Crafting Provers in Racket

Adrian Manea
Coordinator: Assoc. Prof. Traian Şerbănuţă

March 20, 2020

Contents

TO ADD/CLARIFY

INTRODUCTION AND MOTIVATION

The main motivation for starting the work on this project is my interest in the programming language Racket. This grew from me getting acquainted with Emacs and Lisps, from a user standpoint at first, but then my interest grew and I started reading more about lambda calculus and various other related subjects. Before long, I discovered John McCarthy's pioneering work in trying to make lambda calculus suitable for programming languages, but what really captured my interest were the theoretical foundations that McCarthy's articles [McC60, McC61, McC62] set for this task, in a time when programming was either electrical engineering or pure mathematics. His work was said to have introduced a paradigm shift in computer science that is comparable to the non-Euclidean geometry revolution. The next step was discovering [AS96], which showed me how an apparently simple programming language such as Scheme can be used for wonderful constructions and various degrees of abstraction. This is further supported by the thorough specification and revisions that were published by the Scheme Community, the latest being [S+13].

Further reading in the world of Lisp dialects, I have discovered Racket, whose appeal was instantaneous to me, since it is so often described not as a general purpose programming language derived from Scheme,[1]but rather as a toolbox for constructing languages to solve various problems. In fact, as I see it, it offers the necessary items for one to properly understand, craft and teach languages that exhibit particular behaviour.

This is precisely the aim of the current work. Having a background in mathematics, I quickly became interested on the one hand in proof assistants (or so-called "theorem provers") and type theory, on the other. The appeal of category theory mixed with (or, by some sources, morphed into) topos theory, type theory and the background of intuitionistic logic is very strong for me and I have been trying to approach the subject from many of its sides. Of equal interest for me is the teaching aspect. I believe in strong foundations, a thorough understanding of fundamentals before getting into more complex structures and I so often search for methods, tools and examples that I can use to showcase particular aspects of the subject I'm learning or teaching. Racket, for

[1]Racket used to be called PLT Scheme, where PLT is the name of the research group that has been working on this project since the very beginning, scattered throughout the world, as showcased at [plt20].

me, provides the perfect environment to fulfil most of such intentions. On the one hand, the language is big and flexible enough to be expressive for concepts of type theory, logic and proofs and on the other, it can be used in a piecemeal setup to serve as a great teaching toolbox for my purposes.

The plan of the work is as follows. We will start with some basic elements of type theory and intuitionistic logic. While the subject is hugely developed and used in many parts of (theoretical) computer science, we will try to make the work as self-contained as possible and as such, include in the preliminaries strictly the topics that will be developed further. For simplicity, but also for historical reasons, we will be focusing on the main work by one of the creators of the field, at least in terms of making its presentation appropriate for computer science applications, Per Martin-Löf, [ML80]. While Martin-Löf's lectures focus more on the theoretical side of things, the excellent [NP90] provides enough details to see how the concepts can be used in actual programming.

The second preliminary part of the dissertation will focus on Racket itself. We will try to provide a so-called "crash-course" to introduce some elements of syntax. Again, we will be focusing on items that will be used throughout the dissertation.

A first, excellent example will follow, along the lines of P. Ragde's article, [Rag16]. There, the author showcases a toy proof assistant, called *Proust*, that they have written in Racket, the purpose being twofold. On the one hand, it shows the power that the language has for such tasks and on the other, as the author mentions, it helps them and their students to understand at least a part of the inner workings of well established proof assistants, such as Coq and Agda. That is, they will be implementing a small portion of the tools that are available in Coq and Agda in an intentionally verbose style so that their functions are transparent.

Finally, further examples are provided, which will exhibit features of interest from type theory and intuitionistic logic.

change bib style to go well with websites

CHAPTER 1 MARTIN-LÖF TYPE THEORY

We start the theoretical part of this work focusing on Martin-Löf's take on intuitionistic logic and type theory. Before we do that, however, a quick note on the historical evolution of the subject. For brevity, we will use the abbreviation MLTT for Martin-Löf's Type Theory.

1.1 Brief Historical Overview

Type theory was first developed by Bertrand Russell and some of his collaborators and then expanded by Frank Ramsey and others. It was used first to introduce a kind of hierarchy of (mathematical) concepts that would "solve" Russell's paradox. At the same time, in the first part of the twentieth century, the Dutch mathematician L. E. J. Brouwer, then continuing with the American mathematician E. Bishop sowed the seeds of *constructive mathematics*. They put the emphasis on proofs that actually produce examples of the concepts existential quantifiers speak of. Thus, for example, any proof of a proposition of the form $\exists x$ such that P must contain a method of actually instantiating that x. This approach contrasted with the formalism developed by leading mathematicians such as D. Hilbert who proposed that mathematics should be performed simply following abstract rules and that showing a method to obtain an x in the previous example does not necessarily mean that that x should be obtained "practically", but only *in principle*.

At the same time, philosopher and logician A. Heyting summarized in his monograph [Hey66] the approach that became known as *intuitionistic logic*. This emphasized constructive proofs as well and was used by Brouwer as a formal basis for his mathematical program.

In the second half of the twentieth century, the Swedish mathematician and philosopher Per Martin-Löf exposed his take on intuitionistic logic and type theory into what became known as *Martin-Löf type theory*. This approach draws influences from the philosophical work of F. Brentano, G. Frege and E. Husserl, but also uses the *Curry-Howard correspondence* between propositions and programs, terms and proofs (excellently detailed in [SU06]).

Martin-Löf's approach became extremely influential and with the help of works such as [NP90], it was quickly implemented as a foundation for extremely powerful proof assistants such as NuPRL, Coq, Agda, Idris and others.

It is also worth mentioning that type theory and intuitionistic logic have continued to develop independently from applications in computer science. As such, types and toposes are seen as good candidates to provide "proper" foundations of mathematics instead of sets. Some very important contributions have come from the HoTT group ([Gro]).

We will not go into more historical or philosophical details of this subject and instead further focus on the mathematical aspects that were then implemented in proof assistants.

1.2 Fundamentals of MLTT

MLTT draws inspiration from Gentzen natural deduction system from the 1930s, explained in more modern terms in [Gir90]. As such, judgments will be written in the so-called *proof tree form*, such as:

$$\frac{A}{A \vee B}$$

where above the line we have the hypotheses and below the inferences. In particular, the above rule takes for granted that we have some formulas A and B and it infers that $A \vee B$ is true given the hypothesis that A is true.

Given some A, which can be a set or a proposition, we write $a \in A$ to mean, using the Curry-Howard correspondence, that either:

- a is an element of the set A;
- a is a proof of the proposition A.

Also, the judgment $a = b \in A$ means more than meets the eye:

- A is a proposition or a set;
- a and b are proofs or elements respectively;
- a and b are identical elements of the set A or represent identical proofs of the proposition A.

In fact, more than these readings can be given for the simple judgment $a \in A$, as shown in the table of figure 1.1.

In particular, the reading which refers to problem-solving is commonly attributed to A. Kolmogorov ([Kol32]) and called the *Brouwer-Heyting-Kolmogorov interpretation.*

What is characteristic of MLTT is that in order to ascertain that we have a set (proposition), we must prescribe an *introduction* rule, by means of showing how a *canonical* element of the set

A set	$a \in A$	implies
A is a set	a is an element of the set A	A is nonempty
A is a proposition	a is a (constructive) proof of A	A is true
A is an intention (expectation)	a is a method of fulfilling A	A is fulfillable (realizable)
A is a problem (task)	a is a method of solving A	A is solvable

Figure 1.1: Readings of the judgment $a \in A$, cf. [ML80, p. 4]

(proof of the proposition, respectively) is constructed an *equality* rule which shows how we know that two canonical elements are equal.

For example, for the set of positive integers (using the Peano approach and denoting by a' the successor of the element a), we can give the rules:

- for the canonical elements: $0 \in \mathbb{N}$ and $\dfrac{a \in \mathbb{N}}{a' \in \mathbb{N}}$;
- equality of canonical elements: $0 = 0 \in \mathbb{N}$ and $\dfrac{a = b \in \mathbb{N}}{a' = b' \in \mathbb{N}}$.

Another example, for the product of two sets (propositions) $A \times B$, we have:

- canonical elements:
$$\frac{a \in A \qquad b \in B}{(a, b) \in A \times B};$$
- equality of canonical elements:
$$\frac{a = c \in A \qquad b = d \in B}{(a, b) = (c, d) \in A \times B}$$

Now, equality of the sets (propositions) as a whole is prescribed by showing how equal and canonical elements are formed for the sets (propositions). For example, for sets (propositions), the equality is simply:
$$\frac{\dfrac{a = b \in A}{}}{a = b \in B}.$$
For non-canonical elements, to explain what it means for them to be elements of a set (proposition) A, we must specify a method (proof, program) which, when executed (performed), it yields a *canonical* element of the set as a result.

Then, finally, two arbitrary (not necessarily canonical) elements of a set A are equal if, when executed as above, yield equal *canonical* elements of the set.

It is now worth focusing our attention on the particular cases of propositions. Given the setup above, we can write the table in figure 1.2.

We can be more precise than that, noting further:

a proof of	consists of
$\bot$	—
$A \wedge B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B
$A \supset B$	a method taking any proof of A into a proof of B
$(\forall x)B(x)$	a method taking any individual a into a proof of $B(a)$
$(\exists x)B(x)$	an individual a and a proof of $B(a)$

Figure 1.2: Proofs of propositions, cf. [ML80, p. 7]

- the proposition $\bot$ has no possible proof;
- (a, b) is a proof of $A \wedge B$, provided that a is a proof of A and b is a proof of B;
- $i(a)$ or $j(b)$ are proofs of $A \vee B$, provided that a is a proof of A and b is a proof of B, where i and j are the canonical inclusions (which we detail later);
- $\lambda x.b(x)$ is a proof of $A \supset B$, provided that $b(a)$ is a proof of B under the hypothesis that a is a proof of A;
- $\lambda x.b(x)$ is a proof of $(\forall x)B(x)$, provided that $b(a)$ is a proof of $B(a)$, where a is some individual;
- (a, b) is a proof of $(\exists x)B(x)$, given that a is an individual and b is a proof of $B(a)$.

Also, the presentation can be extended further to *types*. From a historical, philosophical and mathematical point of view, types themselves are taken as primary notions, hence not properly defined. Intuitively though, one can think of types as "hierarchies", "levels" of certain kinds of elements or rather use the computer science intuition of *data types*.

For this case, the judgment "a is an element of type A" is commonly denoted by $a : A$ and if this is the case, we say that the type A is *inhabited*.

It can be formally shown that the type-theoretical approach is isomorphic to the propositional approach and the set-theoretical approach, making what is commonly known as the *formulas-as-types* or *propositions-as-sets* interpretations (for intuitionistic logic).

Programming in MLTT!

dependent types

Dependent types references

- Rijke, Introduction to HoTT:
  - L1: Dependent Type Theory;
  - L2: Dependent Function Types and $\mathbb{N}$;

- L3: Inductive Types and the Universe;
- L4: Identity Types (just to make the connection with homotopy);

- The HoTT Book:
  - 1.1: Type Theory vs. Set Theory;
  - 1.2: Function Types;
  - 1.3: Universes and Families;
  - 1.4: Dependent Function Types (Π)
- McBride — The View from Left: §2: Dependent type theory for functional programming;
- PMLTT:
  - III.19: Types;
  - III.20: Defining sets in terms of types;
  - IV.21: Some small examples;

CHAPTER 2 RACKET CRASH COURSE

skip it and explain directly on code?

Historically speaking, Racket is based on Scheme, being formerly called PLT Scheme. As such, it expands on the standards R5RS (1998) and R6RS (2009) of Scheme and diverge more significantly from R7RS (2013). However, we will not be concerned about the differences between the languages and we will assume that all Racket syntax that we introduce is valid Scheme syntax and vice versa, unless explicitly mentioned otherwise.

Both languages emerged from a common ancestor, Lisp and as such, they use a *list syntax*. Furthermore, they use a prefix notation for the functions, predicates and mathematical operations, the so-called Polish (Łukasiewicz) notation.

The only delimiters Scheme uses are parentheses (and double quotations for strings) whereas Racket strongly suggests the use of square brackets inside parentheses for delimiting more important statements[1]

Loosely speaking, both Scheme and Racket are untyped, but they do recognize three basic types: *lists, functions* and *symbols*, the latter encompassing variables and "everything else".

A specific feature is the so-called *quoting mechanism*, denoted by a single quote (`'`) or a backtick (`` ` ``) which turns anything into a symbol and the reverse, the *unquoting mechanism*, denoted by a comma (`,`) which enforces evaluation, as in the case of a call by name versus call by value approach. For example, `'(+ 1 2)` is interpreted as the symbol (string) `(+ 1 2)`, whereas `,(+ 1 2)` means `3`.

For simple expressions, quoting can be done either with a single quote or with a backtick. However, in more complex expressions, a difference appears. The backtick is actually called *quasiquote* and it can be used in expressions containing unquote. So basically we have:

```
'(+ 1 2)                    ;; => (+ 1 2)
```

[1]Some Scheme implementations allow the use of square brackets as well, but do not enforce it and furthermore, there are interpreters that reject this syntax.

```
(define x 3)
`(+ 2 ,x)                ;; => 5     (used quasiquote)
'(+ 2 ,x)                ;; error    (used quote)
```

Moreover, there is also the *splice unquote* syntax, which is used to unquote lists. Instead of returning the whole list, as a regular unquote would do, splice unquote, denoted by `,@`, actually inserts the elements of the list:

```
(define x '(1 2 3))
`(+ 4 ,@x)               ;; => 10
```

Notice also the use of the quote in the first line, because we are just defining a (literal) list to store in `x`, whereas in the second line, we used the quasiquote, since the expression contains an unquote (the spliced one).

Another aspect of syntax is that a semicolon is used for comments and the convention is to use a single semicolon for an inline comment and two or more semicolons for comments spanning multiple lines. The output of a program is commonly written as `;; => output` inside the source code or documentation.

Remark 2.1: A short word on implementation: all the examples that we will be showcasing, as well as the included source code is tested on a Manjaro Linux operating system using the Emacs 26 editor, the included Scheme mode and the third party Racket mode.

After writing the source code, `C-c` loads the file into the respective REPL.

Some simple examples follow.

```
(+ (* 5 3) 1)        ;; => 16

(define (mod2 x)
  (lambda (x) (rem x 2)))
(mod2 5)             ;; => 1

(define (mod3 x)
  "Write the remainder of x when divided by 3"
  (cond                                     ; multiple branching
     ((equal (rem x 3) 1) write "it's 1")   ; if (x % 3 == 1)
     ((equal (rem x 3) 2) write "it's 2")   ; if (x % 3 == 2)
     (#t write "it's 0")))                  ; default (true) case

(define (add-or-quote x)
  "Add 3 if x is even or write 'hello' else"
  (cond
    ((equal (mod2 x) 0) (lambda (x) ,(+ 1 2)))
    (#t (lambda (x) 'hello))))
```

```
(set sum '(+ 1 2 3))         ; defines the variable "sum"
(set x (* ,sum 2))            ; defines x to be 12
```

Hopefully, the rest of the syntax can be understood directly from the examples that will follow. For further investigation, we recommend the official Racket documentation [rac20] and the book [FF14]. As the need requires, we will also further explain the syntax.

CHAPTER 3 PROUST

This chapter will provide an example of a great use of Racket to craft a “nano proof assistant”, called Proust. The presentation closely follows [Rag16] and it will contain further clarifications as needed.

As the author mentions, the article is intended as a DIY approach for a simple proof assistant (in fact, the name derives from a weird contraction of the expression “proof assistant”). In fact, the goal is much more interesting than that, in that it will also delve into the inner works of the underlying mechanisms of proof asistants, for the purpose of implementing them in a simpler manner.

Note that since Racket is rather a toolbox for crafting one’s own toy languages, any source code must start with a `# lang` pragma, which mentions what part of Racket one wants to use. Common pragmas (formally, *modules*) include:

- `htdp` – the special module with syntax used in the [FF14] book;
- `racket` – the full-fledged module with all syntax;
- `racket/base` – the simplest submodule of `racket`;
- `racket/typed` – the module of Racket with strong typing;
- `scheme` – the backwards compatible module allowing one to use Scheme syntax.

To not overcomplicate the example and to skip any decision process, we will just use `#lang racket`.

3.1 The Grammar and Basic Parsing

First, we specify the language that we will use to make proof terms, which will be a mix of untyped lambda calculus and simple and function types. As such, the BNF grammar of the language is as follows:

$$\begin{aligned}\texttt{expr} ::&= (\lambda x \Rightarrow \texttt{expr})\\ &\mid (\texttt{expr}\ \texttt{expr})\\ &\mid (\texttt{expr} : \texttt{type})\\ &\mid x\\ \texttt{type} ::&= (\texttt{type} \longrightarrow \texttt{type})\\ &\mid X.\end{aligned}$$

The reader will recognize, respectively: lambda abstraction, application, type inhabitance and free variables. These are the legal expressions and the types are either simple or functional.

To recognize the structures that appear, we will use the standard Racket structures (records), which will also allow pattern-matching to extract the parts that are needed:

```
(struct Lam (var body) #:transparent)         ; lambda abstractions
(struct App (func arg) #:transparent)         ; application
(struct Ann (expr type) #:transparent)        ; explicit type annotations
(struct Arrow (domain range) #:transparent)   ; functional types
```

A little remark about *transparent* vs. *opaque* structures. By default, all structures defined in Racket are opaque. What this means for our purposes is that if you print such a structure, it doesn't show anything about its internal contents. For example, printing a `Lam` will output `#<Lam>` (see the code examples that follow below). On the other hand, a transparent structure shows its internals when printed, i.e. it will print something like `(Lam 'x 'y)`.

By default, all Racket structures are opaque, so transparency must be enabled explicitly.

The first goal of the simple proof assistant will be to parse expressions, in order to understand them appropriately and extract the needed parts. For this, we define a function `parse-expr`, which takes a so-called S-expression (standard Racket/Lisp expression) and produces an element that is a legal `expr`.

```
(define (parse-expr s)
  (match s                            ; pattern-match s
    [`(lambda ,(? symbol? x) => ,e)   ; is it a lambda expression?
        (Lam x (parse-expr e))]       ; make it a Lam
    [`(,e0 ,e1)                       ; is it an application?
        (App (parse-expr e0)          ; make it an App
             (parse-expr e1))]
    [`(,e : ,t)                       ; is it a type annotation (e : t)?
        (Ann (parse-expr e)           ; make it an Ann
         (parse-expr t))]
    [`(,e1 ,e2 ,e3 ,r ...)              ; is it a general list?
        (parse-expr
            `((,e1 ,e2) ,e3 ,@r))]      ; parse it by parts
    [(? symbol? x) x]                   ; else, it's a symbol, return it
    [else (error 'parse "bad syntax: ~a" s)]))
```

Notice the very clever use of the quoting and unquoting (including splice) mechanism in the definition of `parse-expr`, as well as the square bracket delimitation of the matching cases. The quote for the patterns ensures that we are searching for expressions that are either (literally) `(lambda ...)` or `(... ...)`, but inside the quoting we actually want to see what's there, so we evaluate the components using the unquote. The special predicate `(? symbol? x)` is used only in pattern matching, where the first question mark matches anything (similar to `*` in regular expressions). Also note that there is a convention in Lisps to name predicates (i.e. functions that return true or false) ending with a question mark.

Most of the syntax should be clear and it is also accompanied by comments which should ease understanding. The only new piece of syntax which we comment on is the *ellipse* (`...`), which is used to match anything that follows ("the rest").

Similarly, we parse type expressions:

```
(define (parse-type t)
  (match t
    [`(,t1 -> ,t2)                              ; is it a functional type?
        (Arrow (parse-type t1)                  ; make it an Arrow
               (parse-type t2))]
    [`(,t1 -> ,t2 -> ,r ...)                    ; maybe it's a multi-arg function
        (Arrow (parse-type t1)
               (parse-type `(,t1 -> ,@r)))]
    [(? symbol? X) X]                           ; is it a simple type? return it
    [else (error "can't parse this type")]))
```

To make printing clear enough, we define helper functions that do "unparsing", both for expressions and for types:

```
;; pretty-print-expr : expr -> String
(define (pretty-print-expr e)
  (match e
    [(Lam x b) (format "(lambda ~a => ~a)" x (pretty-print-expr b))]
    [(App e1 e2) (format "(~a ~a)"
                         (pretty-print-expr e1)
                         (pretty-print-expr e2))]
    [(? symbol? x) (format "~a" x)]
    [(Ann e t) (format "(~a : ~a)"
                        (pretty-print-expr e)
```

```
                                 (pretty-print-expr t))]))

;; pretty-print-type : Type -> String
(define (pretty-print-type t)
  (match t
    [(Arrow t1 t2) (format "(~a -> ~a)"
                           (pretty-print-type t1)
                           (pretty-print-type t2))]
    [else t]))
```

When evaluating and checking proofs, we will need a *context*, which is defined as a `(listof (List Symbol Type))`. The `listof` keyword is called a *contract* in this case and in short, it *asserts* that what it expects is a list with a symbol and a type. If the context does not respect the contract, we get a specific error. Racket provides extensive support for software contracts, for various items such as structures, functions, variables, but we do not actually need any such complexity here, so we will only indicate [Com] for further information.

evaluation context theory!

Now, given the fact that a context is a list of a symbol and a type, we can `pretty-print` it as well using the function:

```
(define (pretty-print-context ctx)
  (cond
    [(empty? ctx) ""]
    [else (string-append (format "\n~a : a"
                                 (first (first ctx))
                                 (pretty-print-type (second (first ctx))))
          (pretty-print-context (rest ctx)))]))
```

3.2 Checking Lambdas

Now, given this setup, we can actually start writing the functions for the proofs. **We remark explicitly that for the purposes of this chapter, we will only present the part that uses lambda terms for proofs.**

First, type-checking, which checks whether an expression has a certain type in a given context.

```
;; type-check : Context Expr Type -> Boolean
;; produces true if expr has type t in context ctx
;; (else, error)
(define (type-check ctx expr type)
  (match expr
    [(Lam x t)              ; is expr a lambda expression?
```

```
      (match type           ; then the type must be an arrow
        [(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    ; fail for other types
            (cannot-check ctx expr type))]))

;; the error function
(define (cannot-check ctx expr type)
  (error 'type-check "cannot typecheck ~a as ~a in context:\n~a"
    (pretty-print-expr expr)
    (pretty-print-type type)
    (pretty-print-context ctx)))
```

Then the function that tries to make a type inference.

```
;; type-infer : Context Expr -> Type
;; tries to produce type of expr in context ctx
;; (else, error)
(define (type-infer ctx expr)
  (match expr
    [(Lam _ _) (cannot-infer ctx expr)]      ; lambdas are handled in type-check
    [(Ann e t) (type-check ctx e t) t]       ; check type annotations
    [(App f a)                               ; function application
        (define tf (type-infer ctx f))
          (match tf                          ; must be arrow type
            [(Arrow tt tw) #:when (type-check ctx a tt) tw] ; when the rest typechecks
            [else (cannot-infer ctx expr)])]
    [(? symbol? x)                           ; for symbols
       (cond
         [(assoc x ctx) => second]           ; if it's a list, the second is the type
         [else (cannot-infer ctx expr)])])) ; else, not okay

;; the error function
(define (cannot-infer ctx expr)
  (error 'type-infer "cannot infer type of ~a in context:\n~a"
    (pretty-print-expr expr)
    (pretty-print-context ctx)))
```

3.3 Basic Testing

Now we can define a function that checks some basic proofs like so:

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

This way, if errors do not appear, the function will successfully apply the `type-infer` procedure and return `true`, which can be seen as a "successful exit code".

Using this, we can use the Racket testing module to check some basic proofs, such as:

```
(require test-engine/racket-tests)             ; import the test module

;; check whether check-proof returns true => good proof
;; lambda xy.x : A -> (B -> A)
(check-expect
  (check-proof '((lambda x => (lambda y => x)) : (A -> (B -> A))))
    true)

;; lambda xy.yx : (A -> ((A -> B) -> B))
(check-expect
  (check-proof '((lambda x => (lambda y => (y x))) : (A -> ((A -> B) -> B))))
    true)

;; lambda fgx.f(gx) : ((B -> C) -> ((A -> B) -> (A -> C)))
(check-expect
  (check-proof '((lambda f => (lambda g => (lambda x => f (g x)))) :
                    ((B -> C -> ((A -> B) -> (A -> C))))))
    true)

(test)           ;; => All 3 tests passed.
```

comment on what the proofs mean

add some more, e.g. ifs, booleans, Church nums

BIBLIOGRAPHY

[AS96] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[Com] The Racket Community. Contracts. `https://docs.racket-lang.org/guide/contracts.html`. Accessed March 2020.

[FF14] Matthias Felleisen and Bruce Findler. *How to Design Programs.* MIT Press, 2014.

[Gir90] Jean-Yves Girard. *Proofs and Types.* Cambridge University Press, 1990.

[Gro] The HoTT Group. Homotopy type theory. `https://homotopytypetheory.org/`. Accessed March 2020.

[Hey66] Arend Heyting. *Intuitionism, an Introduction.* Dover, 1966.

[Kol32] Andrei Kolmogorov. Zur deutung der intuitionistichen logik. *Mathematische Zeitschrift*, 1932.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, April 1960.

[McC61] John McCarthy. A basis for a mathematical theory of computation. *Western Joint Computer Conference*, 1961.

[McC62] John McCarthy. Towards a mathematical science of computation. *IFIP-62*, 1962.

[ML80] Per Martin-Löf. Intuitionistic type theory. Lectures in Padua, 1980. Notes by Giovanni Sambin.

[NP90] Bengt Nordström and Kent Petersson. *Programming in Martin-Löf Type Theory*. Oxford University Press, 1990. Freely available at `http://www.cse.chalmers.se/research/group/logic/book/`.

[plt20] The PLT group. `https://racket-lang.org/people.html`, 2020. Accessed: March 2020.

[rac20] Racket. `https://racket-lang.org/`, 2020. Accessed: March 2020.

[Rag16] Prabhakar Ragde. Proust: A nano proof assistant. In Johan Jeuring and Jay McCarthy, editors, *Trends in Functional Programming in Education*, pages 63–75. arXiv/cs.PL, 2016.

[S^+13] Gerald Sussman et al. Revised7 Report on the Algorithmic Language Scheme. Technical report, Scheme Working Group, 7 2013.

[SU06] Morten Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier Science, 2006.