompt and helpful answers to my questions, and the Recurse Center in New York City for hosting me as a resident in October 2015, during which time I workshopped some of this material.

References

[1] (2016): *The Agda wiki*. Available at `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[2] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh & Nicholas Oury (2010): *ΠΣ:Dependent Types without the Sugar*. In: *Functional and Logic Programming*, *Lecture Notes in Computer Science* 6009, Springer, pp. 40–55.

[3] Lennart Augustsson (2007): *Simpler, Easier!* Available at `http://augustss.blogspot.ru/2007/10/simpler-easier-in-recent-paper-simply.html`.

[4] Andrej Bauer (2012): *How to Implement Dependent Type Theory*. Available at `http://math.andrej.com/2012/11/08/how-to-implement-dependent-type-theory-i/`.

[5] (2016): *The Coq proof assistant*. Available at `https://coq.inria.fr`.

[6] Thierry Coquand (1996): *An algorithm for type-checking dependent types*. *Science of Computer Programming* 26(1-3), pp. 167–177.

[7] Jean-Yves Girard (1972): *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de doctorat d'état, Université Paris 7.

[8] Andres Löh, Conor McBride & Wouter Swierstra (2001): *A tutorial implementation of a dependently typed lambda calculus*. *Fundamenta Informaticae* XXI.

[9] (2012): *Racket*. Available at `http://www.racket-lang.org`.

[10] M.H. Sørensen & P. Urzyczyn (2006): *Lectures on the Curry-Howard Isomorphism*. *Studies in Logic and the Foundations of Mathematics* 149, Elsevier.

[11] Stephanie Weirich (2014): *Lectures at the Oregon Programming Languages Summer School: Types, Logic, Semantics, and Verification*. Available at `https://www.cs.uoregon.edu/research/summerschool/summer14/index.html`.