

Introduction

Much of the theory developed in the underlying course “Logic II” can be implemented in a “proof assistant”. In the present setting this is interesting, since we can then machine extract from a formal proof its computational content, in the form of a term t realizing the proven formula A (the “specification”). Since we can also formalize the proof of the soundness theorem, this approach will give us a formal verification that the program t satisfies the specification A .

If one takes a proof assistant “from the shelf”, there is a danger that the theory underlying it does not match the theory one wants to formalize. Moreover, in a fundamental matter like tracing the development of mathematical theories down to their logical grass roots, one should understand in detail how proofs are represented and generated. As an additional benefits, one can

- experiment with new concepts and proof ideas,
- get a helpful display of the proof structure as a lambda-term (Curry-Howard correspondence),
- display the computational content of a proof again as a lambda-term, and – last not least –
- machine check the correctness of proofs.

As an experiment, the present appendix tries to guide the reader through the first steps of development of such a proof assistant, away from the “user”-mentality.

CHAPTER 1

Scheme

The first decision to be taken is to select a programming language. We pick Scheme (see ?), which is a particularly simple variant of Lisp (for LISt Processing); the latter was invented by ?. A special feature of languages in the Lisp family is that procedures can be represented as data; this makes it easy to write programs that manipulate procedures.

1.1. Expressions; names and assignments

We begin with simple examples; note the prefix notation.

```
(+ 3 4) ==> 7
(- 27 19) ==> 8
(* 13 (+ 2 4)) ==> 78
```

It is possible to assign values to names, and use them as abbreviations:

```
(define size 2) ==> size
size ==> 2
(* 5 size) ==> 10
(+ (* 5 size) (* size size)) ==> 14
```

1.2. User defined procedures

Apart from the primitive procedures like + and * the user can define his own procedures

```
(define (square x) (* x x))
or equivalently
(define square (lambda (x) (* x x)))
```

Then

```
(square 3) ==> 9
(square (+ 2 5)) ==> 49
(square (square 3)) ==> 81
```

One can use square to define further procedures:

```
(define (sum-of-squares x y) (+ (square x) (square y)))
(sum-of-squares 3 4) ==> 25
```

1.3. Conditional expressions

To express case distinctions like

$$\text{abs}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

one can use

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

An alternative is

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

or

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Notice that `cond` and `if` are *not* procedures, but so-called special forms (they do not first evaluate all their arguments).

1.4. Recursion

The factorial function

$$\begin{aligned} 0! &= 1, \\ (n+1)! &= (n+1) \cdot n!. \end{aligned}$$

can be implemented as

```
(define (factorial n)
  (if (= 0 n)
      1
      (* n (factorial (- n 1)))))
```

An iterative form of this definition is

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
```

```

product
  (fact-iter (* counter product)
             (+ counter 1)
             max-count)))

```

In `product` the result is accumulated.

In an (unnested) tree recursion there can be many (unnested) calls to the function to be defined. An example is the Fibonacci function

$$\text{Fib}(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise.} \end{cases}$$

A direct translation gives

```

(define (fib n)
  (cond ((= 0 n) 0)
        ((= 1 n) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))

```

Notice that the resulting computation is rather inefficient, since it involves many recomputations of values. Hence the following iterative version is preferable.

```

(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= 0 count)
      b
      (fib-iter (+ a b) a (- count 1))))

```

One can also have higher order procedures like

```

(define (sum summand-fct next-fct a b)
  (if (> a b)
      0
      (+ (summand-fct a)
          (sum summand-fct next-fct (next-fct a) b))))

```

Notice that procedures are passed as arguments.

We will also need `do`-loops. For example, we can compute the least $n!$ larger than or equal to a given bound m by

```

(define (facbound m)
  (do ((n 0 (+ 1 n))
      (facn 1 (* (+ 1 n) facn)))
      ((<= m facn) facn)))

```

1.5. Pairs and lists

Pairing is the main tool to build data in Lisp. To form a pair of two arguments we use the primitive procedure `cons`. From a pair one can read off its components by the primitive procedures `car` and `cdr`. The names `car` and `cdr` come from the original implementation of Lisp (on an IBM 704 machine). `car` is for “contents of address register” and `cdr` is for “contents of decrement register”.

```
(define x (cons 1 2)) ==> x
(car x) ==> 1
(cdr x) ==> 2
```

Note that a pair is a usual data object, that can be given a name and manipulated.

```
(define x (cons 1 2)) ==> x
(define y (cons 3 4)) ==> y
(define z (cons x y)) ==> z
(car (car z)) ==> 1
(car (cdr z)) ==> 3
```

One can use `(caar z)` and `(cadr z)` to abbreviate the final two expressions.

Pairs are mainly used to form lists. Lists are represented by iterated pairing, where the `cdr`-field of the last pair remains empty. Notice that we need the empty list `()` as a new data object; the primitive procedure `null?` tests whether its argument is the empty list. For list processing the following procedures are helpful:

```
list?
list
length
append
reverse
list-ref
member
map
```

An example for the usage of `map` is

```
(define a (list 1 2 3 4)) ==> a
```

```

a                ==> (1 2 3 4)
(map square a)   ==> (1 4 9 16)

```

1.6. Symbols and the necessity of quoting

Suppose we want to have symbols as data objects, for instance

```

(a b c d)
((a 1) (b 7) (c 3))
(* (+ 1 2) (+ x 7))
(define (factorial n)
  (if (= 0 n) 1 (* n (factorial (- n 1)))))

```

To avoid searching for values of symbols we need quotation:

```

(define a 1) ==> a
(define b 2) ==> b
(list a b)   ==> (1 2)
(list 'a 'b) ==> (a b)
(list 'a b)  ==> (a 2)

(car '(a b c)) ==> a
(cdr '(a b c)) ==> (b c)

```

1.7. Exercises

EXERCISE 1.7.1. (a) Compute in Scheme:

$$50 \cdot 100 - 4999$$

$$50 \cdot (100 - 4999)$$

(b) Define the following functions in Scheme:

$$f(x, y) = x^2 \cdot y^2$$

$$s(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

(c) Define a function **compose** that takes two (unary, numerical) functions f and g and returns the new function $f \circ g$.

EXERCISE 1.7.2 (Recursion, Lists). (a) Recursively define (**pow** x y) to compute x^y for non-negative integer y , using multiplication and recursion.

(b) Define a function **maximum** that maps a (non-empty) list of numbers to their maximum.

(c) Define a function **rev** that takes a list and returns it with elements in reverse order.

EXERCISE 1.7.3 (Nested lists). Tree structures can naturally be represented in Scheme as nested lists.

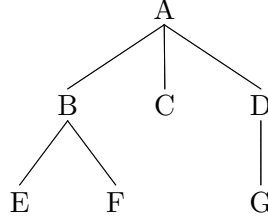


FIGURE 1. A labeled tree T of height 2, with associated nested list $L_T = '(A (B (E) (F)) (C) (D (G)))'$

For a labeled tree T inductively define its associated list L_T and its height $|T|$ as follows:

- If T is a single leaf with label x (x not being a list), then $L_T := '(x)'$ and $|T| := 0$.
- If otherwise T is a node with label x and with children $T_1, \dots, T_n$, then $L_T := '(x L_{T_1} \dots L_{T_n})'$ and $|T| := 1 + \max_{1 \leq k \leq n} |T_k|$.

Define a Scheme function that computes $|T|$, given a list L_T .

EXERCISE 1.7.4 (Untyped λ -calculus). Let $\mathcal{V}$ be a set of variable symbols. λ -terms are syntactic expressions, inductively defined by these clauses:

- Variables $x_1, x_2, \dots \in \mathcal{V}$ are λ -terms.
- For λ -terms M and N , $(M N)$ is a λ -term.
(the *application* of M to N)
- For a λ -term M and a variable x , $(\lambda_x M)$ is a λ -term.
(λ -*abstraction* of x : the function that maps x to $M(x)$)

We associate λ -terms with Scheme expressions as follows:

$$x = 'x \quad | \quad (M N) = '(M N) \quad | \quad (\lambda_x M) = '(\text{lambda } (x) M)$$

where M, N denote the expressions associated with M, N , respectively. As pool of variable symbols, use $\mathcal{V} := \{a - z, a1, a2, \dots\}$.

- Define a Scheme function (**is-lambda-term?** M) that checks whether the expression M represents a valid λ -term M .
- A variable occurrence of x in a term M is *free*, if it is not in the scope of a λ -abstraction of x . Define a function (**free-vars** M) that returns the list of variables that occur freely M .
- Define a function (**rename-var** M x y) that returns the result of replacing free occurrences of x by y in the λ -term M (α -conversion).

- (d) Define a function (`substitute M1 x M2`) that replaces free occurrences of x in $M1$ by $M2$, while avoiding variable capture. *Variable capture* occurs when $M2$ is substituted into a context where a variable symbol that occurs freely in $M2$ is already bounded by λ -abstraction. Then the free occurrences accidentally become bounded. To avoid this, the bounded variable should be renamed, using a fresh symbol. For example, when substituting x for y in the expression `'(lambda (x) (y x))`, the result should be `'(lambda (x1) (x x1))`, not `'(lambda (x) (x x))`.
- (e) To evaluate a λ -term M , one converts any instances of $((\lambda_x M_1) M_2)$ in M to $M_1[M_2/x]$, where the latter is the result of substituting M_2 for x in M_1 (β -conversion). The resulting term is said to be in *normal form*. Define a function (`eval-term M`) that recursively replaces instances of type `'(lambda (x) M1) M2` in M with (`substitute M1 x M2`).

REMARK. (i) Not every term in untyped lambda calculus has a normal form. Observe that, for example, (`eval-term '(lambda (x) (x x) (lambda (x) (x x)))`) does not terminate. By assigning types to λ -terms, one can restrict the term language to guarantee unique existence of normal forms (e.g. simply typed lambda calculus).

- (ii) Any recursive function can be represented within lambda calculus. Using the implementation of exercise 1.7.4, we can already perform arithmetic computations. Natural numbers are represented in λ -calculus as follows (Church numerals):

$$n := \lambda_f \lambda_x \underbrace{(f(f \dots (f x)))}_{n \text{ times}}$$

The Scheme-routines below generate and translate Church numerals:

```
(define (churchify n)
  (letrec ((nest-f (lambda (t n)
    (if (= 0 n) t (nest-f (list 'f t) (- n 1))))))
    (list 'lambda 'f (list 'lambda 'x (nest-f 'x n)))))
(define (unchurchify t)
  (eval (list (list t '(lambda (x) (+ 1 x))) 0)))
```

We can define and evaluate arithmetic expressions in λ -calculus:

```
(define plus '(lambda (m) (lambda (n)
  (lambda (f) (lambda (x) ((m f) ((n f) x)))))))
(define times '(lambda (m) (lambda (n)
  (lambda (f) (n (m f))))))
(define n (churchify 17))
(define m (churchify 42))
(define n+m (list (list plus n) m))
(unchurchify (eval-term n+m))           ;; => 59
```

```
(define n*m (list (list times n) m))
(unchurchify (eval-term n*m))      ;; => 714
```

EXERCISE 1.7.5 (Derivation checking). Derivations in implicational propositional logic correspond to (simply typed) λ -terms. Building on the previous exercise, we can implement simple proof checking for propositions with only $\rightarrow$ and $\perp$. Assume the following representation of implicational formulas in Scheme:

- Propositional variables $p, q, \dots$ are written as quoted symbols 'p, 'q, ... Falsum is written as 'bot.
- Implications $(A \rightarrow B)$ are represented as triples '(-> A B), where A, B are the Scheme representations of the formulas A, B .

For example, the formula $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$ is represented by '(-> (-> A (-> B C)) (-> (-> A B) (-> A C))).

We next inductively associate derivations with λ -terms:

$$\frac{[u: A] \quad \frac{}{A \rightarrow B} \rightarrow^+ u \quad \frac{}{B} \rightarrow^-}{\frac{}{A \rightarrow B} \rightarrow^+ u} = '(\text{lambda } (u) \text{ M}) \quad \frac{\frac{}{A \rightarrow B} \rightarrow^- \quad \frac{}{A} \rightarrow^-}{\frac{}{B} \rightarrow^-} = '(M \ N)$$

where M and N are the terms associated with M and N , respectively. For assumptions, set $[u: A] = 'u$.

For instance, the derivation

$$\frac{\frac{\frac{u: A \rightarrow B \rightarrow C}{B \rightarrow C} \rightarrow^- \quad \frac{v: A}{A} \rightarrow^-}{\frac{C}{A \rightarrow C} \rightarrow^+ v} \quad \frac{\frac{w: A \rightarrow B}{B} \rightarrow^- \quad \frac{v: A}{A} \rightarrow^-}{\frac{C}{A \rightarrow C} \rightarrow^+ v} \rightarrow^-}{\frac{(A \rightarrow B) \rightarrow A \rightarrow C \rightarrow^+ w}{(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C} \rightarrow^+ u}$$

is then associated with the term

```
'(lambda (u) (lambda (w) (lambda (v) ((u v) (w v))))))
```

To determine the formula derived by such a term, one needs to know which formulas the assumption variables stand for. The types of the variables determine the type of the λ -term. For each derivation term we maintain a list `context` to keep track of the formulas (types) assigned to assumption variables:

```
'((u (-> A (-> B C))) (v A) (w (-> A B)))
```

This requires that variable names are used non-conflictingly in the derivation term. Instances of the same formula may be assigned the same assumption label, but no label can be used for different formulas simultaneously.

A valid derivation is thus a pair of a λ -term and a context, satisfying the following conditions: (i) the context uniquely fixes the type of all assumption variables in the derivation term, and (ii) in cases of application $\lambda(M\ N)$, M is of implication type and the antecedent has the type of N .

- (a) Define a Scheme function (`is-formula? a`) that checks whether `a` represents a formula in minimal implicational logic.
- (b) Define a function (`is-valid-derivation? term context`) that returns `#t` whenever `term` is a valid derivation, given `context`.
- (c) Define a function (`infer-formula term context`) that returns the formula derived by `term`, given `context` and a function

(`is-derivation-of? term context formula`)

that checks whether `term` is a derivation of `formula`, given `context`.

EXERCISE 1.7.6 (Proof automation for minimal implicational logic). Building on the two preceding exercises, define a Scheme function

(`prove goal-formula context`)

that returns a derivation term for `goal-formula` given `context`, if one exists.

One can employ the following search strategy:

- (1) If `goal-formula` is part of `context`, return the corresponding assumption term.
- (2) If otherwise `goal-formula` is an implication, move the antecedent into the `context` (using abstraction) and attempt to prove the consequent.
- (3) Else try applying formulas from the context to each other and add the resulting new inferences to `context`. If new formulas can be added, recursively call `prove` with extended `context`, otherwise return `#f` – the formula was not derivable.