Decorating natural deduction

Helmut Schwichtenberg

Contents

Preface v

Introduction vii
- Basic minimal logic vii
- The Curry-Howard correspondence viii
- Extracting computational content ix

Overview xi

Chapter 1. Logic 1
- 1.1. Logic in Minlog: basic examples 1
- 1.2. Minimal logic 3
- 1.3. Conversion rules in the extended logic 5
- 1.4. Classical and intuitionistic logic 7

Chapter 2. The model of partial continuous functionals 15
- 2.1. Natural numbers 16
- 2.2. Lists 21
- 2.3. Binary trees 25

Chapter 3. Formulas as problems 29
- 3.1. Decorated predicates and formulas 30
- 3.2. Logical rules for the decorated connectives 33
- 3.3. Decorated axioms 34

Chapter 4. Computational content of proofs 35
- 4.1. Realizability 35
- 4.2. Computational content of classical proofs 41

Chapter 5. Decorating proofs 49
- 5.1. Decoration algorithm 49
- 5.2. Applications 53

Appendix A. Logic in Minlog: a tutorial 61
- A.1. Propositional logic 62
- A.2. Predicate logic 69

A.3. Automatic proof search 78

Appendix B. Constructive existence proofs 81
B.1. Quotient and remainder 81
B.2. List reversal 83
B.3. Linearity of the Brouwer-Kleene ordering 85

Appendix C. Useful commands 101
C.1. Emacs 101
C.2. Scheme 101
C.3. Minlog 101

Bibliography 109

Index 111

Preface

These are lecture notes for a course "Decorating natural deduction" at Dipartimento di Informatica, Università degli Studi di Verona, March 2016. It is mainly based on Schwichtenberg and Wainer (2012), but with a special emphasis on proof decoration and its use for the fine tuning of the computational content of proofs. To some extent these notes are also meant as a tutorial introduction to the Minlog proof assistant and its use for program extraction from proofs.

This text owes a lot to other people's work.

- My coauthor Stan Wainer has contributed in many aspects.
- Much of the material is due to my former student Ulrich Berger.
- The decoration algorithm and its applications have been studied by Diana Ratiu in her thesis (2011).
- I have made heavy use of the Minlog tutorial, originally written by Laura Crosilla and extended by Monika Seisenberger.

München, 7. März 2016
Helmut Schwichtenberg

Introduction

In this introduction we deal with the basics of formalizing proofs and, via the Curry-Howard correspondence, analysing their computational structure.

Basic minimal logic

Basic minimal logic is a system of rules for deriving logical formulas based just on the two symbols $\to$ (implication) and $\forall$ (for all). Each symbol has two rules: an introduction rule ($\to^+$, $\forall^+$) and an elimination rule ($\to^-$, $\forall^-$). The rules for implication are

$$\dfrac{\begin{array}{c}[A]\\ \mid M\\ B\end{array}}{A \to B}\,\to^+ \qquad \dfrac{\begin{array}{c}\mid M\\ A \to B\end{array}\qquad \begin{array}{c}\mid N\\ A\end{array}}{B}\,\to^-$$

and the rules for universal quantification are

$$\dfrac{\begin{array}{c}\mid M\\ A\end{array}}{\forall_x A}\,\forall^+ x \qquad \dfrac{\begin{array}{c}\mid M\\ \forall_x A(x)\end{array}\qquad r}{A(r)}\,\forall^-$$

These are Gentzen's (1935) Natural Deduction rules (and there are others for $\exists$, $\vee$ and $\wedge$ which we shall come to later). Gentzen's idea was that natural deduction rules do indeed reflect the ways in which we construct logical arguments.

Notice that subderivations of the premises of rules are labelled M, N. In order to avoid obvious invalid derivations (for example $Px \to \forall_x Px$), the rule $\forall^+ x$ with conclusion $\forall_x A$ is subject to the following *(eigen-)variable condition*: the derivation M of the premise A should not contain any open (undischarged) assumptions having x as a free variable.

It is clear that derivations need to be started off somewhere, so in addition we need to introduce assumptions and – as in the implication introduction – allow some assumptions to be closed or discharged in the course of a derivation. The notation for a discharged assumption A is $[A]$.

A simple example is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall_x(A\to Px)] \qquad x}{A\to Px}\,\forall^- \qquad A}{Px}\,\to^-}{\forall_x Px}\,\forall^+x}{A\to\forall_x Px}\,\to^+}{\forall_x(A\to Px)\to A\to\forall_x Px}\,\to^+$$

Note that the variable condition is satisfied: x is not free in A (and also not free in $\forall_x(A\to Px)$).

For an unnecessary detour via implication:

$$\dfrac{\dfrac{\begin{array}{c}[A]\\ \mid M\\ B\end{array}}{A\to B}\,\to^+ \qquad \begin{array}{c}\mid N\\ A\end{array}}{B}\,\to^- \qquad \text{reduces to} \qquad \begin{array}{c}\mid N\\ A\\ \mid M\\ B\end{array} \tag{0.1}$$

For an unnecessary detour via universal quantification:

$$\dfrac{\dfrac{\begin{array}{c}\mid M(x)\\ A(x)\end{array}}{\forall_x A(x)}\,\forall^+x \qquad r}{A(r)}\,\forall^- \qquad \text{reduces to} \qquad \begin{array}{c}\mid M(r)\\ A(r)\end{array} \tag{0.2}$$

The Curry-Howard correspondence

Clearly the tree structure of logical derivations of any complexity at all becomes quite cumbersome, and the availability of some alternative representation therefore becomes increasingly important, especially when we wish to operate on derivations. The Curry-Howard correspondence provides a neat, computationally inspired alternative. The underlying idea is that if we have a derivation $M(x)$ of $A(x)$ then any means of (universally) binding the x should then represent a derivation of $\forall_x A(x)$. The notation chosen for binding the x is $\lambda_x M(x)$, denoting the function $x \mapsto M(x)$. On the side of $\to$ a derivation M of B from some assumptions A, each of which must now in addition have a label u, is then represented as $\lambda_u M(u)$, denoting the function $u \mapsto M(u)$. This requires the labelling of assumptions so that all assumptions discharged by an application of $\to^+$ must have the same label. For example the unnecessary detour via $\to$ in example (0.1) will thus be represented as

$$(\lambda_u M(u))N \quad \text{reduces to} \quad M(N).$$

Similarly the unnecessary detour via $\forall$ in example (0.2) will be represented as

$$(\lambda_x M(x))r \quad \text{reduces to} \quad M(r).$$

These reductions are instances of what, in lambda calculus terms, is known as *beta reduction* and we write

$$\begin{aligned} (\lambda_u M(u))N &\quad \mapsto_\beta \quad M(N), \\ (\lambda_x M(x))r &\quad \mapsto_\beta \quad M(r). \end{aligned}$$

The lambda calculus provides an abstract setting for representing and computing functions and beta reduction is the main computational mechanism. Now there comes one further detail: we want to be able to move back and forth from derivations to lambda representations of them and back again from lambda terms to derivations. For this reason it is necessary to assign to each lambda term a "type", which will be the formula whose proof it represents. The formula type will be written as a superscript. In detail then, the first beta reduction example above now becomes

$$(\lambda_u M(u^A)^B)^{A \to B} N^A \quad \mapsto_\beta \quad M(N^A)^B.$$

In summary, the Curry-Howard correspondence is completely described by the Table 1 below.

Extracting computational content

Suppose that we now extend minimal logic with axioms introducing the existential quantifier:

$$\exists^+ \colon \forall_x (A \to \exists_x A), \qquad \exists^- \colon \exists_x A \to \forall_x (A \to B) \to B \quad (x \text{ not free in } B).$$

These now allow us to make computationally meaningful derivations. The underlying principle is this: suppose one has a derivation of a closed formula $\exists_x A(x)$ resulting from an existential introduction axiom $\exists^+$, i.e., the derivation (written as a Curry-Howard term) is of the form $\exists^+ r u^{A(r)}$. Then r (the computational content) is a witness for the existential quantifier, and it may be read off immediately. Of course the derivation may not end with an existential introduction. However, the process of normalization will beta-reduce the derivation term into one in which $\exists^+$ *is* the final operator to be applied. In general normalization is the process of computing out a lambda term, until no further beta reductions can be made. In other words, normalization reduces away all unnecessary detours.

Now suppose that the formula $\exists_x A(x)$ is not closed, say it has one free variable z. By instantiating z we obtain again a closed formula depending on the instantiated value. Extracting a witnessing term from a normalized

Derivation	Term
$u\colon A$	u^A
$$\dfrac{\begin{matrix}[u\colon A]\\ \mid M\\ B\end{matrix}}{A \to B}\,\to^+ u$$	$(\lambda_{u^A} M^B)^{A\to B}$
$$\dfrac{\begin{matrix}\mid M\\ A\to B\end{matrix}\qquad \begin{matrix}\mid N\\ A\end{matrix}}{B}\,\to^-$$	$(M^{A\to B}N^A)^B$
$$\dfrac{\begin{matrix}\mid M\\ A\end{matrix}}{\forall_x A}\,\forall^+ x$$ (with var.cond.)	$(\lambda_x M^A)^{\forall_x A}$ (with var.cond.)
$$\dfrac{\begin{matrix}\mid M\\ \forall_x A(x)\end{matrix}\qquad r}{A(r)}\,\forall^-$$	$(M^{\forall_x A(x)} r)^{A(r)}$

TABLE 1. Derivation terms for $\to$ and $\forall$

derivation term, as above, then provides a witness depending on the instantiated value. However, to bring out the uniformity involved in this process requires a new method, *realizability.*

These course notes present many fundamental concepts and principles underlying our proof assistant Minlog. For example, inductively defined data and predicates, recursion, induction and decoration. All will be developed within the Minlog setting, and in each case a wide variety of practical case studies will illustrate program extraction.

Overview

We begin with a short presentation of minimal logic in natural deduction style, including the Gödel-Gentzen embedding of classical and intuitionistic logic into minimal logic. Since these notes should also serve as tutorial introduction to the Minlog proof assistant, we conclude this chapter with some rather basic examples of how to use Minlog to generate formal proofs and represent them as Curry-Howard lambda terms. For the convenience of the reader, in Appendix A we have included the initial part of the Minlog tutorial written originally by Laura Crosilla.

In Chapter 2 we leave the realm of pure logic and introduce some simple (finitary) data types (or free algebras): natural numbers, lists and binary trees. It is discussed how one defines functions (called program constants in Minlog) by their defining equations, and how proofs by induction are done in Minlog. As an example, a proof by induction that every natural number n can be divided by $m+1$ with some quotient q and remainder r is presented in detail. Then we consider lists over an arbitrary type α, and give some examples to illustrate again the use of induction, this time an existence proof for list reversal. Finally we consider binary trees, and prove by an appropriate induction that the Brouwer-Kleene ordering is linear. Later (in 4.1) we come back to these proofs, and extract their computational content.

Chapter 3 develops Kolmogorov's idea of viewing a formula A as a computational problem, asking for a solution. Such a solution will make use of the data given by the assumptions in the formula A, i.e., the parameters of the problem. To be able to express dependence on and independence of such parameters we split each of our (only) logical connectives $\to, \forall$ into two variants, a "computational" one $\forall^{\mathrm{c}}, \to^{\mathrm{c}}$ and a "non-computational" one $\forall^{\mathrm{nc}}, \to^{\mathrm{nc}}$. One can view this "decoration" of $\to, \forall$ as turning our (minimal) logic into a "computational logic". We define when a formula is computationally relevant (c.r.) or not (n.c.), and refine this distinction by defining the type $\tau(A)$ of a formula A. We also introduce inductively defined predicates I either with or without computational content, and in the former case define the free algebra ι_I associated with (the clauses defining) I. Since the logical connectives $\exists$, $\wedge$ and $\vee$ can be viewed as inductively defined (by their standard introduction axioms recalled in Chapter 1), we can now introduce computational variants of all of them by the different choices of decorations for $\to$ and $\forall$ occurring in their introduction axioms (or clauses). We also adapt our logical rules to the decorated connectives $\to, \to^{\mathrm{nc}}$ and $\forall, \forall^{\mathrm{nc}}$.

In Chapter 4 the Brouwer-Heyting-Kolmogorov interpretation of proofs is introduced, and in particular the realizability interpretation. We assign to any derivation M of a formula A its extracted term $\mathrm{et}(M)$, which is viewed as the realizer of A provided by the proof M. Finally we state the soundness theorem, saying that this indeed is the case. Then we give some examples of program extraction from some of the constructive proofs discussed in Chapter 2: quotient-and-remainder, list reversal and linearity of the Brouwer-Kleene ordering. Finally we address the problem how to extract (possibly hidden) computational content in classical proofs. This can indeed be done: we present a refined form of the so-called A-translation of Friedman (1978) and Dragalin (1979), and as an example consider list reversal again, this time by means of a weak (or classical) existence proof, which turns out to be similar to the one presented in Chapter 2. Its computational content is a quadratic algorithm. We come back to this proof in Chapter 5, where we prove that the decoration algorithm introduced there transforms it into another one with linear complexity.

In the final Chapter 5 we are interested in fine-tuning the computational content of proofs, by inserting decorations. We present an algorithm to find a (unique) "optimal" decoration of a given proof, and consider some applications: list reversal, computing the Fibonacci numbers in continuation passing style, and finally the Maximal Scoring Segment (MSS) algorithm. For instance in the latter case, directly deriving such an algorithm from a proof leads to quadratic complexity. We will see that the (automatically found) optimal decoration of this proof results in a linear extracted algorithm.

CHAPTER 1

Logic

The main subject of Mathematical Logic is mathematical proof. We now set out the system of minimal logic, and extend it by adding disjunction, conjunction and existential quantification. Thus the language consists of all those formulas built up from a countable list of propositional and predicate symbols $(P, Q, R\dots)$ by application of the following rules: (i) a prime formula is one of the form $Pt_1 \dots t_k$ where $t_1, \dots, t_k$ are *terms* constructed from variables or constants by application of given function symbols; (ii) if A and B are formulas the so are $A \to B$, $A \vee B$, $A \wedge B$, $\forall_x A$ and $\exists_x A$. The set of variables $x, y \dots$ which occur free in a formula A (i.e., not bound by a quantifier) is denoted $\mathrm{FV}(A)$.

Notice that negation is not yet properly present, but $\neg A$ is defined by $A \to \bot$, where $\bot$ is, as yet, just an arbitray propositional symbol. Later it will get its intended meaning "falsity" when we move to intuitionistic logic by adding the ex-falso-quodlibet scheme $\bot \to A$.

1.1. Logic in Minlog: basic examples

As a first encounter with the Minlog proof assistant we consider two simple logical facts. Let A, B, C be propositional variables.

$$(A \to B \to C) \to (A \to B) \to A \to C.$$

Informal proof. Assume $A \to B \to C$. To show: $(A \to B) \to A \to C$. So assume $A \to B$. To show: $A \to C$. So finally assume A. To show: C. Using the third assumption twice we have $B \to C$ by the first assumption, and B by the second assumption. From $B \to C$ and B we then obtain C. Then $A \to C$, cancelling the assumption on A; $(A \to B) \to A \to C$ cancelling the second assumption; and the result follows by cancelling the first assumption. □

For the second example involving quantifiers, let P be a unary predicate variable.

$$\forall_x (A \to Px) \to A \to \forall_x Px.$$

Informal proof. Assume $\forall_x (A \to Px)$. To show: $A \to \forall_x Px$. So assume A. To show: $\forall_x Px$. Let x be arbitrary; note that we have not made any assumptions on x. To show: Px. We have $A \to Px$ by the first assumption. Hence also Px by the second assumption. Hence $\forall_x Px$. Hence $A \to \forall_x Px$, cancelling the second assumption. Hence the result, cancelling the first assumption. $\square$

We now give derivations of the two example formulas treated informally above. Since in many cases the rule used is determined by the conclusion, we suppress in such cases the name of the rule.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{u\colon A\to B\to C \qquad w\colon A}{B\to C} \qquad \cfrac{v\colon A\to B \qquad w\colon A}{B}}{C}}{A\to C}\,{\to^+}w}{(A\to B)\to A\to C}\,{\to^+}v}{(A\to B\to C)\to(A\to B)\to A\to C}\,{\to^+}u$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{u\colon \forall_x(A\to Px) \qquad x}{A\to Px} \qquad v\colon A}{Px}}{\forall_x Px}\,\forall^+ x}{A\to\forall_x Px}\,{\to^+}v}{\forall_x(A\to Px)\to A\to\forall_x Px}\,{\to^+}u$$

Note that the variable condition is satisfied: x is not free in A (and also not free in $\forall_x(A \to Px)$).

Let us now formalize these proofs in Minlog. We describe the interactions rather shortly; for a more thorough introduction the reader should consult the Minlog tutorial, whose initial part is reproduced in Appendix A. After starting the system by typing

```
(load "~/git/minlog/init.scm")
```

we declare three propositional variables by executing

```
(add-pvar-name "A" "B" "C" (make-arity))
```

The proof then is generated by the following sequence of commands:

```
(set-goal "(A -> B -> C) -> (A -> B) -> A -> C")
(assume "u" "v" "w")
(use "u")
(use "w")
(use "v")
(use "w")
```

We save the proof and display it as lambda-expression, with formulas assigned to the assumption variables:

```
(define proof (current-proof))
(proof-to-expr-with-formulas proof)
```

The result is

```
u73: A -> B -> C
v74: A -> B
w75: A

(lambda (u73)
  (lambda (v74) (lambda (w75) ((u73 w75) (v74 w75)))))
```

For the second example involving quantifiers we proceed similarly. We declare x as a variable of type α (a type variable) and P as a unary predicate variable

```
(add-var-name "x" (py "alpha"))
(add-pvar-name "P" (make-arity (py "alpha")))
```

The proof is generated by the following sequence of commands:

```
(set-goal "all x(A -> P x) -> A -> all x P x")
(assume "u" "v" "x")
(use "u")
(use "v")
```

We again save the proof and display it as lambda-expression, with formulas assigned to the assumption variables:

```
(define proof (current-proof))
(proof-to-expr-with-formulas proof)
```

The result is

```
u80: all x(A -> P x)
v81: A

(lambda (u80) (lambda (v81) (lambda (x) ((u80 x) v81))))
```

These lambda-expression are exactly what the Curry-Howard correspondence gives us.

1.2. Minimal logic

Departing from Gentzen we introduce disjunction, conjunction and existential quantification by means of axioms rather than rules. As Curry-Howard derivation terms they appear as constants. It is often notationally convenient to write the "type" of a derivation term (i.e., its formula) after a colon instead of as a superscript, thus $u\colon A$ instead of u^A. We will use both of these notations, as convenient.

For disjunction the introduction and elimination axioms are

$$\vee_0^+\colon A \to A \vee B,$$
$$\vee_1^+\colon B \to A \vee B,$$

$$\vee^-\colon A \vee B \to (A \to C) \to (B \to C) \to C.$$

Note the important convention that implication associates to the right, e.g., $A \to B \to C \to D$ means $A \to (B \to (C \to D))$. For conjunction we have

$$\wedge^+\colon A \to B \to A \wedge B, \qquad \wedge^-\colon A \wedge B \to (A \to B \to C) \to C$$

and for the existential quantifier

$$\exists^+\colon A \to \exists_x A, \qquad \exists^-\colon \exists_x A \to \forall_x (A \to B) \to B \quad (x \notin \mathrm{FV}(B)).$$

Remark. All these axioms can be seen as special cases of a general schema, that of an *inductively defined predicate* I, given by some introduction axioms (also called clauses) I_i^+. It is understood as the *least* predicate satifying these rules; this is expressed by an elimination (or least-fixed-point) axiom I^-. Later we will study this kind of definition in full generality.

Gentzen's original introduction and elimination rules for $\vee$, $\wedge$ and $\exists$ are

$$\frac{\begin{array}{c}\mid M\\ A\end{array}}{A \vee B}\,\vee_0^+ \qquad \frac{\begin{array}{c}\mid M\\ B\end{array}}{A \vee B}\,\vee_1^+ \qquad \frac{\begin{array}{ccc} & [u\colon A] & [v\colon B]\\ \mid M & \mid N & \mid K\\ A \vee B & C & C\end{array}}{C}\,\vee^- u, v$$

For conjunction we have

$$\frac{\begin{array}{cc}\mid M & \mid N\\ A & B\end{array}}{A \wedge B}\,\wedge^+ \qquad \frac{\begin{array}{cc} & [u\colon A] \quad [v\colon B]\\ \mid M & \mid N\\ A \wedge B & C\end{array}}{C}\,\wedge^- u, v$$

and for the existential quantifier

$$\frac{\begin{array}{cc} & \mid M\\ r & A(r)\end{array}}{\exists_x A(x)}\,\exists^+ \qquad \frac{\begin{array}{cc} & [u\colon A]\\ \mid M & \mid N\\ \exists_x A & B\end{array}}{B}\,\exists^- x, u \text{ (var.cond.)}$$

Similar to $\forall^+ x$ the rule $\exists^- x, u$ is subject to an *(eigen-)variable condition*: in the derivation N the variable x (i) should not occur free in the formula of any open assumption other than $u\colon A$, and (ii) should not occur free in B.

These rules are easily derivable form the axioms (and conversely). For example to derive the elimination rule for $\vee$ take the given derivation of $A \vee B$, apply $\to^+$ to the derivation of C from A, and to the derivation of C from B. Using the $\vee^-$ axiom, three applications of $\to^-$ yield C. To derive the existence elimination rule take the given derivation of $\exists_x A$, apply $\to^+$ to the derivation of B from A and then $\forall^+$ (noting the eigenvariable condition) to yield $\forall_x(A \to B)$. Then using the $\exists^-$ axiom, two applications of $\to^-$ yield B.

Examples. We prove

$$\neg\neg\forall_x A \to \forall_x \neg\neg A, \tag{1.1}$$

$$\exists_x(A \to B) \to A \to \exists_x B \quad \text{with } x \notin \mathrm{FV}(A). \tag{1.2}$$

Proof of (1.1).

$$\dfrac{\dfrac{\dfrac{\dfrac{u\colon \neg\neg\forall_x A \qquad \dfrac{\dfrac{v\colon \neg A \qquad \dfrac{w\colon \forall_x A \qquad x}{A}}{\bot}}{\neg\forall_x A}\to^+ w}{\bot}}{\neg\neg A}\to^+ v}{\forall_x\neg\neg A}\forall^+ x}{\neg\neg\forall_x A \to \forall_x\neg\neg A}\to^+ u$$

Proof or (1.2).

$$\dfrac{\dfrac{\dfrac{u\colon \exists_x(A \to B) \qquad \dfrac{x \qquad \dfrac{w\colon A \to B \qquad v\colon A}{B}}{\exists_x B}\exists^+}{\exists_x B}\exists^- x, w}{A \to \exists_x B}\to^+ v}{\exists_x(A \to B) \to A \to \exists_x B}\to^+ u$$

The reader should note that the variable conditions of $\forall^+$ in (1.1) and $\exists^-$ in (1.2) are satisfied. Note also that the use of labelled assumptions provides an explicit check on the stage in a derivation at which a given assumption gets discharged. Therefore the square bracket signifying a closed-off assumption is no longer necessary.

1.3. Conversion rules in the extended logic

In addition to the $\to, \forall$-conversions, we need

$\vee$-conversion.

$$\dfrac{\dfrac{\begin{array}{c}\mid M\\ A\end{array}}{A \vee B}\vee_0^+ \qquad \begin{array}{c}[u\colon A]\\ \mid N\\ C\end{array} \qquad \begin{array}{c}[v\colon B]\\ \mid K\\ C\end{array}}{C}\vee^- u, v \qquad \mapsto \qquad \begin{array}{c}\mid M\\ A\\ \mid N\\ C\end{array}$$

∧-conversion.

$$\dfrac{\dfrac{\begin{matrix}\mid M\\ A\end{matrix}\qquad\begin{matrix}\mid N\\ B\end{matrix}}{A\wedge B}\,\wedge^+\qquad \begin{matrix}[u\colon A]\quad[v\colon B]\\ \mid K\\ C\end{matrix}}{C}\,\wedge^-u,v\quad\mapsto\quad \begin{matrix}\mid M & & \mid N\\ A & & B\\ & \mid K & \\ & C & \end{matrix}$$

∃-conversion.

$$\dfrac{\dfrac{r\qquad\begin{matrix}\mid M\\ A(r)\end{matrix}}{\exists_xA(x)}\,\exists^+\qquad \begin{matrix}[u\colon A(x)]\\ \mid N\\ B\end{matrix}}{B}\,\exists^-x,u\quad\mapsto\quad \begin{matrix}\mid M\\ A(r)\\ \mid N'\\ B\end{matrix}$$

The corresponding conversions will then be

$$\exists^-(\exists^+rM)(\lambda_{x,u}N(x,u))\mapsto_\beta N(r,M).$$

However, there is a difficulty here: an introduced formula may be used as a minor premise of an application of an elimination rule for ∨, ∧ or ∃, then stay the same throughout a sequence of applications of these rules, being eliminated at the end. This also constitutes a local maximum, which we should like to eliminate; *permutative conversions* are designed for exactly this situation. In a permutative conversion we permute an E-rule upwards over the minor premises of $\vee^-$, $\wedge^-$ or $\exists^-$. They are defined as follows.

∨-permutative conversion.

$$\dfrac{\dfrac{\begin{matrix}\mid M\\ A\vee B\end{matrix}\qquad\begin{matrix}\mid N\\ C\end{matrix}\qquad\begin{matrix}\mid K\\ C\end{matrix}}{C}\qquad\begin{matrix}\mid L\\ C'\end{matrix}}{D}\,\text{E-rule}\quad\mapsto$$

$$\dfrac{\begin{matrix}\mid M\\ A\vee B\end{matrix}\qquad\dfrac{\begin{matrix}\mid N\\ C\end{matrix}\qquad\begin{matrix}\mid L\\ C'\end{matrix}}{D}\,\text{E-rule}\qquad\dfrac{\begin{matrix}\mid K\\ C\end{matrix}\qquad\begin{matrix}\mid L\\ C'\end{matrix}}{D}\,\text{E-rule}}{D}$$

∧-permutative conversion.

$$\dfrac{\dfrac{\begin{matrix}\mid M\\ A\wedge B\end{matrix}\qquad\begin{matrix}\mid N\\ C\end{matrix}}{C}\qquad\begin{matrix}\mid K\\ C'\end{matrix}}{D}\,\text{E-rule}\quad\mapsto$$

$$\dfrac{\begin{matrix}\mid M\\ A\wedge B\end{matrix}\qquad\dfrac{\begin{matrix}\mid N\\ C\end{matrix}\qquad\begin{matrix}\mid K\\ C'\end{matrix}}{D}\,\text{E-rule}}{D}$$

$\exists$-permutative conversion.

$$\dfrac{\dfrac{\begin{matrix}\mid M\\ \exists_x A\end{matrix} \qquad \begin{matrix}\mid N\\ B\end{matrix}}{B} \qquad \begin{matrix}\mid K\\ C\end{matrix}}{D}\,\text{E-rule} \qquad \mapsto$$

$$\dfrac{\begin{matrix}\mid M\\ \exists_x A\end{matrix} \qquad \dfrac{\begin{matrix}\mid N\\ B\end{matrix}\qquad\begin{matrix}\mid K\\ C\end{matrix}}{D}\,\text{E-rule}}{D}$$

For a detailed treatment of these conversions we refer the reader to Schwichtenberg and Wainer (2012, Section 1.2).

1.4. Classical and intuitionistic logic

We distinguish between two kinds of "exists" and two kinds of "or": the "weak" or classical ones and the "strong" or non-classical ones, with constructive content. In the present context both kinds occur together and hence we must mark the distinction; we shall do this by writing a tilde above the weak disjunction and existence symbols thus

$$A \tilde{\vee} B := \neg A \to \neg B \to \bot, \qquad \tilde{\exists}_x A := \neg\forall_x \neg A.$$

These weak variants of disjunction and the existential quantifier are no stronger than the proper ones (in fact, they are weaker):

$$A \vee B \to A \tilde{\vee} B, \qquad \exists_x A \to \tilde{\exists}_x A.$$

This can be seen easily by putting $C := \bot$ in $\vee^-$ and $B := \bot$ in $\exists^-$.

Remark. Since $\tilde{\exists}_x \tilde{\exists}_y A$ unfolds into a rather awkward formula we extend the $\tilde{\exists}$-terminology to lists of variables:

$$\tilde{\exists}_{x_1,\dots,x_n} A := \forall_{x_1,\dots,x_n}(A \to \bot) \to \bot.$$

Moreover let

$$\tilde{\exists}_{x_1,\dots,x_n}(A_1 \tilde{\wedge} \dots \tilde{\wedge} A_m) := \forall_{x_1,\dots,x_n}(A_1 \to \dots \to A_m \to \bot) \to \bot.$$

This allows to stay in the $\to, \forall$ part of the language. Notice that $\tilde{\wedge}$ only makes sense in this context, i.e., in connection with $\tilde{\exists}$.

In the definition of derivability in falsity $\bot$ plays no role. We may change this and require *ex-falso-quodlibet* axioms, of the form

$$\forall_{\vec{x}}(\bot \to R\vec{x}\,)$$

with R a relation symbol distinct from $\bot$. Let Efq denote the set of all such axioms. A formula A is called *intuitionistically derivable*, written $\vdash_i A$, if $\mathrm{Efq} \vdash A$. We write $\Gamma \vdash_i B$ for $\Gamma \cup \mathrm{Efq} \vdash B$.

We may even go further and require *stability* axioms, of the form

$$\forall_{\vec{x}}(\neg\neg R\vec{x} \to R\vec{x}\,)$$

with R again a relation symbol distinct from $\bot$. Let Stab denote the set of all these axioms. A formula A is called *classically derivable*, written $\vdash_c A$, if $\mathrm{Stab} \vdash A$. We write $\Gamma \vdash_c B$ for $\Gamma \cup \mathrm{Stab} \vdash B$.

It is easy to see that intuitionistically (i.e., from Efq) we can derive $\bot \to A$ for an *arbitrary* formula A, using the introduction rules for the connectives. A similar generalization of the stability axioms is only possible for formulas in the language not involving $\vee, \exists$. However, it is still possible to use the substitutes $\tilde{\vee}$ and $\tilde{\exists}$.

Theorem (Stability, or principle of indirect proof).

(a) $\vdash (\neg\neg A \to A) \to (\neg\neg B \to B) \to \neg\neg(A \wedge B) \to A \wedge B$.
(b) $\vdash (\neg\neg B \to B) \to \neg\neg(A \to B) \to A \to B$.
(c) $\vdash (\neg\neg A \to A) \to \neg\neg\forall_x A \to A$.
(d) $\vdash_c \neg\neg A \to A$ *for every formula A without* $\vee, \exists$.

Proof. (a) is left as an exercise.

(b) For simplicity, in the derivation to be constructed we leave out applications of $\to^+$ at the end.

$$\dfrac{u\colon \neg\neg B \to B \qquad \dfrac{\dfrac{v\colon \neg\neg(A \to B) \qquad \dfrac{\dfrac{u_1\colon \neg B \qquad \dfrac{u_2\colon A \to B \qquad w\colon A}{B}}{\bot}}{\neg(A \to B)}\to^+ u_2}{\bot}}{\neg\neg B}\to^+ u_1}{B}$$

(c)

$$\dfrac{u\colon \neg\neg A \to A \qquad \dfrac{\dfrac{v\colon \neg\neg\forall_x A \qquad \dfrac{\dfrac{u_1\colon \neg A \qquad \dfrac{u_2\colon \forall_x A \qquad x}{A}}{\bot}}{\neg\forall_x A}\to^+ u_2}{\bot}}{\neg\neg A}\to^+ u_1}{A}$$

(d) Induction on A. The case $R\vec{t}$ with R distinct from $\bot$ is given by Stab. In the case $\bot$ the desired derivation is

$$\dfrac{v\colon (\bot\to\bot)\to\bot \qquad \dfrac{u\colon \bot}{\bot\to\bot}\,\to^+ u}{\bot}$$

In the cases $A\wedge B$, $A\to B$ and $\forall_x A$ use (a), (b) and (c), respectively. $\square$

Using stability we can prove some well-known facts about the interaction of weak disjunction and the weak existential quantifier with implication. We first prove a more refined claim, stating to what extent we need to go beyond minimal logic.

Lemma. *The following are derivable.*

$$(\tilde\exists_x A\to B)\to\forall_x(A\to B) \quad \textit{if } x\notin \mathrm{FV}(B), \tag{1.3}$$

$$(\neg\neg B\to B)\to \quad \forall_x(A\to B)\to\tilde\exists_x A\to B \quad \textit{if } x\notin \mathrm{FV}(B), \tag{1.4}$$

$$(\bot\to B[x{:=}c])\to(A\to\tilde\exists_x B)\to\tilde\exists_x(A\to B) \quad \textit{if } x\notin \mathrm{FV}(A), \tag{1.5}$$

$$\tilde\exists_x(A\to B)\to A\to\tilde\exists_x B \quad \textit{if } x\notin \mathrm{FV}(A). \tag{1.6}$$

The last two items can also be seen as simplifying a weakly existentially quantified implication whose premise does not contain the quantified variable. In case the conclusion does not contain the quantified variable we have

$$(\neg\neg B\to B)\to \quad \tilde\exists_x(A\to B)\to\forall_x A\to B \quad \textit{if } x\notin \mathrm{FV}(B), \tag{1.7}$$

$$\forall_x(\neg\neg A\to A)\to(\forall_x A\to B)\to\tilde\exists_x(A\to B) \quad \textit{if } x\notin \mathrm{FV}(B). \tag{1.8}$$

Proof. (1.3)

$$\dfrac{\tilde\exists_x A\to B \qquad \dfrac{\dfrac{\dfrac{u_1\colon \forall_x\neg A \qquad x}{\neg A} \qquad A}{\bot}}{\neg\forall_x\neg A}\,\to^+ u_1}{B}$$

(1.4)

$$\dfrac{\neg\neg B\to B \qquad \dfrac{\dfrac{\neg\forall_x\neg A \qquad \dfrac{\dfrac{\dfrac{u_2\colon \neg B \qquad \dfrac{\dfrac{\forall_x(A\to B) \qquad x}{A\to B} \qquad u_1\colon A}{B}}{\bot}}{\neg A}\,\to^+ u_1}{\forall_x\neg A}}{\bot}}{\neg\neg B}\,\to^+ u_2}{B}$$

(1.5) Writing B_0 for $B[x{:=}c]$ we have

$$\dfrac{\dfrac{\forall_x\neg(A\to B)\quad c}{\neg(A\to B_0)}\qquad \dfrac{\dfrac{\perp\to B_0\qquad \dfrac{\dfrac{A\to\tilde\exists_x B\quad u_2\colon A}{\tilde\exists_x B}\qquad \dfrac{\dfrac{\dfrac{\dfrac{\forall_x\neg(A\to B)\quad x}{\neg(A\to B)}\qquad \dfrac{u_1\colon B}{A\to B}}{\perp}}{\neg B}\to^+u_1}{\forall_x\neg B}}{\perp}}{B_0}}{A\to B_0}\to^+u_2}{\perp}$$

(1.6)

$$\dfrac{\tilde\exists_x(A\to B)\qquad \dfrac{\dfrac{\dfrac{\dfrac{\forall_x\neg B\quad x}{\neg B}\qquad \dfrac{u_1\colon A\to B\quad A}{B}}{\perp}}{\neg(A\to B)}\to^+u_1}{\forall_x\neg(A\to B)}}{\perp}$$

(1.7)

$$\dfrac{\neg\neg B\to B\qquad \dfrac{\dfrac{\tilde\exists_x(A\to B)\qquad \dfrac{\dfrac{\dfrac{u_2\colon\neg B\qquad \dfrac{u_1\colon A\to B\qquad \dfrac{\forall_x A\quad x}{A}}{B}}{\perp}}{\neg(A\to B)}\to^+u_1}{\forall_x\neg(A\to B)}}{\perp}}{\neg\neg B}\to^+u_2}{B}$$

(1.8) We derive $\forall_x(\perp\to A)\to(\forall_x A\to B)\to\forall_x\neg(A\to B)\to\neg\neg A$. Writing Ax, Ay for $A(x), A(y)$ we have

$$\dfrac{\dfrac{\dfrac{\forall_x\neg(Ax\to B)\quad x}{\neg(Ax\to B)}\qquad \dfrac{\dfrac{\forall_x Ax\to B\qquad \dfrac{\dfrac{\dfrac{\dfrac{\forall_y(\perp\to Ay)\quad y}{\perp\to Ay}\qquad \dfrac{u_1\colon\neg Ax\quad u_2\colon Ax}{\perp}}{Ay}}{\forall_y Ay}}{B}}{Ax\to B}\to^+u_2}{\perp}}{\neg\neg Ax}\to^+u_1$$

Using this derivation M we obtain

$$\dfrac{\dfrac{\forall_x\neg(Ax\to B)\quad x}{\neg(Ax\to B)}\qquad \dfrac{\dfrac{\forall_x Ax\to B\qquad \dfrac{\dfrac{\dfrac{\dfrac{\forall_x(\neg\neg Ax\to Ax)\quad x}{\neg\neg Ax\to Ax}\qquad \begin{array}{c}\mid M\\ \neg\neg Ax\end{array}}{Ax}}{\forall_x Ax}}{}}{B}}{Ax\to B}}{\bot}$$

Since clearly $\vdash (\neg\neg A\to A)\to\bot\to A$ the claim follows. $\square$

Remark. An immediate consequence of (1.8) is the classical derivability of the "drinker formula" $\tilde{\exists}_x(Px\to\forall_x Px)$, to be read "in every non-empty bar there is a person such that, if this person drinks, then everybody drinks". To see this let $A := Px$ and $B := \forall_x Px$ in (1.8).

Corollary.

$\vdash_c (\tilde{\exists}_x A\to B)\leftrightarrow\forall_x(A\to B)$ *if* $x\notin \mathrm{FV}(B)$ *and* B *without* $\vee,\exists$,

$\vdash_i (A\to\tilde{\exists}_x B)\leftrightarrow\tilde{\exists}_x(A\to B)$ *if* $x\notin\mathrm{FV}(A)$,

$\vdash_c \tilde{\exists}_x(A\to B)\leftrightarrow(\forall_x A\to B)$ *if* $x\notin\mathrm{FV}(B)$ *and* A, B *without* $\vee,\exists$.

There is a similar lemma on weak disjunction:

Lemma. *The following are derivable.*

$$\begin{aligned}
&& (A\mathbin{\tilde{\vee}}B\to C)&\to(A\to C)\wedge(B\to C),\\
(\neg\neg C\to C)&\to & (A\to C)&\to(B\to C)\to A\mathbin{\tilde{\vee}}B\to C,\\
(\bot\to B)&\to & (A\to B\mathbin{\tilde{\vee}}C)&\to(A\to B)\mathbin{\tilde{\vee}}(A\to C),\\
&& (A\to B)\mathbin{\tilde{\vee}}(A\to C)&\to A\to B\mathbin{\tilde{\vee}}C,\\
(\neg\neg C\to C)&\to & (A\to C)\mathbin{\tilde{\vee}}(B\to C)&\to A\to B\to C,\\
(\bot\to C)&\to & (A\to B\to C)&\to(A\to C)\mathbin{\tilde{\vee}}(B\to C).
\end{aligned}$$

Proof. The derivation of the final formula is

$$\dfrac{\neg(B\to C)\qquad \dfrac{\dfrac{\bot\to C\qquad \dfrac{\neg(A\to C)\qquad \dfrac{\dfrac{\dfrac{A\to B\to C\quad u_1\colon A}{B\to C}\qquad u_2\colon B}{C}}{A\to C}\to^+u_1}{\bot}}{C}}{B\to C}\to^+u_2}{\bot}$$

The other derivations are similar to the ones above, if one views $\tilde{\exists}$ as an infinitary version of $\tilde{\vee}$. $\square$

Corollary.

$$\vdash_c (A \mathbin{\tilde\vee} B \to C) \leftrightarrow (A \to C) \wedge (B \to C) \quad \textit{for } C \textit{ without } \vee, \exists,$$
$$\vdash_i (A \to B \mathbin{\tilde\vee} C) \leftrightarrow (A \to B) \mathbin{\tilde\vee} (A \to C),$$
$$\vdash_c (A \to C) \mathbin{\tilde\vee} (B \to C) \leftrightarrow (A \to B \to C) \quad \textit{for } C \textit{ without } \vee, \exists.$$

Remark. It is easy to see that weak disjunction and the weak existential quantifier satisfy the same axioms as the strong variants, if one restricts the conclusion of the elimination axioms to formulas without $\vee, \exists$. In fact, we have

$$\vdash A \to A \mathbin{\tilde\vee} B, \quad \vdash B \to A \mathbin{\tilde\vee} B,$$
$$\vdash_c A \mathbin{\tilde\vee} B \to (A \to C) \to (B \to C) \to C \quad (C \text{ without } \vee, \exists),$$
$$\vdash A \to \tilde\exists_x A,$$
$$\vdash_c \tilde\exists_x A \to \forall_x (A \to B) \to B \quad (x \notin \mathrm{FV}(B),\ B \text{ without } \vee, \exists).$$

The derivations of the second and the fourth formula are

$$\dfrac{\neg\neg C \to C \qquad \dfrac{\dfrac{\dfrac{\neg A \to \neg B \to \bot \qquad \dfrac{\dfrac{u_1\colon \neg C \qquad \dfrac{A \to C \quad u_2\colon A}{C}}{\bot}}{\neg A}\to^+ u_2}{\neg B \to \bot} \qquad \dfrac{\dfrac{u_1\colon \neg C \qquad \dfrac{B \to C \quad u_3\colon B}{C}}{\bot}}{\neg B}\to^+ u_3}{\bot}}{\neg\neg C}\to^+ u_1}{C}$$

and

$$\dfrac{\neg\neg B \to B \qquad \dfrac{\dfrac{\neg\forall_x \neg A \qquad \dfrac{\dfrac{\dfrac{u_1\colon \neg B \qquad \dfrac{\dfrac{\forall_x (A \to B) \quad x}{A \to B} \qquad u_2\colon A}{B}}{\bot}}{\neg A}\to^+ u_2}{\forall_x \neg A}}{\bot}}{\neg\neg B}\to^+ u_1}{B}$$

Classical derivability $\Gamma \vdash_c B$ was defined above by $\Gamma \cup \mathrm{Stab} \vdash B$. This embedding of classical logic into minimal logic can be expressed in a somewhat different and very explicit form, namely as a syntactic translation $A \mapsto A^g$ of formulas such that A is derivable in classical logic if and only if its translation A^g is derivable in minimal logic.

Definition (Gödel-Gentzen translation A^g).

$$(R\vec{t}\,)^g \quad := \neg\neg R\vec{t} \quad \text{for } R \text{ distinct from } \bot,$$

$$\begin{aligned}
\bot^g &:= \bot,\\
(A\vee B)^g &:= A^g \mathbin{\tilde\vee} B^g,\\
(\exists_x A)^g &:= \tilde\exists_x A^g,\\
(A\circ B)^g &:= A^g\circ B^g \quad \text{for } \circ = {\to}, \wedge,\\
(\forall_x A)^g &:= \forall_x A^g.
\end{aligned}$$

Lemma. *$\vdash \neg\neg A^g \to A^g$.*

Proof. Induction on A.

Case $R\vec t$ with R distinct from $\bot$. We must show $\neg\neg\neg\neg R\vec t \to \neg\neg R\vec t$, which is a special case of $\vdash \neg\neg\neg B\to\neg B$.

Case $\bot$. Use $\vdash\neg\neg\bot\to\bot$.

Case $A\vee B$. We must show $\vdash \neg\neg(A^g \mathbin{\tilde\vee} B^g)\to A^g\mathbin{\tilde\vee} B^g$, which is a special case of $\vdash\neg\neg(\neg C\to\neg D\to\bot)\to\neg C\to\neg D\to\bot$:

$$\dfrac{\neg\neg(\neg C\to\neg D\to\bot) \qquad \dfrac{\dfrac{\dfrac{u_1\colon \neg C\to\neg D\to\bot \qquad \neg C}{\neg D\to\bot}\qquad \neg D}{\bot}}{\neg(\neg C\to\neg D\to\bot)}\to^+ u_1}{\bot}$$

Case $\exists_x A$. In this case we must show $\vdash\neg\neg\tilde\exists_x A^g\to\tilde\exists_x A^g$, but this is a special case of $\vdash\neg\neg\neg B\to\neg B$, because $\tilde\exists_x A^g$ is the negation $\neg\forall_x\neg A^g$.

Case $A\wedge B$. We must show $\vdash\neg\neg(A^g\wedge B^g)\to A^g\wedge B^g$. By induction hypothesis $\vdash\neg\neg A^g\to A^g$ and $\vdash\neg\neg B^g\to B^g$. Now use part (a) of the stability theorem.

The cases $A\to B$ and $\forall_x A$ are similar, using parts (b) and (c) of the stability theorem instead. □

Theorem. (a) *$\Gamma\vdash_c A$ implies $\Gamma^g\vdash A^g$.*
(b) *$\Gamma^g\vdash A^g$ implies $\Gamma\vdash_c A$ for Γ, A without $\vee,\exists$.*

Proof. (a) Use induction on $\Gamma\vdash_c A$. For a stability axiom $\forall_{\vec x}(\neg\neg R\vec x\to R\vec x\,)$ we must derive $\forall_{\vec x}(\neg\neg\neg\neg R\vec x\to\neg\neg R\vec x\,)$, which is easy (as above). For the rules $\to^+$, $\to^-$, $\forall^+$, $\forall^-$, $\wedge^+$ and $\wedge^-$ the claim follows immediately from the induction hypothesis, using the same rule again. This works because the Gödel-Gentzen translation acts as a homomorphism for these connectives. For the rules $\vee_i^+$, $\vee^-$, $\exists^+$ and $\exists^-$ the claim follows from the induction hypothesis and the remark above. For example, in case $\exists^-$ the induction hypothesis gives

$$\begin{array}{c} \mid M \\ \tilde\exists_x A^g \end{array} \qquad\text{and}\qquad \begin{array}{c} u\colon A^g \\ \mid N \\ B^g \end{array}$$

with $x \notin \mathrm{FV}(B^g)$. Now use $\vdash (\neg\neg B^g \to B^g) \to \tilde{\exists}_x A^g \to \forall_x (A^g \to B^g) \to B^g$. Its premise $\neg\neg B^g \to B^g$ is derivable by the lemma above.

(b) First note that $\vdash_c (B \leftrightarrow B^g)$ if B is without $\lor, \exists$. Now assume that Γ, A are without $\lor, \exists$. From $\Gamma^g \vdash A^g$ we obtain $\Gamma \vdash_c A$ as follows. We argue informally. Assume Γ. Then Γ^g by the note, hence A^g because of $\Gamma^g \vdash A^g$, hence A again by the note. $\square$

CHAPTER 2

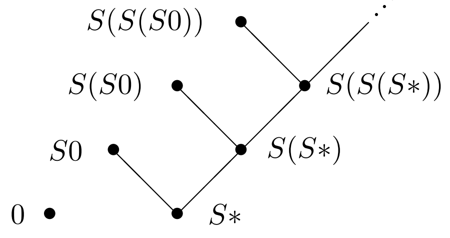
The model of partial continuous functionals

We now leave the general realm of logic and want to use it for concrete data types. To this end we concentrate on a particular model, the partial continuous functionals.

Proofs in mathematics generally deal with abstract, "higher-type" objects. Therefore an analysis of computational aspects of such proofs must be based on a theory of computation in higher types. A mathematically satisfactory such theory has been provided by Scott (1970) and Ershov (1977) (see also Chernov (1976)). The basic concept is that of a *partial continuous functional*. Since each such can be seen as a limit of its finite approximations, we get for free the notion of a computable functional: it is given by a recursive enumeration of finite approximations. The price to pay for this simplicity is that functionals are now *partial*, in stark contrast to the view of Gödel (1958) and Martin-Löf (1984). However, the total functionals can be defined as a subset of the partial ones. In fact, as observed by Kreisel, they form a dense subset with respect to the Scott topology.

A basic intuition is that we want describe an assignment $x \mapsto f(x)$ in the infinite (or "ideal") world by means of finite approximations. Then given an atomic piece b (a "token") of information on the value $f(x)$, we should have a finite set U (a "formal neighborhood") of tokens approximating the argument x such that $b \in f_0(U)$, where f_0 is a finite approximation of f.

Consider a base type, say the natural numbers given by their constructors zero 0 and successor S. Since we want the constructors to be continuous and with disjoint ranges we are forced to view also base type objects as given by approximations via tokens. For the natural numbers we have



Here the standard natural number objects are given by constructor expressions like $S(S0)$ not containing the token $*$ (which carries no information), but there is also an infinite object given by all tokens $S^n(*)$.

Important examples of computable functionals are the (structural) recursion operators of Hilbert (1925) and Gödel (1958), for instance

$$\mathcal{R}^{\tau}_{\mathbf{N}} \colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau$$

given by the defining equations

$$\begin{aligned} \mathcal{R}^{\tau}_{\mathbf{N}}(0, a, f) &= a, \\ \mathcal{R}^{\tau}_{\mathbf{N}}(S(n), a, f) &= f(n, \mathcal{R}^{\tau}_{\mathbf{N}}(n, a, f)). \end{aligned}$$

Similarly for lists of objects of type ρ we have

$$\mathcal{R}^{\tau}_{\mathbf{L}(\rho)} \colon \mathbf{L}(\rho) \to \tau \to (\rho \to \mathbf{L}(\rho) \to \tau \to \tau) \to \tau$$

with defining equations

$$\begin{aligned} \mathcal{R}^{\tau}_{\mathbf{L}(\rho)}(\mathrm{Nil}, a, f) &= a, \\ \mathcal{R}^{\tau}_{\mathbf{L}(\rho)}(x :: l, a, f) &= f(x, l, \mathcal{R}^{\tau}_{\mathbf{L}(\rho)}(l, a, f)). \end{aligned}$$

However, the defining equation

$$Y(f) = f(Y(f))$$

is admitted as well, and it defines a *partial* functional. Here f of type $\rho \to \sigma$ is called *total* if it maps total objects of type ρ to total objects of type σ.

Since the emphasis of this course is on the usage of decoration of proofs for program extraction, we do not give details on partial continuous functionals but refer the reader to Schwichtenberg and Wainer (2012, Ch. 6). In the rest of this chapter we introduce some simple (finitary) data types (or free algebras): natural numbers, lists and binary trees. It is discussed how one defines functions (called program constants in Minlog) by their defining equations, and how proofs by induction are done in Minlog.

2.1. Natural numbers

The standard example of a data type is that of the natural numbers. Natural numbers are implemented in Minlog as an algebra; the distribution comes equipped with a file, called `nat.scm`, which introduces this algebra. The algebra's constructors are 0 and `Succ` (zero and successor). To display these constructors, we simply write:

```
(display-alg "nat")
```

We obtain Minlog's reply:

```
> nat
        Zero:         nat
        Succ:         nat=>nat
```

Note also that for convenience Minlog allows us to write `0, 1, 2, 3, ...` instead of `Zero, (Succ Zero), Succ(Succ Zero), ....`

Algebras usually come equipped with some functions, which are called *program constants* in Minlog. For example, in the case of the natural numbers, one has the program-constants `NatPlus` and `NatTimes`, for addition and multiplication, respectively. These are displayed as `+` and `*`. The behaviour of program constants is specified by means of appropriate term rewriting rules which in Minlog are called *computation rules* and *rewrite rules* [1].

For example, to see the program constant `NatPlus` and its rules type:

```
(display-pconst "NatPlus")


> NatPlus
  comprules
        nat+0         nat
        nat1+Succ nat2         Succ(nat1+nat2)
  rewrules
        0+nat         nat
        Succ nat1+nat2         Succ(nat1+nat2)
        nat1+(nat2+nat3)      nat1+nat2+nat3
```

Note that `nat` is the default variable name of type nat. We recommend to have a look at the file `nat.scm` to familiarize oneself with the way program constants are defined.

To see the effect of term rewriting rules for `+` we type

```
(pp (nt (pt "3+4")))
(pp (nt (pt "Succ n+Succ m+0")))
```

which yields as results the number `7` and `Succ(Succ(n+m))`. Here `pp` stands for `pretty print` and `nt` stands for `normalize term`; this essentially consists in repeatedly applying[2] the term rewriting rules until no new term is obtained.

[1]The idea is that a computation rule can be understood as a description of a computation in a suitable *semantical* model, provided the syntactic constructors correspond to semantic ones in the model, whereas the other rules should be proved before being introduced.

[2]Term rewriting in Minlog makes use of normalisation-by-evaluation (see the reference manual).

2.1.1. Adding new program constants and computation rules.

We now wish to exemplify the introduction of new program constants on the natural numbers.

Recall that if the file `nat.scm` is not already loaded[3], we can type:

```
(set! COMMENT-FLAG #f)
(libload "nat.scm")
(set! COMMENT-FLAG #t)
```

We now introduce a new program constant which represents the function which doubles a natural number. The command used to introduce a new program constant is `add-program-constant`. It requires the name of the constant and its type; further arguments may be the degree of totality, the token type (e.g. `const`) and the arity (see the reference manual). In particular, note that in Minlog we can treat not only total objects but also partial ones[4]. Therefore, when we introduce a new program constant, we may also specify its totality degree. A totality degree of one (`t-deg-one`) indicates that the program constant is total, while zero (which is the default) denotes non–totality. As to the type, in the present case, the new constant `Double` is of arrow type, as it takes natural numbers as input and produces natural numbers as output.

```
(add-program-constant "Double" (py "nat=>nat"))
```

In case we wish to remove this program constant, we simply write:

```
(remove-program-constant "Double")
```

The behaviour of a new program constant can be specified by introducing one or more computation rules for it. This is accomplished by use of the command `add-computation-rule`, having two arguments: a left hand side and a right hand side. The right hand side specifies the result of the computation rule for the argument indicated in the left hand side.

The following example should clarify how to use these commands. The function "Double" can be defined by specifying primitively recursively how it acts on zero and on the successor of each natural number.

```
(add-computation-rules
 "Double 0" "0"
 "Double(Succ n)" "Succ(Succ(Double n))")
```

To see the effect of the newly introduced computation rules:

[3]Clearly, it is good practice to run a new Minlog session when loading new files which could turn out to be incompatible with previously loaded files or previously introduced definitions.

[4]For the notion of totality see (Stoltenberg-Hansen et al., 1994, Chapter 8.3); see also Schwichtenberg and Wainer (2012).

```
(pp (nt (pt "Double 3")))
(pp (nt (pt "Double(n+2)")))
```

2.1.2. Proof by induction. We give an example of a proof by induction on the natural numbers. The goal is very simple: we wish to show that `Double` $n = n + n$. The first step of the proof consists in using the command `ind`. This command requires a universally quantified goal and proves it by induction, according to the definition of the specific algebra type.

```
(set-goal "all n Double n=n+n")
(ind)
```

The effect of applying `ind` is to refine the goal to a proof of the base and the step cases of the induction. In the present case, where the constructors are Zero and Successor, we have to prove two cases: one for Zero and one for Successor. Minlog's reply will be something like this:

```
ok, ?_1 can be obtained from

  n2218
------------------------------------------------------
?_3:all n(Double n=n+n -> Double(Succ n)=Succ n+Succ n)

  n2218
------------------------------------------------------
?_2:Double 0=0+0
```

We then replace the goal with its normal form by letting:

```
(normalize-goal)

  n2218
------------------------------------------------------
?_4:T
```

The latter command can be abbreviated with `ng` and it normalizes the goal by using the computation rules for "+" introduced in the file `nat.scm`. More specifically, as both Double 0 and 0+0 reduce to 0, the normalization will first of all produce 0=0. This in turn reduces to truth, here indicated by `T`. It comes equipped with an axiom `Truth`, by means of which we prove the base case.

```
(use "Truth")

ok, ?_4 is proved.  The active goal now is
  n2218
------------------------------------------------------
?_3:all n(Double n=n+n -> Double(Succ n)=Succ n+Succ n)
```

As to the step, we make use of the induction hypothesis, `IH`, and write:

```
(assume "n" "IH")
(ng)
(use "IH")

> ok, we now have the new goal
  n2218  n  IH:Double n=n+n
-------------------------------------------------------
?_5:Double(Succ n)=Succ n+Succ n
> ok, the normalized goal is
  n2218  n  IH:Double n=n+n
-------------------------------------------------------
?_6:Double n=n+n
> ok, ?_6 is proved.  Proof finished.
```

Also in this case, when we write `ng` the term rewriting rules for `Double` and "+" are applied.

Finally, we wish to recall that one could also define the Double function without making use of a primitive recursive definition.

```
(add-program-constant "DoubleN" (py "nat=>nat"))
(add-computation-rules "DoubleN n" "n+n")
```

As an exercise, prove that the two definitions of the doubling function are equivalent:

```
(set-goal "all n Double n=DoubleN n")
```

Once we have proved the above statement for which the two definitions of Double are equivalent, we may add a rewrite rule which replaces each occurrence of `Double` by `DoubleN`.

```
(add-rewrite-rule (pt "Double n") (pt "DoubleN n"))
```

As an exercise, prove

```
(set-goal "all n,m n+m=m+n")
```

2.1.3. Boolean-valued functions. We now present some more examples of induction on the natural numbers, which introduce additional features of Minlog.

Suppose we want to prove that for all natural numbers n, Double n is even. We define two new boolean-valued program constants `Odd` and `Even` which take a natural number as argument and give a boolean (true or false) as output. As usual, the behaviour of these program constants can be specified by means of appropriate computation rules. In this case the computation rules will simultaneously characterize `Odd` and `Even`.

```
(add-program-constant "Odd" (py "nat=>boole"))
(add-program-constant "Even" (py "nat=>boole"))

(add-computation-rules
 "Odd 0" "False"
 "Even 0" "True"
 "Odd(Succ n)" "Even n"
 "Even(Succ n)" "Odd n")
```

The steps of the proof are self–explanatory:

```
(set-goal "all n Even(Double n)")
(ind)
(prop)
(search)
```

Boolean-valued functions provide a convenient way to represent decidable predicates. For example, in the case of the natural numbers, one has the program-constant `NatLt`, for the less-than relation. It is displayed as `<`.

```
(display-pconst "NatLt")

NatLt
  comprules
        nat<0                      False
        0<Succ nat                 True
        Succ nat1<Succ nat2        nat1<nat2
  rewrules
        nat<Succ nat               True
        nat<nat                    False
        Succ nat<nat               False
        nat1+nat2<nat1             False
```

We now treat a slightly more complex example of an arithmetical proof, which will be used below to discuss the computational content of proofs in an easy case. We prove by induction on n that n can be divided by $m+1$ with some quotient q and remainder r. The proof formalizes the informal proof by cases: if $r < m$ let $q' = q$ and $r' = r+1$, and if $r = m$ let $q' = q+1$ and $r' = 0$. Viewing `<` as a boolean-valued function makes it easy carry out such case distinctions in proofs. The formalization of the proof is in Appendix B.1.

2.2. Lists

We now consider lists over an arbitrary type α, and give some examples to illustrate again the use of induction. However, since we now deal with *parametrized algebras* (see the reference manual or Schwichtenberg and Wainer (2012)) the task turns out to be a bit harder than when working with the algebra of natural numbers.

2.2.1. List reversal as a program constant. To start with we load the file `list.scm`, which contains basic definitions and operations on lists over an arbitrary type α[5]. Then we introduce a function, Rv, on lists which has the effect of reverting a list. Finally we prove:

$$\forall_{v,w} \mathrm{Rv}\,(v * w) =^{\mathrm{d}} (\mathrm{Rv}\,w) * (\mathrm{Rv}\,v),$$

where v and w are lists over an arbitrary type α and $*$ denotes the append function on lists as defined in `list.scm`. Further, $=^{\mathrm{d}}$ represent Leibniz' equality[6]: two elements are equal if they have the same properties, i.e., they are indistinguishable.

We begin as follows:

```
;; (libload "nat.scm")
(set! COMMENT-FLAG #f)
(libload "list.scm")
(set! COMMENT-FLAG #t)

(add-var-name "x" "a" "b" "c" "d" (py "alpha"))
(add-var-name "xs" "v" "w" "u" (py "list alpha"))
```

We now need to define Rv. This is defined inductively, by first giving its value for the empty list and then saying how it applies to a non-empty list. The two defining conditions for Rv are the following:

$$\mathrm{Rv}\,(\mathrm{Nil}\,\alpha) = (\mathrm{Nil}\,\alpha),$$
$$\mathrm{Rv}\,(a :: w) = (\mathrm{Rv}\,w) * (a{:})$$

where, according to the notation in `list.scm`, $(\mathrm{Nil}\,\alpha)$ denotes the empty list over the type α, $a :: w$ denotes the list obtained by adding the object a of type α to the list w (over α), while $a{:}$ is the one element list obtained from a. We thus write:

```
(add-program-constant "ListRv"
            (py "list alpha => list alpha") t-deg-one)
(add-prefix-display-string "ListRv" "Rv")

(add-computation-rules
```

[5]Note that `list.scm` does require to first upload `nat.scm`. We recommend to go through the list file before working out this example.

[6]Internally Leibniz equality is printed `eqd`, where the `d` stands for "defined", since Leibniz equality is the (non-computational) predicate inductively defined by the clause `InitEqD`: $\forall_x (x =^{\mathrm{d}} x)$.

```
 "Rv(Nil alpha)" "(Nil alpha)"
 "Rv(x::xs)" "Rv xs++x:")
```

Note that for simplicity we have stated that `ListRv` is a total function. Minlog's output will include a warning, to remind us that we should have proved that `ListRv` is in fact total. Also, `add-prefix-display-string` allows us to define a token for the program constant[7].

The following proof makes use of a program constant, `ListAppd`, already available within the file `list.scm`. This has the following computation rules:

```
        (Nil alpha)++xs2      xs2
        (x1::xs1)++xs2        x1::xs1++xs2
```

And rewrite rules:

```
        xs++(Nil alpha)       xs
        xs1++x2: ++xs2        xs1++(x2::xs2)
```

To check `ListAppd` we type:

```
(display-pconst "ListAppd")
```

Now we can set the goal and start the proof by calling `ind`:

```
(set-goal "all v,w Rv(v++w)eqd Rv w++Rv v")
(ind)
```

This has the effect of producing two subgoals, corresponding to the base case and the step case, respectively. We tackle the base case as follows:

```
(ng)
(assume "w")
(use "InitEqD")
```

Here we have used `InitEqD`, which is the axiom: `xs eqd xs`.

Subsequently we move to the step case:

```
(assume "a" "v" "IHw" "w")
(ng)
(simp "IHw")
```

And finally we use a theorem proved in the file `list.scm` and there called `ListAppdAssoc`:

```
all xs1,xs2,xs3 xs1++(xs2++xs3)eqd xs1++xs2++xs3
```

Then we carry on by

```
(simp "ListAppdAssoc")
(use "InitEqD")
```

[7]In the file `list.scm` there already exists a program constant `ListRev` with display string `Rev` defined exactly as our `ListRv`. Here we have just duplicated the definition for pedagogical reasons, since it is a nice and easy example. However, we had to choose a different name to avoid a clash with the already loaded `list.scm`.

Here is an exercise: in `list.scm` `ListMap` is introduced by

```
(add-program-constant
 "ListMap" (py "(alpha1=>alpha2)=>list alpha1=>list alpha2"))

(add-infix-display-string "ListMap" "map" 'pair-op)

(add-var-name "phi" (py "alpha1=>alpha2"))

(add-computation-rules
 "phi map(Nil alpha1)" "(Nil alpha2)"
 "phi map y::ys" "phi y::phi map ys")
```

Prove that `map` commutes with `Rv`:

```
(add-var-name "f" (py "alpha=>alpha"))
(set-goal "all f,xs (f map Rv xs)eqd Rv(f map xs)")
```

In the proof it is helful to use the theorem `MapAppd`, which can be found in `list.scm`.

2.2.2. Proving the existence of the reverted list.

We give an informal existence proof for list reversal. Write vw for the result $v * w$ of appending the list w to the list v, vx for the result $v * x$: of appending the one element list x: to the list v, and xv for the result $x :: v$ of constructing a list by writing an element x in front of a list v, and omit the parentheses in $R(v, w)$ for (typographically) simple arguments. Assume

$$\text{InitRevI}\colon R(\text{Nil}, \text{Nil}), \tag{2.1}$$

$$\text{GenRevI}\colon \forall_{v,w,x}(Rvw \to R(vx, xw)). \tag{2.2}$$

We view R as an inductively defined predicate without computational content. The reader should not be confused: of course these formulas involving R do express how a computation of the reverted list should proceed. But the predicate R itself only represents the graph of the list reversal function.

A straightforward proof of $\forall_v \exists_w Rvw$ proceeds as follows. We first prove a lemma `ListInitLastNat` stating that every non-empty list can be written in the form vx. Using it, $\forall_v \exists_w Rvw$ can be proved by induction on the length of v. In the step case, our list is non-empty, and hence can be written in the form vx. Since v has smaller length, the induction hypothesis yields its reversal w. Then we can take xw. The formalization of this proof is in Appendix B.2. Later (in 4.1) we will come back to this proof, and extract its computational content.

2.3. Binary trees

We now wish to show how to introduce new algebras; we shall also give one more example on how to use them. The example we shall consider is that of binary trees. First of all we introduce a new algebra, called “bin” which has constructors “BinNil” and “BinBranch”.

```
(add-algs "bin"
          '("bin" "BinNil")
          '("bin=>bin=>bin" "BinBranch"))
(add-var-name "r" "s" "t" (py "bin"))
```

Clearly nodes in a binary tree can be viewed as lists of booleans, where “True” means left and “False” means right. The *Brouwer-Kleene ordering* on a binary tree is defined as follows:

```
;; (set! COMMENT-FLAG #f)
;; (libload "nat.scm")
;; (libload "list.scm")
;; (set! COMMENT-FLAG #t)

(add-var-name "p" "q" (py "boole"))
(add-var-name "a" "b" "c" (py "list boole"))

(add-program-constant
 "LtBK" (py "list boole=>list boole=>boole") t-deg-one)
(add-infix-display-string "LtBK" "<<" 'rel-op)

(add-computation-rules
 "(Nil boole)<<b" "False"
 "(p::a)<<(Nil boole)" "True"
 "(True::a)<<(True::b)" "a<<b"
 "(True::a)<<(False::b)" "True"
 "(False::a)<<(True::b)" "False"
 "(False::a)<<(False::b)" "a<<b")
```

We will show that the Brouwer-Kleene ordering `<<` is linear.

To express when a node `a` is an element of a binary tree `r` we use

```
(add-program-constant
 "NodeIn" (py "list boole=>bin=>boole") t-deg-one)
(add-infix-display-string "NodeIn" "nodein" 'rel-op)

(add-computation-rules
 "(Nil boole)nodein r" "True"
 "(p::a)nodein BinNil" "False"
 "(True::a)nodein BinBranch r s" "a nodein r"
 "(False::b)nodein BinBranch r s" "b nodein s")
```

We will need the *size* of a binary tree, defined by

```
(add-program-constant "Size" (py "bin=>nat") t-deg-one)
(add-computation-rules
 "Size BinNil" "1"
 "Size(BinBranch r s)" "Succ(Size r+Size s)")
```

and also when a list of booleans in *increasing* w.r.t. <<:

```
(add-program-constant
 "Incr" (py "list list boole=>boole") t-deg-one)

(add-var-name "as" "bs" "cs" (py "list list boole"))

(add-computation-rules
 "Incr(Nil list boole)" "True"
 "Incr a:" "True"
 "Incr(a::b::bs)" "a<<b andb Incr(b::bs)")
```

We will need the following propositions, whose proofs are either in `list.scm`:

```
ListProjAppdLeft:
all xs1,n,xs2(n<Lh xs1 -> (n thof(xs1++xs2))eqd(n thof xs1))

ListProjAppdRight:
all xs1,n,xs2 (Lh xs1)+n thof(xs1++xs2)eqd(n thof xs2)
```

or left as exercises:

```
ListAppdProp:
  all as(all n(n<Lh as -> (Pvar list boole)(n thof as)) ->
  all bs(all n(n<Lh bs -> (Pvar list boole)(n thof bs)) ->
  all n(n<Lh as+Lh bs -> (Pvar list boole)(n thof as++bs))))

IncrAppd: all as(Incr as -> all bs(Incr bs ->
          all n(n<Lh as -> (n thof as)<<(0 thof bs)) ->
          Incr(as++bs)))

IncrMap: all as(Incr as -> all p Incr((Cons boole)p map as))

IncrTrueFalse: all as,n(n<Lh as -> all bs(0<Lh bs ->
               (n thof(Cons boole)True map as)<<
               (0 thof(Cons boole)False map bs)))
```

We prove that the nodes of a binary tree `r` are linearly ordered by the Brouwer-Kleene ordering. More precisely, we show that for every binary tree `r` we can find a strictly increasing (in the sense of the Brouwer-Kleene ordering) list consisting exactly of all nodes of `r`. The proof is by induction on `r`. In the step one takes the two increasing lists obtained by applying the induction hypothesis to the two subtrees, and places the present node in between. The formalization of this proof is in Appendix B.3. Later (in 4.1.5) we will come back to this proof, and extract its computational content.

CHAPTER 3

Formulas as problems

In his paper "Zur Deutung der intuitionistischen Logik" of 1932 Kolmogorov proposes to view a formula A as a "computational problem". Then what should be the solution to the problem posed by the formula $I\vec{r}$, where I is inductively defined? The obvious idea here is to take a "generation tree", witnessing how the arguments $\vec{r}$ were put into I. For example, consider the clauses $\mathrm{Even}(0)$ and $\forall^{\mathrm{nc}}_n(\mathrm{Even}(n) \to \mathrm{Even}(S(Sn)))$. A generation tree for Even(6) should consist of a single branch with nodes Even(0), Even(2), Even(4) and Even(6). More formally, such a generation tree can seen as an ideal in a certain algebra ι_I associated naturally with I.

Consider the more general situation when parameters are involved, i.e., when we have a proof of a closed formula $\forall_{\vec{x}}(\vec{A} \to I\vec{r})$. It is of obvious interest which of the variables $\vec{x}$ and assumptions $\vec{A}$ are actually used in the "solution" provided by the proof (in the sense of Kolmogorov (1932)). To be able to express dependence on and independence of such parameters we split each of our (only) logical connectives $\to, \forall$ into two variants, a "computational" one $\forall^{\mathrm{c}}, \to^{\mathrm{c}}$ and a "non-computational" one $\forall^{\mathrm{nc}}, \to^{\mathrm{nc}}$. This distinction (for the universal quantifier) is due to Berger (1993, 2005). One can view this "decoration" of $\to, \forall$ as turning our (minimal) logic into a "computational logic", which is able to express dependence on and independence of parameters. The rules for $\to^{\mathrm{nc}}, \forall^{\mathrm{nc}}$ are similar to the ones for $\to^{\mathrm{c}}, \forall^{\mathrm{c}}$; they will be defined in Section 3.2.

Now the clauses of inductive predicates can and should be decorated as well, or instance in the form

$$\forall^{\mathrm{nc}}_{\vec{x}}\forall^{\mathrm{c}}_{\vec{y}}(\vec{A} \to^{\mathrm{nc}} \vec{B} \to^{\mathrm{c}} X\vec{r}).$$

This will lead to a different (i.e., simplified) algebra ι_I associated with the inductive predicate I.

Prime formulas $I\vec{r}$ with $\iota_I = \mathbf{U}$ (the "unit" type, consisting of one element only) have a trivial generation tree, and in this sense are without computational content. Clearly this is also the case for formulas with such an $I\vec{r}$ as conclusion. These formulas are called *non-computational* (n.c.) or *Harrop formulas*. Moreover, a Harrop formula in a premise can be ignored when we are interested in the computational content of a proof of this formula: its only contribution would be of unit type. We will define the "type of a formula" $\tau(A)$ (i.e., the type of its solution) accordingly; it will not involve the unit type.

3.1. Decorated predicates and formulas

We introduce decorated connectives $\to^{\mathrm{c}}, \forall^{\mathrm{c}}$ and $\to^{\mathrm{nc}}, \forall^{\mathrm{nc}}$, and also decorated *least-fixed-point operators* $\mu^{\mathrm{c}}, \mu^{\mathrm{nc}}$, used to generate inductively defined predicates. Moreover we distinguish two sorts of predicate variables, computationally relevant ones written $X, Y, Z \dots$ and non-computational ones written $\hat{X}, \hat{Y}, \hat{Z} \dots$. Then we can define *decorated predicates and formulas* as follows. *Formulas* are

$$A, B ::= P\vec{r} \mid A \to^{\mathrm{c}} B \mid A \to^{\mathrm{nc}} B \mid \forall^{\mathrm{c}}_x A \mid \forall^{\mathrm{nc}}_x A$$

and *predicates*

$$P, Q ::= X \mid \hat{X} \mid \{\, \vec{x} \mid A \,\} \mid \mu^{\mathrm{c/nc}}_X (\forall^{\mathrm{c/nc}}_{\vec{x}_i}((A_{i\nu})_{\nu<n_i} \to^{\mathrm{c/nc}} X\vec{r}_i))_{i<k}$$

with $k \geq 1$ and $\vec{x}_i$ all free variables in $(A_{i\nu})_{\nu<n_i} \to^{\mathrm{c/nc}} X\vec{r}_i$. In the $\mu^{\mathrm{c/nc}}$ case we require that X occurs only "strictly positive" in the formulas $A_{i\nu}$, i.e., never on the left hand side of an implication.

We define when a predicate or formula is *non-computational (n.c.)* (or *Harrop*) as follows.

- $\hat{X}$ is n.c. but X is not,
- $\{\, \vec{x} \mid A \,\}$ is n.c. if A is,
- $\mu^{\mathrm{nc}}_X \vec{K}$ is n.c. but $\mu_X \vec{K}$ is not,
- $P\vec{r}$ is n.c. if P is,
- $A \to^{\mathrm{c/nc}} B$ is n.c. if B is, and
- $\forall^{\mathrm{c/nc}}_x A$ is n.c. if A is.

The other predicates and formulas are called *computationally relevant* (c.r.).

We usually write $\to$, $\forall$, μ for $\to^{\mathrm{c}}$, $\forall^{\mathrm{c}}$, μ^{c}. Also notice that in the clauses of an n.c. inductive predicate $\mu^{\mathrm{nc}}_X \vec{K}$ decorations play no role; hence we write $\to, \forall$ for $\to^{\mathrm{c/nc}}, \forall^{\mathrm{c/nc}}$.

We refine the distinction between computationally relevant (c.r.) and non-computational (n.c.) predicates and formulas by providing a type in the former case. To indicate that there is no computational content we introduce a "nulltype" symbol $\circ$ and extend the use of $\rho \to \sigma$ by

$$(\rho \to \circ) := \circ, \quad (\circ \to \sigma) := \sigma, \quad (\circ \to \circ) := \circ.$$

Definition (Type $\tau(C)$ of a predicate or formula C). Assume a global injective assignment of a type variable ξ to every c.r. predicate variable X.

Let $\tau(C) := \circ$ if C is non-computational. In case C is c.r. let

$$\begin{aligned}
&\tau(P\vec{r}\,) := \tau(P),\\
&\tau(A \to B) := (\tau(A) \to \tau(B)), \quad \tau(A \to^{\mathrm{nc}} B) := \tau(B),\\
&\tau(\forall_{x^\rho} A) := (\rho \to \tau(A)), \quad \tau(\forall^{\mathrm{nc}}_{x^\rho} A) := \tau(A),\\
&\tau(X) := \xi,\\
&\tau(\{\,\vec{x} \mid A\,\}) := \tau(A),\\
&\tau(\underbrace{\mu_X(\forall^{\mathrm{nc}}_{\vec{x}_i}\forall_{\vec{y}_i}(\vec{A}_i \to^{\mathrm{nc}} \vec{B}_i \to X\vec{r}_i))_{i<k}}_{I}) := \underbrace{\mu_\xi(\tau(\vec{y}_i) \to \tau(\vec{B}_i) \to \xi)_{i<k}}_{\iota_I}.
\end{aligned}$$

We call ι_I the *algebra associated with I.*

To avoid unnecessarily complex types we extend the use of $\rho \times \sigma$ to the nulltype sumbol $\circ$ by

$$(\rho \times \circ) := \rho, \quad (\circ \times \sigma) := \sigma, \quad (\circ \times \circ) := \circ.$$

Moreover we identify the unit type $\mathbf{U}$ with $\circ$.

Examples. For the even numbers we now have two variants:

$$\begin{aligned}
\mathrm{EvenI} &:= \mu_X(X0, \forall^{\mathrm{nc}}_n(Xn \to X(S(Sn)))),\\
\mathrm{EvenI}^{\mathrm{nc}} &:= \mu^{\mathrm{nc}}_X(X0, \forall_n(Xn \to X(S(Sn)))).
\end{aligned}$$

In Minlog this is written as

```
(add-ids
  (list (list "EvenI" (make-arity (py "nat")) "algEvenI"))
  '("EvenI 0" "InitEvenI")
  '("allnc n(EvenI n -> EvenI(n+2))" "GenEvenI"))

(add-ids
  (list (list "EvenNc" (make-arity (py "nat"))))
  '("EvenNc 0" "InitEvenNc")
  '("all n(EvenNc n -> EvenNc(n+2))" "GenEvenNc"))
```

Generally for every c.r. inductive predicate I (i.e., defined as $\mu_X\vec{K}$) we have an n.c. variant I^{nc} defined as $\mu^{\mathrm{nc}}_X\vec{K}$.

Since decorations can be inserted arbitrarily and parameter predicate variables can be chosen as either n.c. or c.r. we obtain many variants of inductive predicates. For the existential quantifier we have

$$\begin{aligned}
\mathrm{ExD}_Y &:= \mu_X(\forall_x(Yx \to X)),\\
\mathrm{ExL}_Y &:= \mu_X(\forall_x(Yx \to^{\mathrm{nc}} X)).\\
\mathrm{ExR}_Y &:= \mu_X(\forall^{\mathrm{nc}}_x(Yx \to X)),
\end{aligned}$$

$$\mathrm{ExU}_Y := \mu^{\mathrm{nc}}_X(\forall^{\mathrm{nc}}_x(Yx \to^{\mathrm{nc}} X)).$$

Here D is for "double", L for "left", R for "right" and U for "uniform". We will use the abbreviations

$$\begin{aligned}
\exists^{\mathrm{d}}_x A &:= \mathrm{ExD}_{\{x|A\}},\\
\exists^{\mathrm{l}}_x A &:= \mathrm{ExL}_{\{x|A\}},\\
\exists^{\mathrm{r}}_x A &:= \mathrm{ExR}_{\{x|A\}},\\
\exists^{\mathrm{u}}_x A &:= \mathrm{ExU}_{\{x|A\}}.
\end{aligned}$$

For intersection we only consider the nullary case (i.e., conjunction). Then

$$\begin{aligned}
\mathrm{CapD}_{Y,Z} &:= \mu_X(Y \to Z \to X),\\
\mathrm{CapL}_{Y,Z} &:= \mu_X(Y \to Z \to^{\mathrm{nc}} X),\\
\mathrm{CapR}_{Y,Z} &:= \mu_X(Y \to^{\mathrm{nc}} Z \to X),\\
\mathrm{CapU}_{Y,Z} &:= \mu^{\mathrm{nc}}_X(Y \to^{\mathrm{nc}} Z \to^{\mathrm{nc}} X).
\end{aligned}$$

We use the abbreviations

$$\begin{aligned}
A \wedge^{\mathrm{d}} B &:= \mathrm{CapD}_{\{|A\},\{|B\}},\\
A \wedge^{\mathrm{l}} B &:= \mathrm{CapL}_{\{|A\},\{|B\}},\\
A \wedge^{\mathrm{r}} B &:= \mathrm{CapR}_{\{|A\},\{|B\}},\\
A \wedge^{\mathrm{u}} B &:= \mathrm{CapU}_{\{|A\},\{|B\}}.
\end{aligned}$$

For union again we only consider the nullary case (i.e., disjunction). Then

$$\begin{aligned}
\mathrm{CupD}_{Y,Z} &:= \mu_X(Y \to X,\ Z \to X),\\
\mathrm{CupL}_{Y,Z} &:= \mu_X(Y \to X,\ Z \to^{\mathrm{nc}} X),\\
\mathrm{CupR}_{Y,Z} &:= \mu_X(Y \to^{\mathrm{nc}} X,\ Z \to X),\\
\mathrm{CupU}_{Y,Z} &:= \mu_X(Y \to^{\mathrm{nc}} X,\ Z \to^{\mathrm{nc}} X),\\
\mathrm{CupNC}_{Y,Z} &:= \mu^{\mathrm{nc}}_X(Y \to X,\ Z \to X).
\end{aligned}$$

The final nc-variant is used to suppress even the information which clause has been used. We use the abbreviations

$$\begin{aligned}
A \vee^{\mathrm{d}} B &:= \mathrm{CupD}_{\{|A\},\{|B\}},\\
A \vee^{\mathrm{l}} B &:= \mathrm{CupL}_{\{|A\},\{|B\}},\\
A \vee^{\mathrm{r}} B &:= \mathrm{CupR}_{\{|A\},\{|B\}},\\
A \vee^{\mathrm{u}} B &:= \mathrm{CupU}_{\{|A\},\{|B\}},\\
A \vee^{\mathrm{nc}} B &:= \mathrm{CupNC}_{\{|A\},\{|B\}}.
\end{aligned}$$

For Leibniz equality we take the definition

$$\mathrm{EqD} := \mu^{\mathrm{nc}}_X(\forall_x Xxx).$$

3.2. Logical rules for the decorated connectives

We also need to adapt our logical rules to the decorated connectives $\to, \to^{\text{nc}}$ and $\forall, \forall^{\text{nc}}$. The introduction and elimination rules for $\to$ and $\forall$ remain as before, and also the elimination rules for $\to^{\text{nc}}$ and $\forall^{\text{nc}}$. However, the introduction rules for $\to^{\text{nc}}$ and $\forall^{\text{nc}}$ must be restricted: the abstracted (assumption or object) variable must be "non-computational", in the following sense. Simultaneously with a derivation M we define the sets $\text{CV}(M)$ and $\text{CA}(M)$ of *computational* object and assumption variables of M, as follows. Let M^A be a derivation. If A is non-computational (n.c.) then $\text{CV}(M^A) := \text{CA}(M^A) := \emptyset$. Otherwise

$$
\begin{aligned}
&\text{CV}(c^A) := \emptyset \quad (c^A \text{ an axiom}),\\
&\text{CV}(u^A) := \emptyset,\\
&\text{CV}((\lambda_{u^A} M^B)^{A\to B}) := \text{CV}((\lambda_{u^A} M^B)^{A\to^{\text{nc}} B}) := \text{CV}(M),\\
&\text{CV}((M^{A\to B} N^A)^B) := \text{CV}(M) \cup \text{CV}(N),\\
&\text{CV}((M^{A\to^{\text{nc}} B} N^A)^B) := \text{CV}(M),\\
&\text{CV}((\lambda_x M^A)^{\forall_x A}) := \text{CV}((\lambda_x M^A)^{\forall^{\text{nc}}_x A}) := \text{CV}(M) \setminus \{x\},\\
&\text{CV}((M^{\forall_x A(x)} r)^{A(r)}) := \text{CV}(M) \cup \text{FV}(r),\\
&\text{CV}((M^{\forall^{\text{nc}}_x A(x)} r)^{A(r)}) := \text{CV}(M),
\end{aligned}
$$

and similarly

$$
\begin{aligned}
&\text{CA}(c^A) := \emptyset \quad (c^A \text{ an axiom}),\\
&\text{CA}(u^A) := \{u\},\\
&\text{CA}((\lambda_{u^A} M^B)^{A\to B}) := \text{CA}((\lambda_{u^A} M^B)^{A\to^{\text{nc}} B}) := \text{CA}(M) \setminus \{u\},\\
&\text{CA}((M^{A\to B} N^A)^B) := \text{CA}(M) \cup \text{CA}(N),\\
&\text{CA}((M^{A\to^{\text{nc}} B} N^A)^B) := \text{CA}(M),\\
&\text{CA}((\lambda_x M^A)^{\forall_x A}) := \text{CA}((\lambda_x M^A)^{\forall^{\text{nc}}_x A}) := \text{CA}(M),\\
&\text{CA}((M^{\forall_x A(x)} r)^{A(r)}) := \text{CA}((M^{\forall^{\text{nc}}_x A(x)} r)^{A(r)}) := \text{CA}(M).
\end{aligned}
$$

The introduction rules for $\to^{\text{nc}}$ and $\forall^{\text{nc}}$ then are

(i) If M^B is a derivation and $u^A \notin \text{CA}(M)$ then $(\lambda_{u^A} M^B)^{A\to^{\text{nc}} B}$ is a derivation.

(ii) If M^A is a derivation, x is not free in any formula of a free assumption variable of M and $x \notin \text{CV}(M)$, then $(\lambda_x M^A)^{\forall^{\text{nc}}_x A}$ is a derivation.

An alternative way to formulate these rules is simultaneously with the notion of the "extracted term" $\text{et}(M)$ of a derivation M.

3.3. Decorated axioms

Consider a c.r. inductive predicate

$$I := \mu_X(\forall^{\mathrm{c/nc}}_{\vec{x}_i}((A_{i\nu}(X))_{\nu<n_i} \to^{\mathrm{c/nc}} X\vec{r}_i))_{i<k}.$$

Then for every $i < k$ we have a *clause* (or *introduction axiom*)

$$I_i^+ \colon \forall^{\mathrm{c/nc}}_{\vec{x}_i}((A_{i\nu}(I))_{\nu<n_i} \to^{\mathrm{c/nc}} I\vec{r}_i). \tag{3.1}$$

Moreover, we have an *elimination axiom*

$$I^- \colon \forall^{\mathrm{nc}}_{\vec{x}}(I\vec{x} \to (\forall^{\mathrm{c/nc}}_{\vec{x}_i}((A_{i\nu}(I \cap^{\mathrm{d}} X))_{\nu<n_i} \to^{\mathrm{c/nc}} X\vec{r}_i))_{i<k} \to X\vec{x}\,). \tag{3.2}$$

For example

$$\begin{aligned}
(\mathrm{ExD}_{\{x|A\}})_0^+ &\colon \forall_x(A \to \exists^{\mathrm{d}}_x A),\\
(\mathrm{ExL}_{\{x|A\}})_0^+ &\colon \forall_x(A \to^{\mathrm{nc}} \exists^{\mathrm{l}}_x A),\\
(\mathrm{ExR}_{\{x|A\}})_0^+ &\colon \forall^{\mathrm{nc}}_x(A \to \exists^{\mathrm{r}}_x A),\\
(\mathrm{ExU}_{\{x|A\}})_0^+ &\colon \forall^{\mathrm{nc}}_x(A \to^{\mathrm{nc}} \exists^{\mathrm{u}}_x A).
\end{aligned}$$

When $\{\, x \mid A \,\}$ is clear from the context we abbreviate

$$\begin{aligned}
(\exists^{\mathrm{d}})^+ &:= (\mathrm{ExD}_{\{x|A\}})_0^+,\\
(\exists^{\mathrm{l}})^+ &:= (\mathrm{ExL}_{\{x|A\}})_0^+,\\
(\exists^{\mathrm{r}})^+ &:= (\mathrm{ExR}_{\{x|A\}})_0^+,\\
(\exists^{\mathrm{u}})^+ &:= (\mathrm{ExU}_{\{x|A\}})_0^+.
\end{aligned}$$

For an n.c. inductive predicate $\hat{I}$ the introduction axioms $(\hat{I})_i^+$ are formed similarly. However, the elimination axiom $(\hat{I})^-$ needs to be restricted to non-computational competitor predicates $\hat{X}$, except when $\hat{I}$ is given by a *one-clause-nc* definition (i.e., with only one clause involving $\to^{\mathrm{nc}}, \forall^{\mathrm{nc}}$ only). Examples are Leibniz equality `EqD`, and uniform variants `ExU` and `AndU` of the existential quantifier and conjunction.

Recall that totality for the natural numbers was defined by the clauses

$$\begin{aligned}
&\texttt{TotalNatZero}\colon \texttt{TotalNat}\ 0\\
&\texttt{TotalNatSucc}\colon \forall^{\mathrm{nc}}_{\hat{n}}(\texttt{TotalNat}\ \hat{n} \to \texttt{TotalNat}(\texttt{Succ}\ \hat{n}))
\end{aligned}$$

Using $\forall_{n\in T}Pn$ to abbreviate $\forall^{\mathrm{nc}}_{\hat{n}}(T_{\mathbf{N}}\hat{n} \to P\hat{n})$, the elimination axiom for `TotalNat` can be written as

$$\mathrm{Ind}_{n,A(n)}\colon \forall_{n\in T}(A(0) \to \forall_{n\in T}(A(n) \to A(Sn)) \to A(n^{\mathbf{N}})).$$

This is the usual induction axiom for natural numbers. We further abbreviate $\forall_{n\in T}Pn$ by $\forall_n Pn$, where using n rather than $\hat{n}$ indicates the n is meant to be restricted to the totality predicate T.

CHAPTER 4

Computational content of proofs

Proofs have two aspects: they provide insight into why an argument is correct, and they can also have computational content. The Brouwer-Heyting-Kolmogorov interpretation (BHK-interpretation for short) gives a good analysis of the latter.

Recall that a formula can be seen as a problem, and its proof as providing a solution to this problem. Accordingly the clauses of the BHK-interpretation are:

(i) p proves $A \to B$ if and only if p is a construction transforming any proof q of A into a proof $p(q)$ of B;
(ii) $\bot$ is a proposition without proof;
(iii) p proves $\forall_{x \in D} A(x)$ if and only if p is a construction such that for all $d \in D$, $p(d)$ proves $A(d)$;

The problem with the BHK-interpretation clearly is its reliance on some unexplained notions, in particular

what is a "construction"?
what is a proof of a prime formula?

Here we propose to take

construction := computable functional,
proof of a prime formula $I\vec{r}$:= a "generation tree" for $I\vec{r}$.

4.1. Realizability

Every constructive proof of an existential theorem (or "problem"; cf. Kolmogorov (1932)) contains – by the very meaning of "constructive proof" – a construction of a solution in terms of the parameters of the problem. To get hold of such a solution we have two methods.

Write-and-verify. Guided by our understanding of how the constructive proof works we directly write down a program to compute the solution, and then formally prove ("verify") that this indeed is the case.

Prove-and-extract. Formalize the constructive proof, and then extract the computational content of this proof in the form of a realizing term t.

The soundness theorem guarantees (and even provides a formal proof) that t is a solution to the problem.

In simple cases the two methods are often essentially the same. However, in more complex situations the prove-and-extract method seems to be preferable, for the following reasons.

(i) Dealing with a problem on the proof level makes it possible to use more abstract mathematical tools.
(ii) Generally a better organization of the material becomes possible, which is an essential aspect of a good mathematical analysis of a problem.
(iii) Such a structural approach leads to a better understanding of what is going on, which will make it easier to adapt the proof to a somewhat changed specification.

Recall the definition of the type $\tau(A)$ of a formula A, the type of the solution to the problem posed by this formula. We define what it means for an x of type $\tau(A)$ to be a "realizer" (i.e., a solution) of the formula A. Next we assign to any derivation M of a formula A its "extracted term" $\mathrm{et}(M)$, which should be the realizer of A provided by the proof M. Finally we state the soundness theorem, saying that this indeed is the case. For a proof of the latter and generally a more detailed exposition of the theory the reader is referred to Schwichtenberg and Wainer (2012, Section 7.2). Then we give some examples of program extraction from constructive proofs.

4.1.1. Realizability relation. Assume that we have a global assignment giving for every (c.r.) predicate variable X of arity $\vec{\rho}$ an n.c. predicate variable $X^{\mathbf{r}}$ of arity $(\tau(X), \vec{\rho})$ together with an invariance axiom

$$\forall^{\mathrm{nc}}_{\vec{x}}(X\vec{x} \leftrightarrow \exists^{\mathrm{l}}_{u} X^{\mathbf{r}} u\vec{x}\,).$$

Definition ($C^{\mathbf{r}}$ for predicates and formulas C). For every predicate or formula C we define an n.c. predicate $C^{\mathbf{r}}$. For n.c. C let

$$C^{\mathbf{r}} := C.$$

In case C is c.r. the arity of $C^{\mathbf{r}}$ is $(\tau(C), \vec{\sigma}\,)$ with $\vec{\sigma}$ the arity of C. For c.r. formulas we define

$$\begin{aligned}
(P\vec{r}\,)^{\mathbf{r}} &:= \{\, u \mid P^{\mathbf{r}} u\vec{r}\,\} \\
(A \to B)^{\mathbf{r}} &:= \begin{cases} \{\, u \mid \forall_v (A^{\mathbf{r}} v \to B^{\mathbf{r}}(uv))\,\} & \text{if } A \text{ is c.r.} \\ \{\, u \mid A \to B^{\mathbf{r}} u\,\} & \text{if } A \text{ is n.c.} \end{cases} \\
(A \to^{\mathrm{nc}} B)^{\mathbf{r}} &:= \{\, u \mid A \to B^{\mathbf{r}} u\,\} \\
(\forall_x A)^{\mathbf{r}} &:= \{\, u \mid \forall_x A^{\mathbf{r}}(ux)\,\} \\
(\forall^{\mathrm{nc}}_x A)^{\mathbf{r}} &:= \{\, u \mid \forall_x A^{\mathbf{r}} u\,\}.
\end{aligned}$$

For c.r. predicates we have provided an n.c. $X^{\mathbf{r}}$ for predicate variables X.

$$\{\vec{x} \mid A\}^{\mathbf{r}} := \{u, \vec{x} \mid A^{\mathbf{r}}u\}.$$

Finally consider a c.r. inductive predicate

$$I := \mu_X(\forall^{\mathrm{c/nc}}_{\vec{x}_i}((A_{i\nu})_{\nu<n_i} \to^{\mathrm{c/nc}} X\vec{r}_i))_{i<k}.$$

Let $\vec{Y}$ be all predicate variables strictly positive in some $A_{i\nu}$ except X, and $\vec{\beta}$ the type variables associated with $\vec{Y}$. We may write $I(\vec{Y})$ for I and $\iota_I(\vec{\beta})$ for $\tau(I(\vec{Y}))$. We define the witnessing predicate $I^{\mathbf{r}}$ with free predicate variables $\vec{Y^{\mathbf{r}}}$ (sometimes written $I^{\mathbf{r}}(\vec{Y^{\mathbf{r}}})$) by

$$I^{\mathbf{r}} := \mu^{\mathrm{nc}}_{X^{\mathbf{r}}}(\forall_{\vec{x}_i,\vec{u}_i}((A^{\mathbf{r}}_{i\nu}u_{i\nu})_{\nu<n_i} \to X^{\mathbf{r}}(\mathrm{C}_i\vec{x}_i\vec{u}_i)\vec{r}_i))_{i<k}$$

with the understanding that

(i) $u_{i\nu}$ occurs only when $A_{i\nu}$ is c.r., and it occurs as an argument in $\mathrm{C}_i\vec{x}_i\vec{u}_i$ only if $A_{i\nu}$ is c.r. and followed by $\to$, and

(ii) only those x_{ij} with a computational $\forall_{x_{ij}}$ occur as arguments in $\mathrm{C}_i\vec{x}_i\vec{u}_i$.

Here C_i is the i-th constructor of the algebra generated from the constructor types $\tau(K_i)$ with K_i the i-th clause of I. We write $u \mathbf{\ r\ } A$ for $A^{\mathbf{r}}u$.

Example. For the even numbers

$$\mathrm{Even} := \mu_X(X0, \forall^{\mathrm{nc}}_n(Xn \to X(S(Sn))))$$

we obtain

$$\mathrm{Even}^{\mathbf{r}} := \mu^{\mathrm{nc}}_{X^{\mathbf{r}}}(X^{\mathbf{r}}00, \forall_{n,m}(X^{\mathbf{r}}mn \to X^{\mathbf{r}}(Sm)(S(Sn)))).$$

4.1.2. Extracted terms. For a derivation M of a c.r. formula A we define its *extracted term* $\mathrm{et}(M)$, of type $\tau(A)$. This definition is relative to a fixed assignment of object variables to assumption variables: to every assumption variable u^A for a c.r. formula A we assign an object variable v_u of type $\tau(A)$.

Definition (Extracted term $\mathrm{et}(M)$ of a derivation M). For derivations M^A with A n.c. let $\mathrm{et}(M^A) := \varepsilon$. Otherwise

$$\begin{aligned}
&\mathrm{et}(u^A) && := v_u^{\tau(A)} \quad (v_u^{\tau(A)} \text{ uniquely associated to } u^A),\\
&\mathrm{et}((\lambda_{u^A}M^B)^{A\to B}) && := \begin{cases} \lambda^{\tau(A)}_{v_u}\mathrm{et}(M) & \text{if } A \text{ is c.r.}\\ \mathrm{et}(M) & \text{if } A \text{ is n.c.}\end{cases}\\
&\mathrm{et}((M^{A\to B}N^A)^B) && := \begin{cases} \mathrm{et}(M)\mathrm{et}(N) & \text{if } A \text{ is c.r.}\\ \mathrm{et}(M) & \text{if } A \text{ is n.c.}\end{cases}\\
&\mathrm{et}((\lambda_{x^\rho}M^A)^{\forall_x A}) && := \lambda^\rho_x\mathrm{et}(M),\\
&\mathrm{et}((M^{\forall_x A(x)}r)^{A(r)}) && := \mathrm{et}(M)r,
\end{aligned}$$

$$\begin{aligned}
\mathrm{et}((\lambda_{u^A} M^B)^{A \to^{\mathrm{nc}} B}) &:= \mathrm{et}(M),\\
\mathrm{et}((M^{A \to^{\mathrm{nc}} B} N^A)^B) &:= \mathrm{et}(M),\\
\mathrm{et}((\lambda_{x^\rho} M^A)^{\forall^{\mathrm{nc}}_x A}) &:= \mathrm{et}(M),\\
\mathrm{et}((M^{\forall^{\mathrm{nc}}_x A(x)} r)^{A(r)}) &:= \mathrm{et}(M).
\end{aligned}$$

It remains to define extracted terms for the axioms. Consider a (c.r.) inductively defined predicate I. For its introduction and elimination axioms (3.1) and (3.2) define $\mathrm{et}(I_i^+) := \mathrm{C}_i$ and $\mathrm{et}(I^-) := \mathcal{R}$, where both the constructor C_i and the recursion operator $\mathcal{R}$ refer to the algebra ι_I associated with I.

THEOREM (Soundness). *Let M be a derivation of a c.r. formula A from assumptions $u_i \colon C_i$ $(i < n)$. Then we can derive* $\mathrm{et}(M)$ $\mathbf{r}$ A *from assumptions* v_{u_i} $\mathbf{r}$ C_i *in case C_i is c.r. and C_i otherwise.*

4.1.3. Quotient and remainder. In Section 2.1.3 we have proved by induction that every natural number n can be divided by $m + 1$ with some quotient q and remainder r, and saved the proof under the name `QR`. We extract from this proof its computational content by

```
(define eterm
 (proof-to-extracted-term (theorem-name-to-proof "QR")))
```

To display this term it is helpful to first add a variable name `p` for pairs of natural numbers and then normalize.

```
(add-var-name "p" (py "nat@@nat"))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
```

We have used `rename-variables` to obtain a more readable term. This "normalized extracted term" `neterm` is the program we are looking for. To display it we write:

```
(pp neterm)
```

The output will be:

```
[n,n0](Rec nat=>nat@@nat)n0(0@0)
     ([n1,p][if (right p<n)
                (left p@Succ right p)
                (Succ left p@0)])
```

Here `[n,n0]` denotes abstraction on the variables `n,n0`, usually written by use of the λ notation. We observe that the extracted term uses the recursion operator `Rec`. In more familiar terms, it amounts to

$$f(m, 0) = 0@0$$

$$f(m,n+1) = \begin{cases} \text{left}(f(m,n))@\text{right}(f(m,n))+1 & \text{if right}(f(m,n)) < m \\ \text{left}(f(m,n))+1@0 & \text{else} \end{cases}$$

Note that we could have also displayed the program by using the command `term-to-scheme-expr`, which produces the λ–term corresponding to the program.

To test the program we can "run" it on input 6 and 754:

```
(pp (nt (mk-term-in-app-form neterm (pt "6") (pt "754"))))
```

and obtain the result 107@5.

4.1.4. List reversal. In Section 2.2.2 we have seen an informal existence proof for list reversal. We now extract a program from the proof and normalize it as follows, using variable names `f` for unary functions on lists and `p` for pairs of lists and numbers:

```
(define eterm (proof-to-extracted-term proof))
(add-var-name "f" (py "list nat=>list nat"))
(add-var-name "p" (py "list nat@@nat"))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
```

This "normalized extracted term" `neterm` is the program we are looking for. To display it we write:

```
(pp neterm)
```

The output will be:

```
[x](Rec nat=>list nat=>list nat)x([v](Nil nat))
 ([x0,f,v]
   [if v
     (Nil nat)
     ([x1,v0][let p (cListInitLastNat v0 x1)
              (right p::f left p)])])
```

where the square brackets in `[x]` is a notation for λ-abstraction λ_x. We observe that the extracted term uses the recursion operator `Rec`, here with value type `list nat=>list nat`. Moreover, the term contains the constant `cListInitLastNat` denoting the content of the auxiliary proposition, and in the step the function defined recursively calls itself via `f`. The underlying algorithm defines an auxiliary function g by

$$\begin{aligned} g(0,v) &:= \text{Nil}, \\ g(n+1,\text{Nil}) &:= \text{Nil}, \\ g(n+1,xv) &:= \text{let } wy = xv \text{ in } y :: g(n,w) \end{aligned}$$

and gives the result by applying g to $|v|$ and v. It clearly takes quadratic time. To run this algorithm one has to normalize (via "nt") the term obtained by applying neterm to the length of a list and the list itself, and pretty print the result (via "pp"):

```
(animate "ListInitLastNat")
(animate "Id")
(pp (nt (mk-term-in-app-form
          neterm (pt "4") (pt "1::2::3::4:"))))
```

The returned value is the reverted list `4::3::2::1:`. We have made use here of a mechanism to "animate" or "deanimate" lemmata, or more precisely the constants that denote their computational content. This method can be described generally as follows. Suppose a proof of a theorem uses a lemma. Then the proof term contains just the name of the lemma, say `L`. In the term extracted from this proof we want to preserve the structure of the original proof as much as possible, and hence we use a new constant `cL` at those places where the computational content of the lemma is needed. When we want to execute the program, we have to replace the constant `cL` corresponding to a lemma `L` by the extracted program of its proof. This can be achieved by adding computation rules for `cL`. We can be rather flexible here and enable/block rewriting by using `animate/deanimate` as desired. To obtain the `let` expression in the term above, we have used implicitly the "identity lemma" `Id`: $P \to P$; its realizer has the form $\lambda_{f,x}(fx)$. If `Id` is not animated, the extracted term has the form `cId`$(\lambda_x M)N$, which is printed as [`let` x N M].

We shall later (in 5.2.1) come back to this example. It will turn out that the method of "refined A-translation" applied to a weak existence proof (of $\forall_v \tilde{\exists}_w Rvw$ rather than $\forall_v \exists_w Rvw$) together with decoration will make it possible to extract the usual linear list reversal algorithm from a proof.

4.1.5. Binary trees. In Section 2.3 we have formally proved that the nodes of a binary tree are linearly ordered by the Brouwer-Kleene ordering. We now extract a program from this proof, which returns an increasing (in the sense of the Brouwer-Kleene ordering) list of the nodes of the given tree.

```
(define eterm (proof-to-extracted-term
                (theorem-name-to-proof "ExBK")))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
```

The result is

```
[r](Rec bin=>list list boole)r(Nil boole):
 ([r0,as,r1,as0]((Cons boole)True map as)++
                ((Cons boole)False map as0)++(Nil boole):)
```

4.2. Computational content of classical proofs

Often in mathematics existence proofs are indirect, i.e., assuming that there is no solution of a problem one derives a contradiction. A good example (suggested by Yiannis Moschovakis) is Euclid's proof that the greatest common divisor of two integers is a linear combination of the two. The usual proof considers the ideal (a_1, a_2) generated by the two numbers and uses the fact the every ideal in the integers is a principal ideal. Its least positive element has a representation $|k_1a_1 - k_2a_2|$ with non-negative integers k_1, k_2. It is a common divisor of a_1 and a_2 (since otherwise the remainder of its division by a_i would be a smaller positive element of the ideal), and it is the greatest common divisor (since any common divisor of a_1 and a_2 must also be a divisor of $|k_1a_1 - k_2a_2|$).

Clearly such proofs can only implicitly provide a solution, and it is a challenge to proof theory to provide tools to access the hidden content of classical existence proofs. Such tools are indeed available: the Dialectica interpretation of Gödel (1958), and a refined form of the so-called A-translation of Friedman (1978) and Dragalin (1979).

Minlog implements both, the refined A-translation and the Dialectica interpretation. Here we only deal with the former, and discuss some examples below.

4.2.1. What is the A-translation, and why refine it?

It is a well-known result of proof theory that from a proof in intuitionistic arithmetic of a weak existence $\tilde{\exists}_m Rnm$ (with R primitive recursive) we can obtain a proof of the strong existence $\exists_m Rnm$. Hence a witness can be extracted.

Remark. This result cannot be extended to cover universally quantified predicates R as well. To see this, consider the following example (due to Kreisel). Let R be a primitive recursive relation such that $\tilde{\exists}_k Rnk$ is undecidable. Clearly – even logically –

$$\vdash \forall_n \tilde{\exists}_m \forall_k (Rnk \to Rnm)$$

but there is no computable f satisfying

$$\forall_n \forall_k (Rnk \to R(n, f(n))),$$

for then $\tilde{\exists}_k Rnk$ would be decidable: it would be true if and only if $R(n, f(n))$ holds.

The main idea which makes the translation work is that in minimal logic $\perp$ is just a propositional variable, which can be replaced throughout the proof by an appropriate formula A; in the discussion above we take $\exists_m Rnm$ for A. Let us look of what this means when we start with a derivation of

$$D \to \forall_y (G \to \perp) \to \perp.$$

The plan is to substitute $\exists_y G$ for $\bot$. This gives

$$D \to \forall_y(G \to \exists_y G) \to \exists_y G.$$

Since $\forall_y(G \to \exists_y G)$ is derivable we obtain $D \to \exists_y G$ as desired.

Unfortunately this simple argument is not quite correct. The problem is that both D and G may contain $\bot$ and hence are changed by the substitution. To obtain a constructive proof which can be used for extraction we should employ the arithmetical falsity $\mathbf{F}$ rather than the logical one, $\bot$. To repair this failure we require the following DG-property. Let $A^{\mathbf{F}}$ denote the result of substituting $\bot$ by $\mathbf{F}$ in A.

$$\begin{aligned} &D^{\mathbf{F}} \to D, \\ &(G^{\mathbf{F}} \to \bot) \to G \to \bot. \end{aligned} \tag{4.1}$$

Using (4.1) we can now correct the argument: from the given derivation of $D \to \forall_y(G \to \bot) \to \bot$ we obtain

$$D^{\mathbf{F}} \to \forall_y(G^{\mathbf{F}} \to \bot) \to \bot,$$

since $D^{\mathbf{F}} \to D$ and $(G^{\mathbf{F}} \to \bot) \to G \to \bot$. Substituting $\bot$ by $\exists_y G^{\mathbf{F}}$ gives

$$D^{\mathbf{F}} \to \forall_y(G^{\mathbf{F}} \to \exists_y G^{\mathbf{F}}) \to \exists_y G^{\mathbf{F}}.$$

Since $\forall_y(G^{\mathbf{F}} \to \exists_y G^{\mathbf{F}})$ is derivable we obtain $D^{\mathbf{F}} \to \exists_y G^{\mathbf{F}}$ as desired.

Therefore we need to pick our assumptions D and goal formulas G from appropriately chosen sets $\mathcal{D}$ and $\mathcal{G}$ which guarantee the DG-property (4.1).

An easy way to achieve this is to replace in D and G every atomic formula P different from $\bot$ by its double negation $(P \to \bot) \to \bot$. This corresponds to the original A-translation of Friedman (1978). However, then the computational content of the resulting constructive proof is unnecessarily complex, since each occurrence of $\bot$ gets replaced by the c.r. formula $\exists_y G^{\mathbf{F}}$. To see this let us look at an example.

Example. Let $f\colon \mathbb{N} \to \mathbb{N}$ be an unbounded function with $f(0) = 0$. Then we want to prove

$$\forall_n \tilde{\exists}_m (f(m) \le n < f(m+1)).$$

If e.g. $f(m) = m^2$, then this formula expresses the existence of an integer square root $m := [\sqrt{n}]$ for any n. More formally, we want to prove

$$\forall_n(\forall_m(\neg(n < f(m)) \to n < f(m+1) \to \bot) \to \bot)$$

from the assumptions

$$\forall_n \neg(n < f(0)), \qquad \forall_n(n < f(g(n))).$$

We expressed $f(m) \le n$ by $\neg(n < f(m))$ and $f(0) = 0$ by $\forall_n \neg(n < f(0))$ to keep the formal proof as simple as possible. To guarantee the DG-property we now double negate every atomic formula different from $\bot$. Since both $n < f(m)$ and $n < f(0)$ already appear negated we can for simplicity omit their double negation. Thus we have to prove

$$\forall_n(\forall_m(\neg(n < f(m)) \to \neg\neg(n < f(m+1)) \to \bot) \to \bot) \tag{4.2}$$

from the assumptions

$$v_1 \colon \forall_n \neg(n < f(0)),, \qquad v_2 \colon \forall_n \neg\neg(n < f(g(n))).$$

Now let us prove (4.2). Let n be given and assume

$$u \colon \forall_m(\neg(n < f(m)) \to \neg\neg(n < f(m+1)) \to \bot).$$

We have to show $\bot$. From v_1 and u we inductively get $\forall_m \neg(n < f(m))$. For $m := g(n)$ this yields a contradiction to v_2.

In Minlog, this proof is implemented as follows, after loading `nat.scm`:

```
(add-var-name "f" "g" (py "nat=>nat"))

(set-goal "all f,g,n(
 all n(n<f 0 -> bot) -> all n((n<f(g n) -> bot) -> bot) ->
 excl m((n<f m -> bot) ! ((n<f(m+1) -> bot) -> bot)))")
(assume "f" "g" "n" "v1" "v2" "u")
(assert "all m(n<f m -> bot)")
 (ind)
 (use "v1")
 (assume "m" "n<f m -> bot" "n<f(m+1)")
 (use "u" (pt "m"))
 (use "n<f m -> bot")
 (assume "n<f(m+1) -> bot")
 (use-with "n<f(m+1) -> bot" "n<f(m+1)")
(assume "Assertion")
(use "v2" (pt "n"))
(use "Assertion")
;; Proof finished.
(save "IntSqRtNegNeg")
```

We can extract its computational content as follows

```
(define eterm (proof-to-extracted-term
                (theorem-name-to-proof "IntSqRtNegNeg")))
(define neterm (rename-variables (nt eterm)))
```

Pretty-printing this normalized extracted term via `(pp neterm)` yields

```
[f,f0,n,(nat=>alpha33),(nat=>alpha33=>alpha33),
 (nat=>alpha33=>(alpha33=>alpha33)=>alpha33)]
 (nat=>alpha33=>alpha33)n
```

```
((Rec nat=>alpha33)(f0 n)((nat=>alpha33)n)
 ([n0,alpha33]
   (nat=>alpha33=>(alpha33=>alpha33)=>alpha33)n0 alpha33
   ([alpha33_0]alpha33_0)))
```

Here alpha33 is a type variable expressing for the (unknown) computational content of $\bot$; after substituting $\exists_m(f(m) \le n < f(m+1))$ for $\bot$ it will become nat.

Still, this term cleary is in need of simplification; its complexity derives from the (as will turn out mostly) unnecessary usage of double negations.

Let us now see how we can eliminate unnecessary double negations. To this end we define certain sets $\mathcal{D}$ and $\mathcal{G}$ of formulas which ensure that their elements $D \in \mathcal{D}$ and $G \in \mathcal{G}$ satisfy the DG-property (4.1).

4.2.2. Definite and goal formulas. We simultaneously inductively define the classes $\mathcal{D}$ of definite formulas, $\mathcal{G}$ of goal formulas, $\mathcal{R}$ of relevant definite formulas and $\mathcal{I}$ of irrelevant goal formulas. Let D, G, R, I range over $\mathcal{D}$, $\mathcal{G}$, $\mathcal{R}$, $\mathcal{I}$, respectively, P over prime formulas distinct from $\bot$, and D_0 over quantifier-free formulas in $\mathcal{D}$.

$\mathcal{D}$, $\mathcal{G}$, $\mathcal{R}$ and $\mathcal{I}$ are generated by the clauses

(a) $R,\ P,\ I \to D,\ \forall_x D \in \mathcal{D}$.
(b) $I,\ \bot,\ R \to G,\ D_0 \to G \in \mathcal{G}$.
(c) $\bot,\ G \to R,\ \forall_x R \in \mathcal{R}$.
(d) $P,\ D \to I,\ \forall_x I \in \mathcal{I}$.

Let $A^{\mathbf{F}}$ denote $A[\bot := \mathbf{F}]$, and $\neg A$, $\neg_\bot A$ abbreviate $A \to \mathbf{F}$, $A \to \bot$.

Lemma. *We have derivations from* $\mathbf{F} \to \bot$ *and* $\mathbf{F} \to P$ *of*

$$D^{\mathbf{F}} \to D, \tag{4.3}$$

$$G \to \neg_\bot\neg_\bot G^{\mathbf{F}}, \tag{4.4}$$

$$\neg_\bot\neg R^{\mathbf{F}} \to R, \tag{4.5}$$

$$I \to I^{\mathbf{F}}. \tag{4.6}$$

This result is due to Ishihara (2000) and has been (independently) rediscovered in Berger et al. (2002). A detailed proof is in Schwichtenberg and Wainer (2012, pp.364-367).

We give some examples of definite and goal formulas. Keep in mind that $\mathcal{R} \subseteq \mathcal{D}$ and $\mathcal{I} \subseteq \mathcal{G}$.

- $P \in \mathcal{D} \cap \mathcal{I}$.
- $\bot \in \mathcal{R} \cap \mathcal{G}$.
- $P \to \bot \in \mathcal{R} \cap \mathcal{G}$.
- $(P \to \bot) \to \bot \in \mathcal{R} \cap \mathcal{G}$.

Lemma. *$C \in \mathcal{D} \cap \mathcal{G}$ for C quantifier-free such that no implication in C has $\bot$ as its final conclusion, and $C \in \mathcal{R}$ ($\in \mathcal{I}$) if and only if $\bot$ is (is not) the final conclusion of C.*

Proof. The cases P and $\bot$ are clear. *Case* $C \to R$. Since $C \in \mathcal{G}$ we have $C \to R \in \mathcal{R}$, and since $C \in \mathcal{D}$ and $R \in \mathcal{G}$ we have $C \to R \in \mathcal{G}$.

Case $C \to I$. Since $C \in \mathcal{I}$ (because $\bot$ is not the final conclusion of C) and $I \in \mathcal{D}$ we have $C \to I \in \mathcal{D}$, and since $C \in \mathcal{D}$ we have $C \to I \in \mathcal{I}$. □

Note that a further condition on C except being quantifier-free is needed since for instance $\bot \to P \notin \mathcal{D}$.

Example. We now take up the previous example again. More formally, we can prove

$$\forall_n \tilde{\exists}_m (\neg(n < f(m)) \wedge n < f(m+1)) \tag{4.7}$$

from the assumptions

$$v_1 \colon \forall_n \neg(n < f(0)), \qquad v_2 \colon \forall_n (n < f(g(n))).$$

Here $<^{\mathbb{N}\to\mathbb{N}\to\mathbb{B}}$ is the characteristic function of the natural ordering of the natural numbers. We expressed $f(m) \leq n$ by $\neg(n < f(m))$ and $f(0) = 0$ by $\forall_n \neg(n < f(0))$ to keep the formal proof as simple as possible. In order to have purely universal assumptions we had to express the unboundedness of f by a witnessing function g.

Now let us prove (4.7). Let n be given and assume

$$u \colon \forall_m (\neg(n < f(m)) \to n < f(m+1) \to \bot).$$

We have to show $\bot$. From v_1 and u we inductively get $\forall_m \neg(n < f(m))$. For $m := g(n)$ this yields a contradiction to v_2.

In Minlog, this proof is implemented as follows, after loading `nat.scm`:

```
(add-var-name "f" "g" (py "nat=>nat"))

(set-goal "all f,g,n(
 all n(n<f 0 -> bot) -> all n n<f(g n) ->
 excl m((n<f m -> bot) ! n<f(m+1)))")
(assume "f" "g" "n" "v1" "v2" "u")
(assert "all m(n<f m -> bot)")
 (ind)
 (use "v1")
 (use "u")
(assume "Assertion")
(use-with "Assertion" (pt "g n") "?")
(use "v2")
;; Proof finished.
```

```
(save "IntSqRt")
```

We save the proof and normalize it.

```
(define proof (theorem-name-to-proof "IntSqRt"))
(define nproof (np proof))
```

Application of the refined A-translation followed by term extraction and normalization gives us

```
(define eterm
 (atr-min-excl-proof-to-structured-extracted-term nproof))

(define neterm (nt eterm))
(pp (rename-variables neterm))

;; [f,f0,n](Rec nat=>nat)(f0 n)0([n0,n1][if (n<f n0) n1 n0])
```

We can test this term by typing

```
(define sqr (pt "[n]n*n"))

(pp (nt (mk-term-in-app-form neterm sqr (pt "Succ")
 (pt "15")))) ;"3"
(pp (nt (mk-term-in-app-form neterm sqr (pt "Succ")
 (pt "16")))) ;"4"
```

4.2.3. List reversal, weak form. We first give an informal *weak* existence proof for list reversal. Recall that the *weak* (or "classical") existential quantifier is defined by

$$\tilde{\exists}_x A := \neg\forall_x \neg A.$$

The proof is similar to the one given in 2.2.2. From the clauses

$$\begin{aligned}\text{InitR}\colon & R(\text{Nil}, \text{Nil}),\\ \text{GenR}\colon & \forall_{v,w,x}(Rvw \to R(vx, xw)).\end{aligned}$$

we prove

$$\forall_v \tilde{\exists}_w Rvw \qquad (\,:= \forall_v(\forall_w(Rvw \to \bot) \to \bot)). \tag{4.8}$$

Fix R, v and assume `InitR`, `GenR` and the "false" assumption $u\colon \forall_w \neg Rvw$; we need to derive a contradiction. To this end we prove that all initial segments of v are non-revertible, which contradicts `InitR`. More precisely, from u and `GenR` we prove

$$\forall_{v_2} A(v_2) \quad \text{with } A(v_2) := \forall_{v_1}(v_1 v_2 = v \to \forall_w \neg R v_1 w)$$

by induction on v_2. For $v_2 = \text{Nil}$ this follows from $u_0\colon v_1 \text{ Nil} = v$ and our ("false") assumption u. For the step case, assume $u_1\colon v_1(xv_2) = v$, fix w and assume further $u_2\colon Rv_1w$. We must derive a contradiction. We use the induction hypotheses with v_1x and xw to obtain the desidered contradiction. This requires us to prove (i) $(v_1x)v_2 = v$ and (ii) $R(v_1x, xw)$. But (i) follows from u_1 using properties of the append function, and (ii) follows from u_2 using `GenR`.

We formalize this proof, to prepare it for the refined A-translation. The following lemmata will be used:

$$\begin{aligned}&\text{Compat}'\colon\ \forall^{\text{nc}}_{v,w}(v =^{\text{d}} w \to Xw \to Xv),\\ &\text{EqToEqD}\colon \forall_{v,w}(v = w \to v =^{\text{d}} w).\end{aligned}$$

The proof term is

$$\begin{aligned}M :=&\lambda_{R,v}\lambda_{u_{\text{InitR}}}\lambda_{u_{\text{GenR}}}\lambda_u^{\forall_w\neg Rvw}(\\ &\quad\text{Ind}_{v_2,A(v_2)}vRvM_{\text{Base}}M_{\text{Step}}\ \text{Nil}\ \text{Truth}^{\text{Nil}\,v=v}\ \text{Nil}\ u_{\text{InitR}})\end{aligned}$$

with

$$\begin{aligned}M_{\text{Base}} :=&\ \lambda_{v_1}\lambda_{u_0}^{v_1\text{Nil}=v}(\\ &\quad\text{Compat}'\,\{\,v \mid \forall_w\neg Rvw\,\}\,R\,v\,v_1\,v\,(\text{EqToEqD}\,v_1vu_0)u),\\ M_{\text{Step}} :=&\ \lambda_{x,v_2}\lambda_{u_0}^{A(v_2)}\lambda_{v_1}\lambda_{u_1}^{v_1(xv_2)=v}\lambda_w\lambda_{u_2}^{Rv_1w}(\\ &\quad u_0(v_1x)u_1(xw)(u_{\text{GenR}}v_1wxu_2)).\end{aligned}$$

We now have a proof M of $\forall_v\tilde{\exists}_wRvw$ from InitR: D_1 and GenR: D_2, with $D_1 := R(\text{Nil},\text{Nil})$ and $D_2 := \forall_{v,w,x}(Rvw \to R(vx, xw))$. Using the refined A-translation we can replace $\bot$ throughout by $\exists_wRvw$. The end formula $\tilde{\exists}_wRvw := \neg\forall_w\neg Rvw := \forall_w(Rvw \to \bot) \to \bot$ is turned into $\forall_w(Rvw \to \exists_wRvw) \to \exists_wRvw$. Since its premise is an instance of existence introduction we obtain a derivation $M^{\exists}$ of $\exists_wRvw$. Moreover, in this case neither the D_i nor any of the axioms used involves $\bot$ in its uninstantiated formulas, and hence the correctness of the proof is not affected by the substitution. The term `neterm` extracted in Minlog from a formalization of the proof above is

```
[R,v](Rec list nat=>list nat=>list nat=>list nat)v([v0,v1]v1)
 ([x,v0,g,v1,v2]g(v1++x:)(x::v2)) (Nil nat) (Nil nat)
```

with `g` a variable for binary functions on lists. In fact, the underlying algorithm defines an auxiliary function h by

$$h(\text{Nil}, v_1, v_2) := v_2, \qquad h(xv, v_1, v_2) := h(v, v_1x, xv_2)$$

and gives the result by applying h to the original list and twice Nil.

Notice that the second argument of h is not needed. However, its presence makes the algorithm quadratic rather than linear, because in each recursion step v_1x is computed, and the list append function is defined by recursion on its first argument. We will be able to get rid of this superfluous second argument by decorating the proof. It will turn out that in the proof (by induction on v_2) of the formula $A(v_2) := \forall_{v_1}(v_1 v_2 = v \to \forall_w \neg R v_1 w))$, the variable v_1 is not used computationally. Hence, in the decorated version of the proof, we can use $\forall^{\mathrm{nc}}_{v_1}$.

CHAPTER 5

Decorating proofs

In this chapter we are interested in "fine-tuning" the computational content of proofs, by inserting decorations. Here is an example (due to Constable) of why this is of interest. Suppose that in a proof M of a formula C we have made use of a case distinction based on an auxiliary lemma stating a disjunction, say $L\colon A \vee B$. Then the extract $\mathrm{et}(M)$ will contain the extract $\mathrm{et}(L)$ of the proof of the auxiliary lemma, which may be large. Now suppose further that in the proof M of C the only computationally relevant use of the lemma was which one of the two alternatives holds true, A or B. We can express this fact by using a weakened form of the lemma instead: $L'\colon A \vee^{\mathrm{u}} B$. Since the extract $\mathrm{et}(L')$ is a boolean, the extract of the modified proof has been "purified" in the sense that the (possibly large) extract $\mathrm{et}(L)$ has disappeared.

In Section 5.1 we consider the question of "optimal" decorations of proofs: suppose we are given an undecorated proof, and a decoration of its end formula. The task then is to find a decoration of the whole proof (including a further decoration of its end formula) in such a way that any other decoration "extends" this one. Here "extends" just means that some connectives have been changed into their more informative versions, disregarding polarities. We show that such an optimal decoration exists, and give an algorithm to construct it.

We then consider some applications: list reversal, computing the Fibonacci numbers in continuation passing style, and finally the Maximal Scoring Segment (MSS) algorithm. For instance in the latter case, directly deriving such an algorithm from a proof leads to quadratic complexity. We will see that the (automatically found) optimal decoration of this proof results in a linear extracted algorithm.

5.1. Decoration algorithm

We denote the *sequent* of a proof M by $\mathrm{Seq}(M)$; it consists of its *context* and *end formula.*

The *proof pattern* $\mathrm{P}(M)$ of a proof M is the result of marking in c.r. parts of M (i.e., not above a n.c. formula) all occurrences of implications and universal quantifiers as non-computational, except the "uninstantiated" formulas of axioms and theorems. For instance, the induction axiom for $\mathbf{N}$ consists of the uninstantiated formula $\forall_n(X0 \to \forall_n(Xn \to X(Sn)) \to Xn^{\mathbf{N}})$ with a predicate variable X and a predicate substitution $X \mapsto \{\, x \mid A(x) \,\}$. Notice that a proof pattern in most cases is not a correct proof, because at axioms formulas may not fit.

We say that a formula D *extends* C if D is obtained from C by changing some (possibly zero) of its occurrences of non-computational implications and universal quantifiers into their computational variants $\to$ and $\forall$.

A proof N *extends* M if (i) N and M are the same up to variants of implications and universal quantifiers in their formulas, and (ii) every formula in c.r. parts of M is extended by the corresponding one in N. Every proof M whose proof pattern $\mathrm{P}(M)$ is U is called a *decoration* of U.

Notice that if a proof N extends another one M, then $\mathrm{FV}(\mathrm{et}(N))$ is essentially (that is, up to extensions of assumption formulas) a superset of $\mathrm{FV}(\mathrm{et}(M))$. This can be proven by induction on N.

In the sequel we assume that every axiom has the property that for every extension of its formula we can find a further extension which is an instance of an axiom, and which is the least one under all further extensions that are instances of axioms. This property clearly holds for axioms whose uninstantiated formula only has $\to$ and $\forall$, for instance induction. However, in $\forall_n(A(0) \to \forall_n(A(n) \to A(Sn)) \to A(n^{\mathbf{N}}))$ the given extension of the four A's might be different. One needs to pick their "least upper bound" as further extension.

We will define a *decoration algorithm*, assigning to every proof pattern U and every extension of its sequent an "optimal" decoration M_∞ of U, which further extends the given extension of its sequent.

THEOREM. *Under the assumption above, for every proof pattern U and every extension of its sequent* $\mathrm{Seq}(U)$ *we can find a decoration M_∞ of U such that*

(a) $\mathrm{Seq}(M_\infty)$ *extends the given extension of* $\mathrm{Seq}(U)$*, and*
(b) M_∞ *is* optimal *in the sense that any other decoration M of U whose sequent* $\mathrm{Seq}(M)$ *extends the given extension of* $\mathrm{Seq}(U)$ *has the property that M also extends M_∞.*

PROOF. By induction on derivations. It suffices to consider derivations with a c.r. endformula. For axioms the validity of the claim was assumed, and for assumption variables it is clear.

Case $(\to^{\mathrm{nc}})^+$. Consider the proof pattern

$$\dfrac{\begin{array}{c}\Gamma, u\colon A\\ \mid U\\ B\end{array}}{A \to^{\mathrm{nc}} B}\,(\to^{\mathrm{nc}})^+, u$$

with a given extension $\Delta \Rightarrow C \to^{\mathrm{nc}} D$ or $\Delta \Rightarrow C \to D$ of its sequent $\Gamma \Rightarrow A \to^{\mathrm{nc}} B$. Applying the induction hypothesis for U with sequent $\Delta, C \Rightarrow D$, one obtains a decoration M_∞ of U whose sequent $\Delta_1, C_1 \Rightarrow D_1$ extends $\Delta, C \Rightarrow D$. Now apply $(\to^{\mathrm{nc}})^+$ in case the given extension is $\Delta \Rightarrow C \to^{\mathrm{nc}} D$ and $x_u \notin \mathrm{FV}(\mathrm{et}(M_\infty))$, and $\to^+$ otherwise.

For (b) consider a decoration $\lambda_u M$ of $\lambda_u U$ whose sequent extends the given extended sequent $\Delta \Rightarrow C \to^{\mathrm{nc}} D$ or $\Delta \Rightarrow C \to D$. Clearly the sequent $\mathrm{Seq}(M)$ of its premise extends $\Delta, C \Rightarrow D$. Then M extends M_∞ by induction hypothesis for U. If $\lambda_u M$ derives a non-computational implication then the given extended sequent must be of the form $\Delta \Rightarrow C \to^{\mathrm{nc}} D$ and $x_u \notin \mathrm{FV}(\mathrm{et}(M))$, hence $x_u \notin \mathrm{FV}(\mathrm{et}(M_\infty))$. But then by construction we have applied $(\to^{\mathrm{nc}})^+$ to obtain $\lambda_u M_\infty$. Hence $\lambda_u M$ extends $\lambda_u M_\infty$. If $\lambda_u M$ does not derive a non-computational implication, the claim follows immediately.

Case $(\to^{\mathrm{nc}})^-$. Consider a proof pattern

$$\dfrac{\begin{array}{c}\Phi, \Gamma\\ \mid U\\ A \to^{\mathrm{nc}} B\end{array} \qquad \begin{array}{c}\Gamma, \Psi\\ \mid V\\ A\end{array}}{B}\,(\to^{\mathrm{nc}})^-$$

We are given an extension $\Pi, \Delta, \Sigma \Rightarrow D$ of $\Phi, \Gamma, \Psi \Rightarrow B$. Then we proceed in alternating steps, applying the induction hypothesis to U and V.

(1) The induction hypothesis for U for the extension $\Pi, \Delta \Rightarrow A \to^{\mathrm{nc}} D$ of its sequent gives a decoration M_1 of U whose sequent $\Pi_1, \Delta_1 \Rightarrow C_1 \to^{\mathrm{c/nc}} D_1$ extends $\Pi, \Delta \Rightarrow A \to^{\mathrm{nc}} D$, where $\to^{\mathrm{c/nc}}$ means $\to$ or $\to^{\mathrm{nc}}$. This already suffices if A is n.c., since then the extension $\Delta_1, \Sigma \Rightarrow C_1$ of V is a correct proof (recall that in n.c. parts of a proof decorations of implications and universal quantifiers can be ignored). If A is c.r.:

(2) The induction hypothesis for V for the extension $\Delta_1, \Sigma \Rightarrow C_1$ of its sequent gives a decoration N_2 of V whose sequent $\Delta_2, \Sigma_2 \Rightarrow C_2$ extends $\Delta_1, \Sigma \Rightarrow C_1$.

(3) The induction hypothesis for U for the extension $\Pi_1, \Delta_2 \Rightarrow C_2 \to^{\mathrm{c/nc}} D_1$ of its sequent gives a decoration M_3 of U whose sequent $\Pi_3, \Delta_3 \Rightarrow C_3 \to^{\mathrm{c/nc}} D_3$ extends $\Pi_1, \Delta_2 \Rightarrow C_2 \to^{\mathrm{c/nc}} D_1$.

(4) The induction hypothesis for V for the extension $\Delta_3, \Sigma_2 \Rightarrow C_3$ of its sequent gives a decoration N_4 of V whose sequent $\Delta_4, \Sigma_4 \Rightarrow C_4$ extends

$\Delta_3, \Sigma_2 \Rightarrow C_3$. This process is repeated until no further proper extension of Δ_i, C_i is returned. Such a situation will always be reached since there is a maximal extension, where all connectives are maximally decorated. But then we easily obtain (a): Assume that in (4) we have $\Delta_4 = \Delta_3$ and $C_4 = C_3$. Then the decoration

$$\dfrac{\begin{array}{c}\Pi_3, \Delta_3 \\ \mid M_3 \\ C_3 \to^{\mathrm{c/nc}} D_3\end{array} \qquad \begin{array}{c}\Delta_4, \Sigma_4 \\ \mid N_4 \\ C_4\end{array}}{D_3} \to^-$$

of UV derives a sequent $\Pi_3, \Delta_3, \Sigma_4 \Rightarrow D_3$ extending $\Pi, \Delta, \Sigma \Rightarrow D$.

For (b) we need to consider a decoration MN of UV whose sequent $\mathrm{Seq}(MN)$ extends the given extension $\Pi, \Delta, \Sigma \Rightarrow D$ of $\Phi, \Gamma, \Psi \Rightarrow B$. We must show that MN extends M_3N_4. To this end we go through the alternating steps again.

(1) Since the sequent $\mathrm{Seq}(M)$ extends $\Pi, \Delta \Rightarrow A \to^{\mathrm{nc}} D$, the induction hypothesis for U for the extension $\Delta \Rightarrow A \to^{\mathrm{nc}} D$ of its sequent ensures that M extends M_1.

(2) Since then the sequent $\mathrm{Seq}(N)$ extends $\Delta_1, \Sigma \Rightarrow C_1$, the induction hypothesis for V for the extension $\Delta_1, \Sigma \Rightarrow C_1$ of its sequent ensures that N extends N_2.

(3) Therefore $\mathrm{Seq}(M)$ extends the sequent $\Pi_1, \Delta_2 \Rightarrow C_2 \to^{\mathrm{c/nc}} D_1$, and the induction hypothesis for U for the extension $\Pi_1, \Delta_2 \Rightarrow C_2 \to^{\mathrm{c/nc}} D_1$ of U's sequent ensures that M extends M_3.

(4) Therefore $\mathrm{Seq}(N)$ extends $\Delta_3, \Sigma_2 \Rightarrow C_3$, and induction hypothesis for V for the extension $\Delta_3, \Sigma_2 \Rightarrow C_3$ of V's sequent ensures that N also extends N_4.

But since $\Delta_4 = \Delta_3$ and $C_4 = C_3$ by assumption, MN extends the decoration M_3N_4 of UV constructed above.

Case $(\forall^{\mathrm{nc}})^+$. Consider a proof pattern

$$\dfrac{\begin{array}{c}\Gamma \\ \mid U \\ A\end{array}}{\forall^{\mathrm{nc}}_x A} (\forall^{\mathrm{nc}})^+$$

with a given extension $\Delta \Rightarrow \forall^{\mathrm{nc}}_x C$ or $\Delta \Rightarrow \forall_x C$ of its sequent. Applying the induction hypothesis for U with sequent $\Delta \Rightarrow C$, one obtains a decoration M_∞ of U whose sequent $\Delta_1 \Rightarrow C_1$ extends $\Delta \Rightarrow C$. Now apply $(\forall^{\mathrm{nc}})^+$ in case the given extension is $\Delta \Rightarrow \forall^{\mathrm{nc}}_x C$ and $x \notin \mathrm{FV}(\mathrm{et}(M_\infty))$, and $\forall^+$ otherwise.

For (b) consider a decoration $\lambda_x M$ of $\lambda_x U$ whose sequent extends the given extended sequent $\Delta \Rightarrow \forall^{\mathrm{nc}}_x C$ or $\Delta \Rightarrow \forall_x C$. Clearly the sequent $\mathrm{Seq}(M)$

of its premise extends $\Delta \Rightarrow C$. Then M extends M_∞ by induction hypothesis for U. If $\lambda_x M$ derives a non-computational generalization, then the given extended sequent must be of the form $\Delta \Rightarrow \forall^{\mathrm{nc}}_x C$ and $x \notin \mathrm{FV}(\mathrm{et}(M))$, hence $x \notin \mathrm{FV}(\mathrm{et}(M_\infty))$ (by the remark above). But then by construction we have applied $(\forall^{\mathrm{nc}})^+$ to obtain $\lambda_x M_\infty$. Hence $\lambda_x M$ extends $\lambda_x M_\infty$. If $\lambda_x M$ does not derive a non-computational generalization, the claim follows immediately.

Case $(\forall^{\mathrm{nc}})^-$. Consider a proof pattern

$$\dfrac{\begin{matrix}\Gamma \\ \mid U \\ \forall^{\mathrm{nc}}_x A(x)\end{matrix} \qquad r}{A(r)}\,(\forall^{\mathrm{nc}})^-$$

and let $\Delta \Rightarrow C(r)$ be any extension of its sequent $\Gamma \Rightarrow A(r)$. The induction hypothesis for U for the extension $\Delta \Rightarrow \forall^{\mathrm{nc}}_x C(x)$ produces a decoration M_∞ of U whose sequent extends $\Delta \Rightarrow \forall^{\mathrm{nc}}_x C(x)$. Then apply $(\forall^{\mathrm{nc}})^-$ or $\forall^-$, whichever is appropriate, to obtain the required $M_\infty r$.

For (b) consider a decoration Mr of Ur whose sequent $\mathrm{Seq}(Mr)$ extends the given extension $\Delta \Rightarrow C(r)$ of $\Gamma \Rightarrow A(r)$. Then M extends M_∞ by induction hypothesis for U, and hence Mr extends $M_\infty r$. $\square$

We illustrate the effects of decoration on a simple example involving implications. Consider $A \to B \to A$ with the trivial proof $M := \lambda^A_{u_1}\lambda^B_{u_2}u_1$. Clearly only the first implication must transport possible computational content. To "discover" this by means of the decoration algorithm we specify as extension of $\mathrm{Seq}(\mathrm{P}(M))$ the formula $A \to^{\mathrm{nc}} B \to^{\mathrm{nc}} A$. The algorithm then returns a proof of $A \to B \to^{\mathrm{nc}} A$.

5.2. Applications

5.2.1. List reversal: decoration of the weak existence proof. Recall the weak (or classical) existence proof for list reversal in 4.2.3. Let us now apply the general method of decorating proofs to this example. First we present our proof in more detail, particularly by writing proof trees with formulas. The decoration algorithm then is applied to its proof pattern with the sequent consisting of the context InitR: $R(\mathrm{Nil},\mathrm{Nil})$ and GenR: $\forall_{v,w,x}(Rvw \to R(vx, xw))$ and the end formula $\forall_v \exists_w Rvw$.

Rather than describing the algorithm step by step we only display the end result. Among the axioms used, the only ones in c.r. parts are list induction, CompatRev and $\exists^+$.

$$\begin{array}{ll}\mathrm{CompatRev}\colon & \forall^{\mathrm{nc}}_{R,v,v_1,v_2}(v_1 =^{\mathrm{d}} v_2 \to \forall_w \neg^{\exists} R v_2 w \to \forall_w \neg^{\exists} R v_1 w),\\ \exists^+\colon & \forall^{\mathrm{nc}}_{R,v}\forall_w(Rvw \to \exists_w Rvw)\end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{CompatRev}\quad R\quad v\quad v_1\quad v}{v_1 =^{\mathrm{d}} v \to \forall_w \neg^{\exists} Rvw \to \forall_w \neg^{\exists} Rv_1w} \qquad \begin{array}{c}[u_1\colon v_1\,\mathrm{Nil} = v]\\ \mid N_1\\ v_1\,\mathrm{Nil} =^{\mathrm{d}} v\end{array}}{\forall_w \neg^{\exists} Rvw \to \forall_w \neg^{\exists} Rv_1w} \qquad \dfrac{\exists^+ \quad R \quad v}{\forall_w \neg^{\exists} Rvw}}{\forall_w \neg^{\exists} Rv_1w}}{v_1\,\mathrm{Nil} = v \to \forall_w \neg^{\exists} Rv_1w}\to^+ u_1}{\forall^{\mathrm{nc}}_{v_1}(v_1\,\mathrm{Nil} = v \to \forall_w \neg^{\exists} Rv_1w) \quad (= A(\mathrm{Nil}))}$$

FIGURE 1. Decorated base derivation M_{Base}

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[u_0\colon A(v_2)] \quad v_1x}{(v_1x)v_2{=}v \to \forall_w \neg^{\exists} R(v_1x,w)} \quad [u_1\colon v_1(xv_2){=}v]}{\forall_w \neg^{\exists} R(v_1x,w)} \qquad xw}{\neg^{\exists} R(v_1x,xw)} \qquad \begin{array}{c}[u_2\colon Rv_1w]\\ \mid N_2\\ R(v_1x,xw)\end{array}}{\exists_w Rvw}}{\neg^{\exists} Rv_1w}\to^+ u_2}{\forall_w \neg^{\exists} Rv_1w}}{v_1(xv_2){=}v \to \forall_w \neg^{\exists} Rv_1w}\to^+ u_1}{\forall^{\mathrm{nc}}_{v_1}(v_1(xv_2){=}v \to \forall_w \neg^{\exists} Rv_1w) \;\; ({=}A(xv_2))}}{A(v_2) \to A(xv_2)}\to^+ u_0}{\forall_{x,v_2}(A(v_2) \to A(xv_2))}$$

FIGURE 2. Decorated step derivation M_{Step}

with $A(v_2) := \forall^{\mathrm{nc}}_{v_1}(v_1v_2{=}v \to \forall_w \neg^{\exists} Rv_1w)$ and $\neg^{\exists} Rv_1w := Rv_1w \to \exists_w Rvw$. $M^{\exists}_{\mathrm{Base}}$ is the derivation in Figure 1 where N_1 is a derivation involving EqToEqD: $\forall_{v,w}(v = w \to v =^{\mathrm{d}} w)$, with a free assumption $u_1\colon v_1\,\mathrm{Nil} = v$. $M^{\exists}_{\mathrm{Step}}$ is the derivation in Figure 2 where N_2 is a derivation involving GenR, with the free assumption $u_2\colon Rv_1w$. From these we obtain the final decorated derivation of $\exists_w Rvw$ by applying

$$\mathrm{Ind}\colon \forall^{\mathrm{nc}}_{v,R}\forall_w(A(\mathrm{Nil}) \to \forall_{x,v_2}(A(v_2) \to A(xv_2)) \to A(w))$$

to v, R, v, M_{Base}, M_{Step}, Nil, Truth, Nil and InitR.

The extracted term `neterm` then is

```
[R,v](Rec list nat=>list nat=>list nat)v([v0]v0)
  ([x,v0,f,v1]f(x::v1))(Nil nat)
```

with `f` a variable for unary functions on lists. To run this algorithm one has to normalize the term obtained by applying `neterm` to a list:

```
(pp (nt (mk-term-in-app-form neterm (pt "1::2::3::4:"))))
```

The returned value is the reverted list `4::3::2::1:`. This time, the underlying algorithm defines an auxiliary function g by

$$g(\mathrm{Nil}, w) := w, \qquad g(x :: v, w) := g(v, x :: w)$$

and gives the result by applying g to the original list and Nil. In conclusion, we have obtained (by machine extraction from an automated decoration of a weak existence proof) the standard linear algorithm for list reversal, with its use of an accumulator.

5.2.2. Fibonacci numbers. An application of decoration occurs when one derives double induction

$$\forall_n(Qn \to Q(Sn) \to Q(S(Sn))) \to \forall_n(Q0 \to Q1 \to Qn)$$

in *continuation passing style*, i.e., not directly, but using as an intermediate assertion (proved by induction)

$$\forall_{n,m}((Qn \to Q(Sn) \to Q(n+m)) \to Q0 \to Q1 \to Q(n+m)).$$

After decoration, the formula becomes

$$\forall_n\forall^{\mathrm{nc}}_m((Qn \to Q(Sn) \to Q(n+m)) \to Q0 \to Q1 \to Q(n+m)).$$

This can be applied to obtain a continuation based tail recursive definition of the Fibonacci function, from a proof of its totality. Let G be the (n.c.) graph of the Fibonacci function, defined by the clauses

$$G(0,0), \quad G(1,1),$$
$$\forall_{n,v,w}(G(n,v) \to G(Sn,w) \to G(S(Sn), v+w)).$$

From these assumptions one can easily derive

$$\forall_n\exists_v G(n,v),$$

using double induction (proved in continuation passing style). The term extracted from this proof is

```
[n](Rec nat=>nat=>(nat=>nat=>nat)=>nat=>nat=>nat)n([n0,k]k)
 ([n0,p,n1,k]p(Succ n1)([n2,n3]k n3(n2+n3)))
```

applied to `0`, `([n0,n1]n0)`, `0` and `1`. An unclean aspect of this term is that the recursion operator has value type

```
nat=>(nat=>nat=>nat)=>nat=>nat=>nat
```

rather than `(nat=>nat=>nat)=>nat=>nat=>nat`, which would correspond to an iteration. However, we can repair this by decoration. After (automatic) decoration of the proof, the extracted term becomes

```
[n](Rec nat=>(nat=>nat=>nat)=>nat=>nat=>nat)n([k]k)
 ([n0,p,k]p([n1,n2]k n2(n1+n2)))
```

applied to `([n0,n1]n0)`, `0` and `1` (`k`, `p` are variables of type `nat=>nat=>nat` and `(nat=>nat=>nat)=>nat=>nat=>nat`, respectively.) This is iteration in continuation passing style: the functional F recursively defined by

$$\begin{aligned} F(0,k) &:= k \\ F(n+1,k) &:= F(n, \lambda_{n,n'}(k(n', n+n'))) \end{aligned}$$

is applied to n, the left projection $\lambda_{n_0,n_1} n_0$ and $0, 1$.

5.2.3. Proof transformations. In the next two examples we allow the decoration algorithm to substitute an auxiliary lemma used in the proof by a lemma that we specify explicitly. The algorithm will verify if the lemma passed to it as an argument is fitting and if this is the case, it will replace the lemma used in the original proof by the specified one. If not, the initial lemma is kept. This will allow for a certain control over the computational content, as shown by the following examples.

5.2.3.1. *Avoiding factorization.* Our first example is an elaboration of Constable's idea described in the introduction. Let Pn mean "n is prime". Consider

$$\begin{aligned} &\forall_n (Pn \vee^{\mathrm{r}} \exists^{\mathrm{d}}_{m,k>1}(n = mk)) && \text{factorization,} \\ &\forall_n (Pn \vee^{\mathrm{u}} \exists^{\mathrm{d}}_{m,k>1}(n = mk)) && \text{prime number test.} \end{aligned}$$

Euler's φ-function has the properties

$$\begin{cases} \varphi(n) = n - 1 & \text{if } Pn, \\ \varphi(n) < n - 1 & \text{if } n \text{ is composed.} \end{cases}$$

Suppose that somewhat foolishly we have used factorization and these properties to obtain a proof of

$$\forall_n (\varphi(n) = n - 1 \vee^{\mathrm{u}} \varphi(n) < n - 1).$$

Our goal is to get rid of the expensive factorization algorithm in the computational content, via decoration.

The decoration algorithm arrives at the factorization theorem

$$\forall_n (Pn \vee^{\mathrm{r}} \exists^{\mathrm{d}}_{m,k>1}(n = mk))$$

with the decorated formula

$$\forall_n (Pn \vee^{\mathrm{u}} \exists^{\mathrm{d}}_{m,k>1}(n = mk)).$$

Since the prime number test can be considered instead of the factorization lemma, we can specify that the decoration algorithm should try to replace the former by the latter. In case this is possible, a new proof is constructed, using the the prime number test lemma. Should this fail, the factorization lemma is kept. As it turns out in this case, the replacement is possible.

In the Minlog implementation the difference is clearly visible. `cFact` denotes the computational content of the factorization lemma `Fact` (i.e., the factorization algorithm), and `cPTest` the computational content of the lemma `PTest` expressing the prime number test. The extract from the original proof involves computing `(cFact n)`, i.e., factorizing the argument, whereas after decoration the prime number test `cPTest` suffices.

```
(define eterm (proof-to-extracted-term nproof))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
;; [n][if (cFact n) True ([algC]False)]

(define decnproof (fully-decorate nproof "Fact" "PTest"))
(pp (nt (proof-to-extracted-term decnproof)))
;; cPTest
```

5.2.3.2. *Maximal scoring segment.* The second example is due to Bates and Constable (1985), and deals with the "maximal scoring segment" (MSS) problem. Let X be a set with a linear ordering $\leq$, and consider an infinite sequence $f\colon \mathbf{N} \to X$ of elements of X. Assume further that we have a function $M\colon (\mathbf{N} \to X) \to \mathbf{N} \to \mathbf{N} \to X$ such that $M(f, i, k)$ "measures" the segment $f(i), \dots, f(k)$. The task is to find a segment determined by $i \leq k \leq n$ such that its measure is maximal. To simplify the formalization let us consider M and f fixed and define $\mathrm{seg}(i, k) := M(f, i, k)$.

Such a problem appears e.g. in computational biology, when one wants to compute regions with high G, C content in DNA. Let

$$\begin{aligned}
&X := \{G, C, A, T\},\\
&g\colon \mathbf{N} \to X \quad \text{(gene)},\\
&f\colon \mathbf{N} \to \mathbf{Z}, \quad f(i) := \begin{cases} 1 & \text{if } g(i) \in \{G, C\},\\ -1 & \text{if } g(i) \in \{A, T\},\end{cases}\\
&\mathrm{seg}(i, k) = f(i) + \dots + f(k).
\end{aligned}$$

Of course we can simply solve this problem by trying all possibilities; these are $O(n^2)$ many. The first proof to be given below corresponds to this general claim. Then we will show that for a more concrete problem with the sum $x_i + \dots + x_k$ as measure the proof can be simplified, using monotonicity of the sum at an appropriate place. From this simplified proof one can extract a better algorithm, which is linear rather than quadratic. Our goal is to achieve this effect by decoration.

Let us be more concrete. The original specification is to find a maximal segment $x_i, \dots, x_k$, i.e.,

$$\forall_n \exists^{\mathrm{d}}_{i \le k \le n} \forall_{i' \le k' \le n} (\mathrm{seg}(i', k') \le \mathrm{seg}(i, k)).$$

A special case is to find the maximal *end* segment

$$\forall_n \exists^{\mathrm{l}}_{j \le n} \forall_{j' \le n} (\mathrm{seg}(j', n) \le \mathrm{seg}(j, n)).$$

We provide two lemmata proving the existence of a maximal end segment for $n+1$. The first one is

$$\mathtt{L}\colon \forall_n \exists^{\mathrm{l}}_{j \le n+1} \forall_{j' \le n+1} (\mathrm{seg}(j', n+1) \le \mathrm{seg}(j, n+1)).$$

Its proof introduces an auxiliary variable m and proceeds by induction on m, with n a parameter:

$$\forall^{\mathrm{nc}}_n \forall_{m \le n+1} \exists^{\mathrm{l}}_{j \le n+1} \forall_{j' \le m} (\mathrm{seg}(j', n+1) \le \mathrm{seg}(j, n+1)).$$

The second one is

$$\mathtt{LMon}\colon \forall^{\mathrm{nc}}_n (\mathtt{ES}_n \to \mathtt{Mon} \to \exists^{\mathrm{l}}_{j \le n+1} \forall_{j' \le n+1} (\mathrm{seg}(j', n+1) \le \mathrm{seg}(j, n+1))).$$

It has as additional assumptions the existence $\mathtt{ES}_n$ of a maximal end segment for n

$$\mathtt{ES}_n\colon \exists^{\mathrm{l}}_{j \le n} \forall_{j' \le n} (\mathrm{seg}(j', n) \le \mathrm{seg}(j, n))$$

and the assumption Mon of monotonicity of seg

$$\mathtt{Mon}\colon \mathrm{seg}(i, k) \le \mathrm{seg}(j, k) \to \mathrm{seg}(i, k+1) \le \mathrm{seg}(j, k+1).$$

The proof proceeds by cases on $\mathrm{seg}(j, n+1) \le \mathrm{seg}(n+1, n+1)$. If $\le$ holds, take $n+1$, else the previous j.

We now prove the existence of a maximal segment by induction on n, simultaneously with the existence of a maximal end segment.

$$\mathtt{MaxSegMon}\colon \forall_n (\exists^{\mathrm{d}}_{i \le k \le n} \forall_{i' \le k' \le n} (\mathrm{seg}(i', k') \le \mathrm{seg}(i, k)) \wedge^{\mathrm{d}} \\ \exists^{\mathrm{l}}_{j \le n} \forall_{j' \le n} (\mathrm{seg}(j', n) \le \mathrm{seg}(j, n)))$$

In the step, we compare the maximal segment i, k for n with the maximal end segment $j, n+1$ provided separately. If $\le$ holds, take the new i, k to be $j, n+1$. Else take the old i, k.

Depending on how the existence of a maximal end segment was proved, we obtain a quadratic or a linear algorithm. If we consider the first proof involving induction on the auxiliary variable m, we obtain a quadratic algorithm as follows.

```
(define eterm (proof-to-extracted-term
               (theorem-name-to-proof "MaxSegMon")))
(add-var-name "ijk" (py "nat@@nat@@nat"))
(define neterm (rename-variables (nt eterm)))
(pp neterm)
```

The result is

```
[le,seg,n](Rec nat=>nat@@nat@@nat)n(0@0@0)
 ([n0,ijk]
   [if (le(seg left ijk right right ijk)
          (seg((cL alpha)le seg n0(Succ n0))(Succ n0)))
     ((cL alpha)le seg n0(Succ n0))
     (left ijk)]@
   (cL alpha)le seg n0(Succ n0)@
   [if (le(seg left ijk right right ijk)
          (seg((cL alpha)le seg n0(Succ n0))(Succ n0)))
     (Succ n0)
     (right right ijk)])
```

The computational content of L involves as additional recursion, since L was proved by induction on m.

```
(pp (rename-variables (nt
 (proof-to-extracted-term (theorem-name-to-proof "L")))))

[le,seg,n,n0](Rec nat=>nat)n0 0
 ([n1,n2][if (le(seg n2(Succ n))(seg(Succ n1)(Succ n)))
             (Succ n1)
             n2])
```

The two nested recursions give a quadratic algorithm.

Now how could the better proof be found by decoration? We have

$$\texttt{L}\colon \forall_n \exists^{\mathrm{l}}_{j\le n+1} \forall_{j'\le n+1}(\mathrm{seg}(j', n+1) \le \mathrm{seg}(j, n+1)),$$

$$\texttt{LMon}\colon \forall^{\mathrm{nc}}_n(\texttt{ES}_n \to \texttt{Mon} \to \exists^{\mathrm{l}}_{j\le n+1} \forall_{j'\le n+1}(\mathrm{seg}(j', n+1) \le \mathrm{seg}(j, n+1))).$$

The decoration algorithm arrives at L with

$$\exists^{\mathrm{l}}_{j\le n+1} \forall_{j'\le n+1}(\mathrm{seg}(j', n+1) \le \mathrm{seg}(j, n+1)).$$

LMon fits as well, its assumptions $\texttt{ES}_n$ and Mon are in the context, and it has the less extended $\forall^{\mathrm{nc}}_n$ rather than $\forall_n$, hence is preferred.

In Minlog we can do the decoration by executing

```
(define decproof
 (decorate (theorem-name-to-proof "MaxSegMon") "L" "LMon"))
(define eterm (proof-to-extracted-term decproof))
(pp (rename-variables (nt eterm)))

[le,seg,n](Rec nat=>nat@@nat@@nat)n(0@0@0)
 ([n0,ijk]
   [if (le(seg left ijk right right ijk)
           (seg((cLMon alpha)le seg n0 left right ijk)
              (Succ n0)))
     ((cLMon alpha)le seg n0 left right ijk)
     (left ijk)]@
   (cLMon alpha)le seg n0 left right ijk@
   [if (le(seg left ijk right right ijk)
          (seg((cLMon alpha)le seg n0 left right ijk)
              (Succ n0)))
     (Succ n0)
     (right right ijk)])
```

The computational content `cLMon` of `LMon` is

```
(pp (rename-variables
     (nt (proof-to-extracted-term
          (theorem-name-to-proof "LMon")))))

[le,seg,n,n0][if (le(seg n0(Succ n))(seg(Succ n)(Succ n)))
                 (Succ n)
                 n0]
```

which does not involve recursion any more. Hence after decoration we have a linear algorithm.

APPENDIX A

Logic in Minlog: a tutorial

The purpose of this appendix is to give a rather basic introduction to the Minlog system, by means of some simple examples. Minlog is implemented in Scheme. Minlog's favorite dialect is Petite Chez Scheme from Cadence Research Systems, which is freely distributed at the Internet address `www.scheme.com`. The Minlog system can be downloaded from the Internet address `http://www.minlog-system.de`

For a thorough presentation of Minlog and the motivation behind it the reader should consult the reference manual. For a more in–depth presentation of the theory underlying Minlog, the reader might find it useful also to consult the book Schwichtenberg and Wainer (2012). The papers listed in the Minlog web page also provide a more detailed and advanced description of specific features of the system. In addition, the Minlog distribution comes equipped with a directory of examples, to which the user is referred.

In the following we shall assume tacitly that you are using an UNIX-like operating system and that Minlog is installed in ~/minlog, where ~ denotes as usual your home directory. Also, C-$\langle chr \rangle$ means: hold the CONTROL key down while typing the character $\langle chr \rangle$, while M-$\langle chr \rangle$ means: hold the META (or EDIT or ESC or ALT) key down while typing $\langle chr \rangle$.

In order to use Minlog, one simply needs a shell in which to run Minlog and also an editor in which to edit and record the commands for later sessions. In this chapter we shall refer to GNU Emacs[1]. While working with Emacs, the ideal would be to split the window in two parts: one containing the file in which to store the commands, and the other with the Minlog interactive session taking place. To this aim, it is recommended to use the startup script *~/minlog/minlog* which takes scheme files as (optional) arguments. For example

```
~/minlog/minlog file.scm
```

opens a new Emacs-window which is split into two parts. The upper part contains the file *(Buffer file.scm)* whereas the lower part shows the Minlog response *(Buffer *minlog*)*.

[1]See also the Appendix C.1 for some useful keyboard commands to start working with Emacs.

Alternatively, one can open emacs and invoke Minlog by loading the file *minlog.el*:

```
M-x load-file <enter>
~/minlog/minlog.el
```

In both cases the file *init.scm* is loaded. In fact, one could also simply evaluate `(load "~/minlog/init.scm")` to start Minlog.

To execute a command of our file, we simply place the cursor at the end of it (after the closed parenthesis), and type `C-x C-e`. In general, `C-x C-e` will enable us to process any command we type in `file.scm`, one at the time. To process a whole series of commands, one can highlight the region of interest and type `C-c C-r`. We should also mention at this point that to undo one step, it is enough to give the command `(undo)`, while `(undo n)` will undo the last n steps. Finally, we can type `(exit)` to end a Scheme session and `C-x C-c` to exit Emacs.

A.1. Propositional logic

A.1.1. A first example. We shall start from a simple example in propositional logic. Suppose we want to prove the tautology:

$$(A \to (B \to C)) \to ((A \to B) \to (A \to C)).$$

In the following we shall make use of the convention for which parenthesis are associated to the right (as this is also implemented in Minlog). Therefore the formula above becomes:

$$(A \to B \to C) \to (A \to B) \to A \to C.$$

It is very important, especially at the beginning, to pay attention to the use of parenthesis to prevent mistakes; it might be a good strategy to rather exceed in their use in the first examples. Minlog will automatically delete the parenthesis which are not needed, therefore facilitating the reading.

A.1.1.1. *Making a sketch of the proof.* The first task will be to make an informal sketch of the proof. While making the plan we should consider the following fact: Minlog (mainly) implements "Goal Driven Reasoning", also called "Backward Chaining". That means that we start by writing the conclusion we aim at as our goal and then, step by step, refine this goal by applying to it appropriate logical rules. A logical rule will have the effect of reducing the proof of a formula, the goal, to the proof of one or more other formulas, which will become the new goals. If our proof is correct, then the formula will be proved when we reach the point of having no more goals to solve. In other words, Minlog keeps a list of goals and updates it each time a logical rule is applied. The proof is completed when the list of goals is empty.

In this case the tautology we want to prove is made of a series of implications, hence we will have to make repeated use of basic rules for "deconstructing" implications. The first move will then be to assume that the antecedent of the outmost implication is true and try to derive the consequent from it. That is, we assume $A \to B \to C$ and want to derive:

$$(A \to B) \to A \to C;$$

hence we set the latter as our new goal. Then we observe that the formula $(A \to B) \to A \to C$ is an implication as well, and thus can be treated in the same way; so we now assume both $A \to B \to C$ and $A \to B$ and wish to derive $A \to C$. Clearly, we can make the same step once more and obtain $A \to B \to C$, $A \to B$ and A as our premises and try to derive C from them. Now we observe that in order to prove C from the assumption $A \to B \to C$, we need to prove both A and B. Obviously A is proved, as it is one of our assumptions, and B immediately follows from $A \to B$ and A by modus ponens.

A.1.1.2. *Implementing the proof.* Once we have a plan for the proof, we can start implementing it in Minlog. The initial step would then be to write the formula in Minlog. For this purpose, we declare three predicate variables A, B and C by writing[2]:

```
(add-pvar-name "A" "B" "C" (make-arity))
```

The expression `(make-arity)` produces the empty arity for A, B and C (see the reference manual for a description of `make-arity`). Minlog will then write:

```
ok, predicate variable A: (arity) added
ok, predicate variable B: (arity) added
ok, predicate variable C: (arity) added
```

We now want to prove the above formula with Minlog; we thus need to set it as our goal.

A.1.1.3. *Setting the goal.* To set a goal, we use the command `set-goal` followed by the formula. In the present case:

```
(set-goal "(A -> B -> C) -> (A -> B) -> A -> C")
```

Alternatively, we can first give a name to the formula we wish to prove and then use this name to set the goal. In the present case, let's call `distr` (for distributivity of implication) the formula to be proved:

```
(define distr (pf "(A -> B -> C) -> (A -> B) -> A -> C"))
```

The `define` command has the effect of defining a new variable, in this case `distr`, and attaching to it the Scheme term which is produced by

[2]We could as well have introduced predicate constants instead of predicate variables. In this case we would have used the command `add-predconst-name`, with the same syntax.

the function `pf` applied to the formula we enter. In fact, the function `pf`, short for "parse formula", takes a string as argument and returns a Scheme term. This Scheme term is the *internal form* in Minlog of our formula, and `distr` is a name referring to it. By typing `distr`, one can see the value of this variable. The strategy of naming a formula might turn out to be particularly useful in case of very long goals.

To set `distr` as our goal we type:

```
(set-goal distr)
```

Typically, Minlog will number the goals occurring in the proof, and will display the top goal as number 1, preceded by a question mark. Minlog will print:

```
?_1: (A -> B -> C) -> (A -> B) -> A -> C
```

A.1.1.4. *The proof.* According to our sketch, the first step in proving the tautology was to assume the antecedent of the implication and turn the consequent into our new goal. This is simply done by writing:

```
(assume 1)
```

Here the number `1` is introduced by us to identify and name the hypothesis. Minlog will thus denote this hypothesis by `1`:

```
ok, we now have the new goal
?_2: (A -> B) -> A -> C from
  1:A -> B -> C
```

We repeat the assume command to decompose the implication in the second goal and obtain a new goal:

```
(assume 2)
```

```
ok, we now have the new goal
?_3: A -> C from
  1:A -> B -> C
  2:A -> B
```

And we decompose the new goal once more:

```
(assume 3)
```

```
ok, we now have the new goal
?_4: C from
  1:A -> B -> C
  2:A -> B
  3:A
```

We now need to start using our assumptions. As already mentioned, in order to prove C it is enough to prove both A and B, by assumption `1`. Therefore we write: `(use 1)`. This has the effect of splitting the goal in two distinct subgoals (note how the subgoals are numbered):

```
(use 1)
ok, ?_4 can be obtained from
?_6: B from
  1:A -> B -> C
  2:A -> B
  3:A
?_5: A from
  1:A -> B -> C
  2:A -> B
  3:A
```

Then we write:

```
 (use 3)
ok, ?_5 is proved.  The active goal now is
?_6: B from
  1:A -> B -> C
  2:A -> B
  3:A
```

And conclude the proof by:

```
(use 2)
ok, ?_6 can be obtained from
?_7: A from
  1:A -> B -> C
  2:A -> B
  3:A
>
(use 3)
ok, ?_7 is proved.  Proof finished.
```

To see a record of the complete proof, simply type `(display-proof)`. Other useful commands are `(proof-to-expr)` and the particularly useful `(proof-to-expr-with-formulas)`. See the manual for a description of the various display commands available in Minlog.

We observe that the first three `assume` commands could be replaced by a single one, i.e., `(assume 1 2 3)`. Also, in alternative to the last two `use` commands, we could have given only one command: `(use-with 2 3)`, which amounts to applying a cut to the premises 2 and 3. A final remark: in case of rather complex proofs, it may be more convenient to use names to denote specific hypothesis, instead of making use of bare numbers. To do so, one can simply use the `assume` command, followed by the name of the assumption in double quotes.

Before starting to read the next section it is advisable to consult the reference manual for a compendium of the commands utilized in this example. It is worth noticing that in general these commands have a wider applicability than their usage as here presented.

A.1.2. A second example: conjunction. The next example is a simple tautology made of conjunctions and an implication. We want to prove[3]:

$$A \wedge B \to B \wedge A.$$

In this case, we shall simply record the code of our Minlog proof, asking the reader to check Minlog's reply at each step. We start as usual with declaring the variables A and B and setting the goal. Note that if one uses the same file for a number of examples, there is no need to re-declare the same predicate variables each time. Hence here and in the example file available with the distribution repeated declarations are commented by prefixing a ";; ".

```
;; (add-pvar-name "A" "B" (make-arity))
(set-goal "A & B -> B & A")
```

We than notice that the main connective is an implication, and thus call the command `assume`:

```
(assume 1)
```

Next we need a command which operates on a conjunction and splits it into its two components. We can simply write:

```
(split)
```

Finally, we impart the command `use` which is utilized to obtain the left (respectively the right) conjunct from an assumption which is a conjunction and "`use`" it to derive the goal.

```
(use 1)
(use 1)
```

This completes the proof.

The reader is encouraged to try and prove other tautologies. For example the following:

(1) $A \to B \to A$
(2) $(A \to B \to C) \to B \to A \to C$
(3) $(A \to B) \to (B \to C) \to A \to C$
(4) $(A \to B \to C) \to A \wedge B \to C$

[3]Recall that $\wedge$ binds stronger than $\to$.

A.1.3. Classical logic. To conclude this section on propositional logic, we give a short example of a tautology which uses classical logic.

Minlog implements minimal logic. However, it is possible to use Minlog to prove a proposition which holds in an extension of minimal logic, like intuitionistic or classical logic. This is achieved by adding specific principles which are characteristics of each kind of logic, like the "ex falso quodlibet" or a version of the "tertium non datur". We recall that intuitionistic logic is obtained from minimal logic by adding a principle which enables us to derive any formula from falsity ("ex falso quodlibet" is Latin for "anything follows from falsity"). This principle is usually stated as: $\bot \to A$, for arbitrary A, where $\bot$ denotes falsity. Classical logic may be obtained by adding to intuitionistic logic the law of "tertium non datur" (Latin for "there is no third option"). This is usually stated as $A \vee \neg A$, for any A. However, classical logic may also be obtained by adding to minimal logic a consequence of "tertium non datur", known as "stability", asserting that $\neg\neg A \to A$, for any A. Negation is represented in Minlog as follows: $\neg A$ is $A \to \bot$; consequently stability is written as:

$$((A \to \bot) \to \bot) \to A,$$

for each A.

We can add "external" principles to Minlog by introducing so–called *"global assumptions"*. Roughly speaking, a global assumption is a proposition whose proof does not concern us at the moment; hence it can also be an assumption with no proof. Some global assumptions are already set by default, like `EfqLog` (which is Minlog's name for "ex falso quodlibet") and `StabLog` (which is Minlog's name for the law of stability). In order to check which global assumptions we have at our disposal we type: `(display-global-assumptions)`. To check a particular global assumption (or theorem) whose name we already know, we write `pp` (for pretty-print) followed by the name of the assumption (or theorem) we want to check, e.g.: `(pp "StabLog")`. Of course we can also introduce our own global assumptions and remove them at any time (see the reference manual for the relevant commands).

In the following we wish to prove the tautology:

$$((A \to B) \to A) \to A,$$

which is known as *Peirce formula.* Also for this example, we will assume that the reader has prepared a sketch of the proof, and we will only give an intuitive idea of the proof, preferring to rather concentrate on the Minlog interaction, which will be given in its complete form.

As in the previous examples, we observe first of all that the goal is an implication, hence we will assume its antecedent, $(A \to B) \to A$, and try to prove its consequent, A. Now classical logic comes into play: in order to prove A we will assume that its negation holds and try to obtain a contradiction from it. This will be achieved by use of Stability. We further note that in order to make the argument work, we will need at some stage to resort also to "ex falso quodlibet".

We start by setting the goal and assuming the antecedent of the implication:

```
;; (add-pvar-name "A" "B" (make-arity))
(set-goal "((A -> B) -> A) -> A")
(assume 1)
```

We obtain:

```
ok, we now have the new goal
?_2: A from
  1:(A -> B) -> A
```

We now apply Stability, `StabLog`, so that the goal A will be replaced by its double negation: $(A \to \bot) \to \bot$. Note that $\bot$ is called `bot` in Minlog.

```
(use "StabLog")
ok, ?_2 can be obtained from
?_3: (A -> bot) -> bot from
  1:(A -> B) -> A
```

Since this is an implication, we let:

```
(assume 2)
ok, we now have the new goal
?_4: bot from
  1:(A -> B) -> A
  2:A -> bot
```

We then use hypothesis 2 to replace the goal $\bot$ by A.

```
(use 2)
ok, ?_4 can be obtained from
?_5: A from
  1:(A -> B) -> A
  2:A -> bot
```

Also A can be replaced by $A \to B$ by use of hypothesis 1. Subsequently, we can assume the antecedent of the new goal, A, and call it hypothesis 3:

```
(use 1)
ok, ?_5 can be obtained from
?_6: A -> B from
  1:(A -> B) -> A
  2:A -> bot
```

```
>
(assume 3)
ok, we now have the new goal
?_7: B from
  1:(A -> B) -> A
  2:A -> bot
  3:A
```

Now we can make use of the principle of "ex falso quodlibet": if we want to prove B, we can instead prove falsum, since from falsum anything follows, in particular B. Our goal can be updated to $\bot$ by the following instance of `use`:

```
(use "EfqLog")
ok, ?_7 can be obtained from:
?_8: bot from
  1:(A -> B) -> A
  2:A -> bot
  3:A
```

The next two steps are obvious.

```
(use 2)
ok, ?_8 can be obtained from
?_9: A from
  1:(A -> B) -> A
  2:A -> bot
  3:A
>
(use 3)
ok, ?_9 is proved.  Proof finished.
```

To familiarize yourself with negation, prove the following propositions.

(1) $(A \to B) \to \neg B \to \neg A$,
(2) $\neg(A \to B) \to \neg B$,
(3) $\neg\neg(A \to B) \to \neg\neg A \to \neg\neg B$.

A.2. Predicate logic

A.2.1. A first example with quantifiers. We now exemplify how to prove a statement in predicate logic. We want to prove:

$$\forall_n(Pn \to Qn) \to \forall_n Pn \to \forall_n Qn.$$

Here we assume that the predicates P and Q take natural numbers as arguments. Therefore, we first of all load a file, already available within the distribution, which introduces the algebra of natural numbers, including some operations on them, like for example addition. The reader is advised to have a look at this file by typing

```
(libload "nat.scm")
```

Note that this command will produce the display of the whole file nat.scm, evaluated. If we wish to load the file "silently", then we can precede the libload command by the following line:

```
(set! COMMENT-FLAG #f)
```

This will have the effect of hiding the output. To revert to full display (which is needed to proceed with the proof) one types:

```
(set! COMMENT-FLAG #t)
```

As we load `nat.scm`, we can make use of all the conventions which are there stipulated; in particular, we can take n, m, k to be variables for natural numbers (i.e. of type `nat`)[4]. The next task is to introduce two new predicate variables P and Q which take natural numbers as arguments:

```
(add-pvar-name "P" "Q" (make-arity (py "nat")))
> ok, predicate variable P: (arity nat) added
ok, predicate variable Q: (arity nat) added
```

We then set the goal:

```
(set-goal "all n(P n -> Q n) -> all n P n -> all n Q n")
> ?_1: all n(P n -> Q n) -> all n P n -> all n Q n
```

As usual, we first have to "deconstruct" the implications:

```
(assume 1 2)
> ok, we now have the new goal
?_2: all n Q n from
  1:all n(P n -> Q n)
  2:all n P n
```

Then we need to take care of the universal quantifier in the goal:

```
(assume "n")
> ok, we now have the new goal
?_3: Q n from
  1:all n(P n -> Q n)
  2:all n P n
  n
```

Note that we could have also used only one command to perform all these actions:

```
(assume 1 2 "n")
```

[4]Note that Minlog automatically infers the type of the variables $x_0, x_1, \ldots$ once the variable x has been declared. In this case, it infers for example that $n_0, n_1, \ldots$ are also natural numbers.

We finally have to "use" our hypothesis to conclude the proof:

```
(use 1)
(use 2)
> ok, ?_3 can be obtained from
?_4: P n from
  1:all n(P n -> Q n)
  2:all n P n
  n
> ok, ?_4 is proved.  Proof finished.
```

A.2.2. Another example. We now wish to prove the following:

$$\forall_n(Pn \to Qn) \to \exists_n Pn \to \exists_n Qn.$$

We start by setting the goal and eliminating the two implications:

```
(set-goal "all n(P n -> Q n) -> ex n P n -> ex n Q n")
(assume 1 2)
?_1: all n(P n -> Q n) -> ex n P n -> ex n Q n
> ok, we now have the new goal
?_2: ex n Q n from
  1:all n(P n -> Q n)
  2:ex n P n
```

Next we want to use the second assumption, now by applying "forward reasoning". Thus we assume there is a witness, say n_0, for this existential formula and call the resulting hypothesis P_{n_0}. To this aim we make use of a command called `by-assume`:

```
(by-assume 2 "n0" "P_n0")
> ok, we now have the new goal
?_5: ex n Q n from
  1:all n(P n -> Q n)
  n0  P_n0:P n0
```

Alternatively, from the existential formula in hypothesis 2, we can extract a witness, say n_0, by applying an existential elimination.

```
(ex-elim 2)
(assume "n0" "P_n0")
> ok, ?_2 can be obtained from
?_3: all n(P n -> ex n0 Q n0) from
  1:all n(P n -> Q n)
  2:ex n P n
> ok, we now have the new goal
?_4: ex n Q n from
  1:all n(P n -> Q n)
  2:ex n P n
  n0  P_n0:P n0
```

As we have already used assumption 2, we may wish to drop it by writing:

```
(drop 2)
```

We conclude the proof by providing a witness, the term n_0, for our existential goal formula. This will be done by calling the command `ex-intro` with argument `(pt "n0")`, where `pt` stands for "parse term"[5].

```
(ex-intro (pt "n0"))
> ok, ?_5 can be obtained from
?_6: Q n0 from
  1:all n(P n -> Q n)
  n0  P_n0:P n0
```

Finally, we first instantiate the universal quantifier in assumption 1 to n_0 and then perform a cut with the third assumption.

```
(use-with 1 (pt "n0") "P_n0")
>  ok, ?_5 is proved.  Proof finished.
```

A.2.3. An example with relations. In the next example we wish to prove that every total relation which is symmetric and transitive is reflexive. For simplicity we shall work also in this case with the algebra of the natural numbers. Our aim is to prove the following statement:

$$\begin{aligned}&\forall_{n,m}\,(Rnm \to Rmn) \,\wedge\, \forall_{n,m,k}\,(Rnm \,\wedge\, Rmk \to Rnk)\\ &\to \forall_n\,(\exists_m\, Rnm \to Rnn),\end{aligned}$$

where n, m, k vary on natural numbers, while R is a binary predicate on natural numbers.

Before attacking our formula, we observe that in general conjunctions are quite complex to deal with, as they normally imply the branching of a proof in two subproofs. Thus we might wish to first find a formula which is equivalent to the one above and "simpler" to prove. We note that we can equivalently express our goal by a formula in which the conjunctions have been replaced by implications. Also, we can express the conclusion with a prenex universal quantifier instead of an existential one. That is, we can instead prove the following equivalent formula:

$$\begin{aligned}&\forall_{n,m}\,(Rnm \to Rmn) \,\wedge\, \forall_{n,m,k}\,(Rnm \,\wedge\, Rmk \to Rnk)\\ &\to \forall_{n,m}\,(Rnm \to Rnn).\end{aligned}$$

[5]Since `ex-intro` can only take a term as an argment, one can also shortly write `(ex-intro "n0")` instead.

We observe that the strategy of first simplifying the goal may in some cases allow one to considerably reduce the amount of time needed to prove a statement. For completeness and for a comparison, we shall also record a proof of the original goal at the end of this section.

We now start by introducing the constant R. We also want to facilitate our work a bit further and introduce names for our two assumptions. In the following we shall use the function `py` (for "parse type"), which is the analogous for types of the function parse formula that we encountered in the first example.

```
(add-pvar-name "R" (make-arity (py "nat") (py "nat")))
(define Sym (pf "all n,m(R n m  -> R m n)"))
(define Trans (pf "all n,m,k(R n m -> R m k -> R n k)"))
```

We now state the goal:

```
(set-goal (mk-imp Sym Trans (pf "all n,m(R n m -> R n n)")))
?_1: all n,m(R n m -> R m n)
       -> all n,m,k(R n m -> R m k -> R n k)
       -> all n,m(R n m -> R n n)
```

Note that also in this case, we could have directly written the two formulas as antecedents of the implication, avoiding the detour through a `define` command. In case of more complex formulas, however, or when we need to use the same formulas for various proofs through one session, the strategy of introducing names for assumptions can be quite useful.

We now observe that the goal is an implication, so that the first step is to write `(assume "Sym" "Trans")`. We now obtain a universally quantified formula and hence need to proceed to eliminate the quantifiers. This can be accomplished by another `assume` command in which we specify two natural numbers, say n and m. So we write `(assume "n" "m")`. This produces an implication which again needs to be eliminated by another `assume` command, say `(assume 3)`. Quite conveniently we can put all these commands together by simply writing:

```
(assume "Sym" "Trans" "n" "m" 3)
ok, we now have the new goal
?_2: R n n from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
```

The next move is to make use of our assumptions. It is clear that if we take k to be n in `Trans`, then the goal can be obtained by an instance of `Sym`, and the proof is easily completed. We here utilize `use` by additionally providing a term, "m", which instantiates the only variable which can not be automatically inferred by unification.

```
(use "Trans" (pt "m"))
?_4: R m n from
  Sym:all n,m (R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
?_3: R n m from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
```

The use command has the effect of replacing the current goal with two new goals. These are obtained from Trans by instantiating the quantifiers with n, m and n (the two n being inferred by unification) and then by replacing the goal with the antecedents of the resulting instance of Trans. We can now write:

```
(use 3)
> ok, ?_3 is proved.  The active goal now is
?_4: R m n from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
```

We finally employ Sym and another use:

```
(use "Sym")
ok, ?_4 can be obtained from
?_5: R n m from
  Sym:all n,m(R n m -> R m n)
  Trans:all n,m,k(R n m -> R m k -> R n k)
  n  m  3:R n m
>
(use 3)
ok, ?_5 is proved.  Proof finished.
```

A.2.4. The same example again. We here present a Minlog proof of the original goal in the previous example, as it allows us to exemplify the use of some new commands. We shall leave the proof uncommented and make a few remarks at the end. The reader will have to examine the proof and check Minlog's interaction.

```
;; (libload "nat.scm")
;; (add-pvar-name "R" (make-arity (py "nat") (py "nat")))
(set-goal "all n,m(R n m  -> R m n)
                 & all n,m,k(R n m & R m k -> R n k)
                 -> all n(ex m R n m -> R n n)")
(assume 1)
(inst-with 1 'left)
(inst-with 1 'right)
(drop 1)
(name-hyp 2 "Sym")
(name-hyp 3 "Trans")
(assume "n" 4)
(ex-elim 4)
(assume "m" 5)
(cut "R m n")
(assume 6)
(use-with "Trans" (pt "n") (pt "m") (pt "n") "?")
(drop "Sym" "Trans" 4)
(split)
(use 5)
(use 6)
(use-with "Sym" (pt "n") (pt "m") 5)
```

The `use-with` command is similar to the `use` command, but when applied to a universal quantifier it requires to explicitly specify the terms one wants to instantiate. In the second occurrence of `use-with`, Minlog will instantiate as specified the universal quantifiers in the second premise and then use hypothesis 5 to prove the goal.

The command `inst-with` is analogous to `use-with`, but operates in forward reasoning; hence it allows one to simplify the hypothesis, instead of the conclusion. In this case, `(inst-with 1 'left)` has the effect of producing the left component of the conjunction which constitutes the first hypothesis. Similarly for the right component.

As to `cut`, this command enables one to introduce new goals: `(cut A)` has the effect of replacing goal B by two new goals, $A \to B$ and A.

In the proof above we have also made use of the commands `drop` and `name-hyp`. We have already seen the first command, which allows one to remove one or more hypothesis from the present context, to make the proof more readable. In fact, it simply replaces the current goal with another goal in which the hypothesis "dropped" are not displayed anymore (but they are not removed in general, as should be clear from the example above). The second command has similar "cosmetic" purposes, and allows one to rename a specific hypothesis and hence to work with names given by the user instead of numbers produced by default. Both these commands result especially useful in the case of long and intricate proofs.

A.2.5. Exercises. Prove the following goals:

(1) $\forall_{m,n} Rmn \to \forall_{n,m} Rmn$
(2) $\forall_{m,n} Rmn \to \forall_n Rnn$
(3) $\exists_m \forall_n Rmn \to \forall_n \exists_m Rmn$

A.2.6. Advanced exercises. Now two examples which involve a function. First declare a new function variable:

```
(add-var-name "f" (py "nat=>nat"))
(set-goal "all f(all n(P(f n) -> Q n) ->
                all n P n -> all n Q n)")
(set-goal "all f(all n(P n -> Q (f n)) ->
                ex n P n -> ex n Q n)")
```

And finally:

```
;; (add-pvar-name "Q" (make-arity (py "nat")))
;; (add-pvar-name "A" (make-arity))
(set-goal "all n(Q n -> A) -> (ex n Q n -> A)")
(set-goal "ex n(Q n -> A) -> all n Q n -> A")
```

A.2.7. Another example with classical logic. We conclude this section on predicate logic with the proofs of two formulas which hold in classical logic. First of all, we prove the inverse of the formula in the last exercise, now generalised to an arbitrary type. Then we use this formula (conveniently stored as a Lemma) to prove another formula which is usually known as the "Drinker" formula. So we start by proving:

$$(\forall_x Qx \to A) \to \tilde{\exists}_x (Qx \to A).$$

Here Q is a unary predicate which ranges on an arbitrary type, say α. In addition, the existential quantifier, $\tilde{\exists}$, is here a **classical** existential quantifier, to be distinguished from the existential quantifier we encountered in the previous example. A classical quantifier $\tilde{\exists}_x$ is nothing more than an abbreviation for $\neg \forall_x \neg$. Note that Minlog implements both quantifiers, with the appropriate corresponding rules.

We start by "removing" the predicate variable Q introduced before, so that we can re-introduce it as a fresh predicate variable which ranges on α rather than on the natural numbers. We also introduce two new variables, x and y, of type α. Finally, we set the goal.

```
(remove-pvar-name "Q")
(add-pvar-name "Q" (make-arity (py "alpha")))
(add-var-name "x" (py "alpha"))
(set-goal "(all x Q x -> A) -> excl x(Q x -> A)")
```

We start by "deconstructing" the two implications. Then we can instantiate the universal quantifier in assumption 2 to a canonical inhabitant of the type α.

```
(assume 1 2)
(use 2 (pt "(Inhab alpha)"))
```

Subsequently, we proceed by eliminating the implication in the goal, using assumption 1 and instantiating the resulting universal quantifier by x.

```
(assume 3)
(use 1)
(assume "x")
```

Now it's time to call in classical logic. The remaining steps should be self–explanatory. Note in particular the use of `EfqLog` and the command `save` at the end of the proof which enables us to save the proof and call it "Lemma".

```
(use "StabLog")
(assume 4)
(use 2 (pt "x"))
(assume 5)
(use 1)
(assume "x1")
(use "EfqLog")
(use-with 4 5)
(save "Lemma")
```

We now wish to prove the following:

$$\tilde{\exists}_x\,(Qx \to \forall_y\,Qy),$$

again with Q a unary predicate ranging on α.

The above formula is known as the "drinker" formula, as it says something like: "in a bar, there is a person such that if that person drinks then everybody drinks". To prove the "drinker", we observe that if we substitute the predicate variable A by $\forall_x Qx$ in the formula just proved, then we obtain the drinker formula. The substitution can be achieved by the following command.

```
(set-goal "excl x(Q x -> all x Q x)")
(use-with "Lemma"
          (make-cterm (pv "x") (pf "Q x"))
          (make-cterm (pf "all x Q x"))
          "?")
```

Here `pv` stands for "parse variable". In addition, `make-cterm` produces a *"comprehension term"* consisting of a list of variables and the formula we wish to substitute. For example, if we wish to replace a predicate variable P with arity $x_1, \ldots, x_n$ by a formula $F(y_1, \ldots, y_n)$ we need to give a comprehension term consisting of a list of variables $y_1, \ldots, y_n$ and the formula F (with free variables $y_1, \ldots, y_n$, plus possibly other variables, bound or free), i.e., write `(make-cterm (pv "y1")...(pv "yn") <formula F>)`. Note that the list of variables can also be empty, as in the second application of `make-cterm` above. We leave the rest of the proof as an exercise.

A.2.8. Equality reasoning. We now wish to prove that for any function f taking natural numbers to natural numbers, for any natural number n, the following holds:

$$fn = n \to f(fn) = n.$$

First of all we recall the nat library and introduce f. Then we set the goal and start by a familiar `assume` command.

```
;; (libload "nat.scm")
(add-var-name "f" (py "nat=>nat"))

(set-goal "all f,n(f n=n ->  f(f n)=n)")
(assume "f" "n" 1)
```

Next we can use the command `simp` which is an essential tool in Minlog's equality reasoning. This command has the effect of *simplifying* a proof which involves equal terms by performing an appropriate substitution in the goal. We conclude with a `use` command.

```
(simp 1)
(use 1)
```

Suppose now we wish to replace the right hand side by the left hand side in the equation above:

```
(set-goal "all f,n(n=f n -> n=f(f n))")
```

This can be proved by the following commands:

```
(assume "f" "n" 1)
(simp "<-" 1)
(use 1)
```

A.3. Automatic proof search

Minlog allows for automatic proof search. There are two distinct facilities for performing an automatic search in Minlog. The first is given by the command `(prop)` and exemplifies Hudelmaier-Dyckhoff's search for the case of minimal propositional logic (see e.g. Hudelmaier (1989), Dyckhoff (1992)). The second is given by the command `(search)` and allows to automatically find proofs also for some quantified formulas.

A.3.1. Search in propositional logic. When we give the command `prop`, Minlog will first look for a proof in propositional minimal logic. If it fails to find a proof for the given proposition, it will try with intuitionistic logic, by adding appropriate instances of "ex falso quodlibet". If this search also gives no positive answer, it will try to find a proof in classical logic, by adding appropriate instances of Stability.

To apply this search algorithm, one simply needs to type `(prop)`. One could do so after stating the goal or at any point in a proof from which one believes that (minimal) propositional logic should suffice. If Minlog finds a proof, one can then display it by means of any of the display commands available for proofs; for example by writing `dnp` (which is a shortcut for `display-normalized-proof`).

The reader is encouraged to try `prop` on the following tautologies:

(1) $(A \to B \to C) \to (A \to B) \to A \to C$
(2) $((A \to B) \to A) \to A$

A.3.2. Search in predicate logic. The command `search` embodies a search algorithm based on Miller (1991) and ideas of U. Berger (see the Minlog reference manual and Schwichtenberg (2004) for details on the algorithm and for some differences with Miller's original algorithm). The `search` command enables us to automatically find a proof for a wider class of formulas compared with `prop`, since it also works for some formulae with quantifiers (see the reference manual for a detailed description of the class of formulae dealt with by `search`). Note, however, that `search` only operates a search in *minimal logic.* If one wishes to apply this command to a classical formula like "Peirce's law", one could for example add the appropriate instances of "ex falso quodlibet" and of "Stability" as antecedents of the goal. In case of more complex proofs, in which one can not easily modify the actual goal, an alternative would be to avail oneself of a more complete use of the `search` command which allows us to specify some global assumptions, theorems or even hypotheses from the given context which one would like to use in the proof. Since the search space in the case of quantified formulas can become really vast, this possibility of declaring specific assumptions to be used in the proof can be very useful, especially if we also state the maximum number of multiplicities we allow for each assumption (i.e., the maximum number of times each assumption can be used in the proof). One can also use this same device to exclude the use of a specific assumption in the proof, simply by letting its multiplicity to be 0.

To use the plain version of `search`, one simply writes `(search)`. See the reference manual for the precise syntax of the command `search` when other assumptions are invoked with the respective multiplicities.

The reader is encouraged to use `search` to prove the following: $\forall_x(Px \to Qx) \to \exists_x Px \to \exists_x Qx$.

Here is a more complex example with `search`. We apply the algorithm to the following problem: if f is a continuous function then f composed with itself is also a continuous function. We suggest to solve the problem as follows.

```
;; (add-var-name "x" "y" (py "alpha"))
(add-tvar-name "beta")
(add-var-name "u" "v" "w" (py "beta"))
(add-infix-display-string "In" "elem" 'rel-op)
(add-var-name "f" (py "alpha=>alpha"))

(set-goal "all f(
 all x,v(f x elem v ->
  excl u(x elem u & all y(y elem u -> f y elem v))) ->
 all x,w(f(f x)elem w ->
  excl u(x elem u & all y(y elem u -> f(f y)elem w))))")
(search)
```

Note that one can switch on a verbose form of search by letting: `(set! VERBOSE-SEARCH #t)` before calling `search`. In this way one can see the single steps performed by the search algorithm and detect possible difficulties in finding a proof.

Also, `add-infix-display-string` allows us to define a token with infix notation for the program constant[6].

[6]Similarly there are commands for prefix and postfix use, for example: `add-prefix-display-string`.

APPENDIX B

Constructive existence proofs

B.1. Quotient and remainder

```
(load "~/git/minlog/init.scm")

(set! COMMENT-FLAG #f)
(libload "nat.scm")
(set! COMMENT-FLAG #t)

(add-var-name "q" "r" (py "nat"))

;; QR
(set-goal "all m,n ex q,r(n=(m+1)*q+r & r<m+1)")
(assume "m")
(ind)
```

We obtain two new goals, for the base and the step case

```
?_4:all n(ex q,r(n=(m+1)*q+r & r<m+1) ->
    ex q,r(Succ n=(m+1)*q+r & r<m+1))

?_3:ex q,r(0=(m+1)*q+r & r<m+1)
```

In the base case we can directly provide the terms

```
(ex-intro "0")
(ex-intro "0")
(split)
(use "Truth")
(use "Truth")
```

In the step case we instantiate the induction hypothesis by q and r.

```
(assume "n" "IH")
(by-assume "IH" "q" "qProp")
(by-assume "qProp" "r" "qrProp")
```

The result is

```
?_15: ex q,r(Succ n=(m+1)*q+r & r<m+1) from
  qrProp:n=(m+1)*q+r & r<m+1
```

At this point we distinguish cases on `r<m`, viewed as a boolean term.

```
(cases (pt "r<m"))
```

This produces two new goals

```
?_17:(r<m -> F) -> ex q,r0(Succ n=(m+1)*q+r0 & r0<m+1)

?_16:r<m -> ex q,r0(Succ n=(m+1)*q+r0 & r0<m+1)
```

both from the same context including `qrProp`. We first treat the case `r<m`. Then we leave `q` and increase `r` by one.

```
(assume "r<m")
(ex-intro "q")
(ex-intro "r+1")
(normalize-goal)
(split)
(use "qrProp")
(use "r<m")
```

Next we treat the case `r<m -> F`. Then we increase `q` by one and set `r` to 0.

```
(assume "r<m -> F")
(ex-intro "q+1")
(ex-intro "0")
```

Now the goal is

```
?_26: Succ n=(m+1)*(q+1)+0 & 0<m+1 from
  qrProp:n=(m+1)*q+r & r<m+1
```

Again we continue by normalizing

```
(normalize-goal)
```

The result is

```
?_27: n=m*q+q+m & T from
  qrProp:n=m*q+q+r & r<Succ m
```

Note that `normalize-goal` takes optional arguments. If there are none, the goal formula and all hypotheses are normalized. Otherwise exactly those among the hypotheses and the goal formula are normalized whose numbers (or names, or just `#t` for the goal formula) are listed as additional arguments. We split the conjunctive goal and observe that `r=m` follows from `n=m*q+q+r` and `r<m -> F` by some arithmetical lemmas (in `nat.scm`)

```
(split)
(assert "r=m")
 (use "NatLeAntiSym")
 (use "NatLtSuccToLe")
 (use "qrProp")
 (use "NatNotLtToLe")
 (use "r<m -> F")
(assume "r=m")
```

The result is

```
?_36: n=m*q+q+m from
  qrProp:n=m*q+q+r & r<Succ m
  r<m -> F:r<m -> F
  r=m:r=m
```

We continue by simplifying `qrProp` with `r=m`

```
(simphyp-with-to "qrProp" "r=m" "qrPropSimp")
```

The result is

```
?_38: n=m*q+q+m from
  qrPropSimp:n=m*q+q+m & m<Succ m
```

Now the goal is part of `qrPropSimp`, and we can finish the proof by

```
(use "qrPropSimp")
(use "Truth")
;; Proof finished.
(save "QR")
```

B.2. List reversal

```
(load "~/git/minlog/init.scm")

(set! COMMENT-FLAG #f)
(libload "nat.scm")
(libload "list.scm")
(set! COMMENT-FLAG #t)

(add-var-name "x" "y" "n" "a" "b" "c" "d" (py "nat"))
(add-var-name "v" "w" "u" (py "list nat"))

(add-ids
 (list (list "RevI"
             (make-arity (py "list nat") (py "list nat"))))
 '("RevI(Nil nat)(Nil nat)" "InitRevI")
 '("all x,v,w(RevI v w  -> RevI(v++x:)(x::w))" "GenRevI"))
```

We first prove that `RevI` satisfies `ListInitLastNat`.

```
;; ListInitLastNat
(set-goal "all u,y ex v,x (y::u)=v++x:")
(ind)
```

```
;; Base
(assume "y")
(ex-intro "(Nil nat)")
(ex-intro "y")
(use "Truth")

;; Step
(assume "x" "u" "IH" "y")
(inst-with-to "IH" (pt "x") "IHx")
(drop "IH")
(by-assume "IHx" "v" "vProp")
(by-assume "vProp" "y1" "vy1Prop")
(ex-intro "y::v")
(ex-intro "y1")
(simp "vy1Prop")
(use "Truth")
;; Proof finished.
(save "ListInitLastNat")
```

Using `ListInitLastNat` we prove our goal by induction on the length of the list, i.e., we prove

```
;; ExRevI
(set-goal "all n,v(n=Lh v -> ex w RevI v w)")
```

We first call `ind`. Subsequently, we tackle the base case by cases on `v`: if it is empty, by providing a witness, `Nil` of type `nat`, and then by using the first closure axiom for `RevI`; otherwise we use ex-falso.

```
(ind)
(cases)
(assume "Useless")
(ex-intro "(Nil nat)")
(use "InitRevI")
(assume "x" "u")
(ng)
(use "Efq")
```

In the step case we first assume the induction hypothesis and then again do cases on `v`. This time we can use ex-falso if `v` is empty. In case `v` is composed, we “cut in” an appropriate instance of `ListInitLastNat` (to produce a “let” in the extracted term; see below), then – after some standard commands – provide a witness, and finally use the second closure axiom for `RevI`.

```
(assume "n" "IHn")
(cases)
```

```
(ng)
(use "Efq")
(assume "x" "v" "LhHyp")
(cut "ex u,y (x::v)=u++y:")
(use "Id")
(assume "ExHyp")
(by-assume "ExHyp" "u" "uProp")
(by-assume "uProp" "y" "uyProp")
(inst-with-to "IHn" (pt "u") "IHnu")
(drop "IHn")
(simphyp-with-to "LhHyp" "uyProp" "n=Lhu")
(drop "LhHyp")
(inst-with-to "IHnu" "n=Lhu" "IHnuInst")
(drop "n=Lhu" "IHnu")
(by-assume "IHnuInst" "u1" "u1Prop")
(ex-intro "y::u1")
(simp "uyProp")
(use "GenRevI")
(use "u1Prop")
```

The formula cut in above is an instance of ListInitLastNat

```
(use "ListInitLastNat")
;; Proof finished.
(save "ExRevI")
```

This concludes the proof; we have saved it under the name `ExRevI`.

B.3. Linearity of the Brouwer-Kleene ordering

```
(load "~/git/minlog/init.scm")

(set! COMMENT-FLAG #f)
(libload "nat.scm")
(libload "list.scm")
(set! COMMENT-FLAG #t)

(add-var-name "p" "q" (py "boole"))
(add-var-name "a" "b" "c" (py "list boole"))

(add-program-constant
 "LtBK" (py "list boole=>list boole=>boole"))
(add-infix-display-string "LtBK" "<<" 'rel-op)

(add-computation-rules
 "(Nil boole)<<b" "False"
 "(p::a)<<(Nil boole)" "True"
 "(True::a)<<(True::b)" "a<<b"
 "(True::a)<<(False::b)" "True"
 "(False::a)<<(True::b)" "False"
 "(False::a)<<(False::b)" "a<<b")

;; LtBKTotal
(set-totality-goal "LtBK")
(assume "a^" "Ta")
(elim "Ta")
(ng #t)
(strip)
(use "TotalBooleFalse")
(assume "p^1" "Tp1" "a^1" "Ta1" "IH"  "b^" "Tb")
(elim "Tb")
(ng #t)
(use "TotalBooleTrue")
(assume "q^1" "Tq1" "b^1" "Tb1" "IHqb1")
(elim "Tp1")
(elim "Tq1")
(ng #t)
(use "IH")
(use "Tb1")
(ng #t)
(use "TotalBooleTrue")
(elim "Tq1")
(ng #t)
(use "TotalBooleFalse")
(ng #t)
(use "IH")
(use "Tb1")
;; Proof finished.
(save-totality)

;; Binary trees, or derivations

(add-algs "bin"
  '("bin" "BinNil")
  '("bin=>bin=>bin" "BinBranch"))
(add-totality "bin")
```

```
(add-var-name "r" "s" "t" (py "bin"))

(add-program-constant "NodeIn" (py "list boole=>bin=>boole"))

(add-infix-display-string "NodeIn" "nodein" 'rel-op)

(add-computation-rules
 "(Nil boole)nodein r" "True"
 "(p::a)nodein BinNil" "False"
 "(True::a)nodein BinBranch r s" "a nodein r"
 "(False::b)nodein BinBranch r s" "b nodein s")

;; NodeInTotal
(set-totality-goal "NodeIn")
(assert
 "allnc r^(TotalBin r^ -> allnc a^(TotalList a^ ->
   TotalBoole(a^ nodein r^)))")
 (assume "r^1" "Tr1")
 (elim "Tr1")
 (assume "a^" "Ta")
 (elim "Ta")
 (ng #t)
 (use "TotalBooleTrue")
 (strip)
 (ng #t)
 (use "TotalBooleFalse")
 (assume "r^" "Tr" "IHr" "s^" "Ts" "IHs" "a^1" "Ta1")
 (elim "Ta1")
 (ng #t)
 (use "TotalBooleTrue")
 (assume "p^" "Tp" "a^" "Ta" "Useless")
 (elim "Tp")
 (ng #t)
 (use "IHr")
 (use "Ta")
 (ng #t)
 (use "IHs")
 (use "Ta")
(assume "Assertion" "a^" "Ta" "r^" "Tr")
(use "Assertion")
```

```
(use "Tr")
(use "Ta")
;; Proof finished.
(save-totality)

(add-program-constant "Size" (py "bin=>nat"))

(add-computation-rules
 "Size BinNil" "1"
 "Size(BinBranch r s)" "Succ(Size r+Size s)")

;; SizeTotal
(set-totality-goal "Size")
(assume "r^1" "Tr1")
(elim "Tr1")
(ng #t)
(use "TotalNatSucc")
(use "TotalNatZero")
(assume "r^" "Tr" "IHr" "s^" "Ts" "IHs")
(ng #t)
(use "TotalNatSucc")
(use "NatPlusTotal")
(use "IHr")
(use "IHs")
;; Proof finished.
(save-totality)
```

We first define the Brouwer-Kleene ordering directly.

```
(add-program-constant "BK" (py "bin=>list list boole"))

(add-computation-rules
 "BK BinNil" "(Nil boole):"
 "BK(BinBranch r s)"
 "(([a]True::a)map BK r)++(([a]False::a)map BK s)++
  (Nil boole):")

;; BKTotal
(set-totality-goal "BK")
(assume "r^1" "Tr1")
(elim "Tr1")
(ng #t)
(use "TotalListCons")
```

```
(use "TotalListNil")
(use "TotalListNil")
(assume "r^" "Tr" "IHr" "s^" "Ts" "IHs")
(ng #t)
(use "ListAppdTotal")
(use "ListAppdTotal")
(use "ListMapTotal")
(assume "a^" "Ta")
(use "TotalListCons")
(use "TotalBooleTrue")
(use "Ta")
(use "IHr")
(use "ListMapTotal")
(assume "a^" "Ta")
(use "TotalListCons")
(use "TotalBooleFalse")
(use "Ta")
(use "IHs")
(use "TotalListCons")
(use "TotalListNil")
(use "TotalListNil")
;; Proof finished.
(save-totality)

(define testbin
  (pt "BinBranch(BinBranch BinNil(BinBranch BinNil BinNil))
               BinNil"))

(pp (nt (make-term-in-app-form (pt "BK") testbin)))

;; (True::True:)::
;; (True::False::True:)::
;; (True::False::False:)::(True::False:)::True: ::False: ::
;;   (Nil boole):

(add-var-name "as" "bs" "cs" (py "list list boole"))
```

We show that BK r is increasing w.r.t. LtBK.

```
(add-program-constant "Incr" (py "list list boole=>boole"))

(add-computation-rules
 "Incr(Nil list boole)" "True"
 "Incr a:" "True"
 "Incr(a::b::bs)" "a<<b andb Incr(b::bs)")

;; IncrTotal
(set-totality-goal "Incr")
(assert
 "allnc as^(TotalList as^ ->
  allnc a^(TotalList a^ -> TotalBoole(Incr(a^ ::as^))))")
 (assume "as^" "Tas")
 (elim "Tas")
 (ng #t)
 (strip)
 (use "TotalBooleTrue")
 (assume "a^" "Ta" "as^1" "Tas1" "IHas1" "a^1" "Ta1")
 (ng #t)
 (use "AndConstTotal")
 (use "LtBKTotal")
 (use "Ta1")
 (use "Ta")
 (use "IHas1")
 (use "Ta")
(assume "Assertion")
(assume "as^" "Tas")
(elim "Tas")
(ng #t)
(use "TotalBooleTrue")
(assume "a^" "Ta" "as^1" "Tas1" "Useless")
(use "Assertion")
(use "Tas1")
(use "Ta")
;; Proof finished.
(save-totality)

;; IncrAppd
(set-goal "all as(Incr as -> all bs(Incr bs ->
           all n(n<Lh as -> (n thof as)<<(0 thof bs)) ->
           Incr(as++bs)))")
(ind)
(ng #t)
(auto)
(ng #t)
```

```
(assume "a")
(cases)
(ng #t)
(assume "Useless1" "Useless2")
(cases)
(ng #t)
(strip)
(use "Truth")
(ng #t)
(assume "a1" "as" "Incr(a1::as)" "Hyp")
(split)
(use-with "Hyp" (pt "0") "Truth")
(use "Incr(a1::as)")
;; Step
(ng #t)
(assume "a0" "as" "IH" "Conj" "bs" "Incr bs" "Hyp")
(split)
(use "Conj")
(use "IH")
(use "Conj")
(use "Incr bs")
(assume "n")
(use-with "Hyp" (pt "Succ n"))
;; Proof finished.
(save "IncrAppd")

;; IncrMap
(set-goal
 "all as(Incr as -> all p Incr((Cons boole)p map as))")
(ind)
(strip)
(use "Truth")
(assume "a")
(cases)
(ng #t)
(strip)
(use "Truth")
(assume "a1" "as" "IH")
(ng #t)
(assume "Conj")
(cases)
```

```
(ng #t)
(split)
(use "Conj")
(use "IH")
(use "Conj")
(ng #t)
(split)
(use "Conj")
(use "IH")
(use "Conj")
;; Proof finished.
(save "IncrMap")

;; IncrTrueFalse
(set-goal "all as,n(n<Lh as ->
 all bs(0<Lh bs -> (n thof(Cons boole)True map as)<<
                   (0 thof(Cons boole)False map bs)))")
(ind)
(assume "n" "Absurd")
(use "Efq")
(use "Absurd")
(assume "a" "as" "IHas")
(cases)
(ng #t)
(assume "Useless")
(cases)
(assume "Absurd")
(use "Efq")
(use "Absurd")
(ng #t)
(strip)
(use "Truth")
(ng #t)
(use "IHas")
;; Proof finished.
(save "IncrTrueFalse")

;; LtBKNil
(set-goal "all a(0<Lh a -> a<<(Nil boole))")
(cases)
(assume "Absurd")
```

```
(use "Efq")
(use "Absurd")
(strip)
(use "Truth")
;; Proof finished.
(save "LtBKNil")

;; LhBK
(set-goal "all r 0<Lh(BK r)")
(cases)
(use "Truth")
(strip)
(use "Truth")
;; Proof finished.
(save "LhBK")

;; ListAppdProp
(set-goal
 "all as(all n(n<Lh as -> (Pvar list boole)(n thof as)) ->
  all bs(all n(n<Lh bs -> (Pvar list boole)(n thof bs)) ->
         all n(n<Lh as+Lh bs ->
               (Pvar list boole)(n thof as++bs))))")
(assume "as" "asHyp" "bs" "bsHyp" "n" "n<Lh as+Lh bs")
(use "NatLeLtCases" (pt "Succ n") (pt "Lh as"))
(assume "Succ n<=Lh as")
(assert "n<Lh as")
 (use "NatSuccLeToLt")
 (use "Succ n<=Lh as")
(assume "n<Lh as")
(drop "Succ n<=Lh as")
(simp "ListProjAppdLeft")
(use "asHyp")
(use "n<Lh as")
(use "n<Lh as")
;; Case Lh as<Succ n
(assume "Lh as<Succ n")
;; For later use we note
(assert "n--Lh as<Lh bs")
 (simp (pf "Lh bs=Lh as+Lh bs--Lh as"))
 (use "NatLtMonMinusLeft")
 (use "n<Lh as+Lh bs")
 (use "NatLtSuccToLe")
 (use "Lh as<Succ n")
 (ng #t)
 (use "Truth")
(assume "n--Lh as<Lh bs")
(assert "n=Lh as+(n--Lh as)")
 (simp "NatPlusMinus")
 (simp "NatPlusComm")
 (ng #t)
 (use "Truth")
 (use "NatLtSuccToLe")
 (use "Lh as<Succ n")
(assume "n=Lh as+(n--Lh as)")
(simp "n=Lh as+(n--Lh as)")
(simp "ListProjAppdRight")
(use "bsHyp")
(use "n--Lh as<Lh bs")
(use "n--Lh as<Lh bs")
;; Proof finished.
(save "ListAppdProp")

;; IncrBK
(set-goal "all r Incr(BK r)")
(ind)
(ng #t)
(use "Truth")
;; Step
(assume "r" "IHr" "s" "IHs")
(ng #t)
(use "IncrAppd")
(use "IncrAppd")
(use "IncrMap")
(use "IHr")
(use "IncrMap")
(use "IHs")
(simp "LhMap")
(assume "n" "Hyp")
(use "IncrTrueFalse")
(use "Hyp")
(use "LhBK")
(ng #t)
```

```
(use "Truth")
(ng #t)
(use "ListAppdProp")
(simp "LhMap")
(assume "n" "n<Lh(BK r)")
(simp "ListProjMap")
(use "Truth")
(use "n<Lh(BK r)")
(simp "LhMap")
(assume "n" "n<Lh(BK s)")
(simp "ListProjMap")
(use "Truth")
(use "n<Lh(BK s)")
;; Proof finished.
(save "IncrBK")

;; ExBK
(set-goal "all r ex as(Lh as=Size r &
                      all n(n<Lh as -> (n thof as)nodein r) &
                      Incr as)")
(ind)
```

The base case is easily done, by providing `(Nil boole)`:

```
(ex-intro "(Nil boole)"))
(ng #t)
(split)
(use "Truth")
(split)
(cases)
(ng #t)
(strip)
(use "Truth")
(assume "n" "Absurd")
(use "Efq")
(use "Absurd")
(use "Truth")
```

For the step case append the lists resulting from the two induction hypotheses and add the singleton list `(Nil boole):` at the end.

```
(assume "r" "IHr" "s" "IHs")
(by-assume "IHr" "as" "asProp")
(by-assume "IHs" "bs" "bsProp")
(ex-intro "(([a]True::a)map as)++
            (([a]False::a)map bs)++(Nil boole):")
```

Next we split the conjunctive goal. The first subgoal

```
?_24:Lh((([a]True::a)map as)++
       (([a]False::a)map bs)++(Nil boole):)
=Size(BinBranch r s)
```

concerns the size and is easily proved using `LhMap` (in `list.scm`) and the induction hypotheses.

```
(ng #t)
(simp "LhMap")
(simp "LhMap")
(simp "asProp")
(simp "bsProp")
(use "Truth")
```

Next we split again and obtain after simplification with `LhAppd` and `LhMap`

```
(split)
(simp "LhAppd")
(simp "LhAppd")
(simp "LhMap")
(simp "LhMap")
(ng #t)

?_37:all n(
      n<Succ(Lh as+Lh bs) ->
      (n thof
       ((Cons boole)True map as)++
       ((Cons boole)False map bs)++(Nil boole):)nodein
      BinBranch r s)
```

We assume the premise `n<Succ(Lh as+Lh bs)` and after applying some arithmetical lemmas arrive at a case distinction with `n<Lh as` first.

```
(assume "n" "n<Succ(Lh as+Lh bs)")
(use "NatLtSuccCases" (pt "n") (pt "Lh as+Lh bs"))
(use "n<Succ(Lh as+Lh bs)")
(assume "n<Lh as+Lh bs")
(use "NatLeLtCases" (pt "Succ n") (pt "Lh as"))
(assume "Succ n<=Lh as")
(assert "n<Lh as")
 (use "NatSuccLeToLt")
 (use "Succ n<=Lh as")
(drop "Succ n<=Lh as")
(assume "n<Lh as")
```

```
?_50:(n thof ((Cons boole)True map as)++
             ((Cons boole)False map bs)++(Nil boole):)nodein
     BinBranch r s
```

Now applying `ListProjAppdLeft` (twice), the induction hypothesis `asProp` and some list lemmas settles this case.

```
(simp "ListProjAppdLeft")
(simp "ListProjAppdLeft")
(simp "ListProjMap")
(ng #t)
(use "asProp")
(use "n<Lh as")
(use "n<Lh as")
(simp "LhMap")
(use "n<Lh as")
(simp "LhAppd")
(simp "LhMap")
(simp "LhMap")
(use "n<Lh as+Lh bs")

?_44:Lh as<Succ n ->
     (n thof ((Cons boole)True map as)++
             ((Cons boole)False map bs)++(Nil boole):)nodein
     BinBranch r s
```

Again we assume the premise and (for later use) prove `n--Lh as<Lh bs`. Then `ListProjAppdLeft` can be used again, and after some easy arithmetic

```
(assume "Lh as<Succ n")
(assert "n--Lh as<Lh bs")
 (simp (pf "Lh bs=Lh as+Lh bs--Lh as"))
 (use "NatLtMonMinusLeft")
 (use "n<Lh as+Lh bs")
 (use "NatLtSuccToLe")
 (use "Lh as<Succ n")
 (ng #t)
 (use "Truth")
(assume "n--Lh as<Lh bs")
(simp "ListProjAppdLeft")
(assert "n=Lh as+(n--Lh as)")
 (simp "NatPlusMinus")
 (simp "NatPlusComm")
 (ng #t)
 (use "Truth")
 (use "NatLtSuccToLe")
 (use "Lh as<Succ n")
(assume "n=Lh as+(n--Lh as)")
(simp "n=Lh as+(n--Lh as)")
(assert "Lh((Cons boole)True map as)=Lh as")
 (use "LhMap")
(assume "Lh((Cons boole)True map as)=Lh as")
(simp "<-" "Lh((Cons boole)True map as)=Lh as")
```

we arrive at

```
?_87:(Lh((Cons boole)True map as)+
     (n--Lh((Cons boole)True map as))thof
      ((Cons boole)True map as)++
      ((Cons boole)False map bs))nodein
     BinBranch r s
```

This now is a case for `ListProjAppdRight`, and `bsProp` with some easy arguments suffices to finish this case.

```
(simp "ListProjAppdRight")
(simp "ListProjMap")
(ng #t)
(use "bsProp")
(simp "LhMap")
(use "n--Lh as<Lh bs")
(simp "LhMap")
(use "n--Lh as<Lh bs")
(simp "LhMap")
(simp "LhMap")
(use "n--Lh as<Lh bs")
(simp "LhAppd")
(simp "LhMap")
(simp "LhMap")
(use "n<Lh as+Lh bs")

?_41:n=Lh as+Lh bs ->
     (n thof ((Cons boole)True map as)++
             ((Cons boole)False map bs)++(Nil boole):)nodein
             BinBranch r s
```

This final case again needs `ListProjAppdRight`

```
(assert "Lh as=Lh((Cons boole)True map as)")
 (simp "LhMap")
 (use "Truth")
(assume "Lh as=Lh((Cons boole)True map as)")
(simp "Lh as=Lh((Cons boole)True map as)")
(assert "Lh bs=Lh((Cons boole)False map bs)")
 (simp "LhMap")
 (use "Truth")
(assume "Lh bs=Lh((Cons boole)False map bs)")
(simp "Lh bs=Lh((Cons boole)False map bs)")
(simp "<-" "LhAppd")
(assume "n=Lh(((Cons boole)True map as)++
             ((Cons boole)False map bs))")
 (assert "n=Lh(((Cons boole)True map as)++
              ((Cons boole)False map bs))+0")
 (use "n=Lh(((Cons boole)True map as)++
           ((Cons boole)False map bs))")
(assume "n=Lh(((Cons boole)True map as)++
             ((Cons boole)False map bs))+0")
(simp "n=Lh(((Cons boole)True map as)++
           ((Cons boole)False map bs))+0")
(simp "ListProjAppdRight")
(ng #t)
(use "Truth")
(use "Truth")

?_32:Incr((([a]True::a)map as)++
          (([a]False::a)map bs)++(Nil boole):)
```

Now finally we need properties of `Incr`. First the induction hypotheses for `as` and `bs` imply that both the first and the second part of our list of booleans increase. We then use `IncrTrueFalse` to show that all elements of the first part are before the first element of the second part. It only remains to prove that every element of the first and the second part come before the final `(Nil boole):`, which again is easy.

```
(use "IncrAppd")
(use "IncrAppd")
(use "IncrMap")
(use "asProp")
(use "IncrMap")
(use "bsProp")
(simp "LhMap")
(assume "n" "Hyp")
(use "IncrTrueFalse")
```

```
(use "Hyp")
(simp "bsProp")
(assert "all t 0<Size t")
 (cases)
 (use "Truth")
 (strip)
 (use "Truth")
(assume "all t 0<Size t")
(use "all t 0<Size t")
(ng #t)
(use "Truth")
(ng #t)
(use "ListAppdProp")
(simp "LhMap")
(assume "n" "n<Lh as")
(simp "ListProjMap")
(use "Truth")
(use "n<Lh as")
(simp "LhMap")
(assume "n" "n<Lh bs")
(simp "ListProjMap")
(use "Truth")
(use "n<Lh bs")
;; Proof finished.
(save "ExBK")
```

This concludes the proof. For use (in 4.1.5) we have saved it under the name `ExBK`.

APPENDIX C

Useful commands

C.1. Emacs

- Start Emacs: `emacs &`
- Leave Emacs: C-x C-c
- Split a window in two: `C-x 2`
- Move to another Buffer: `C-x b` (then specify the Buffer's name)
- Move to another window: `C-x o`
- Load a file: `C-x C-f` (then give a name of a file with extension `.scm`)
- Save a file: `C-x C-s`
- Exit from the Minibuffer: `C-g`

C.2. Scheme

- Load (Petite) Scheme: `M-x run-petite`
- Evaluate a Scheme expression: `C-x C-e`
- Evaluate a region: mark the region and then `C-c C-r`
- Kill a process: `C-c C-c`
- Leave the Debug: `r`
- End a Scheme session: `(exit)`
- Comment: `;`

`C = Control` (or `Strg`), `M = Meta` (or `Edit` or `Esc` or `Alt`).

C.3. Minlog

The following is a list of commands which could be used in a "standard" interactive proof with Minlog. Rather than explaining the commands in detail (many of them have been demonstrated above), we shall write them down, often with a short description of their use gathered from the reference manual. The reader is advised to check the full details with the reference manual.

C.3.1. Some declarations needed to start a proof.

```
(add-tvar-name name1 ...)
(add-algs ...)
```

```
(add-var-name name1 ... type)
(add-predconst-name name1 ... arity)
(add-pvar-name name1 ... type)
(add-program-constant name type <rest>)
(add-computation-rule lhs rhs)
(add-rewrite-rule lhs rhs)
(add-global-assumption name formula)
```
(abbr. `aga`)

For each introduction command above there corresponds another one having the effect of removing the item so introduced (constants, variables, etc). For example:

`(remove-predconst-name name1 ...)`

There are also numerous display commands, in particular the following:

```
(display-pconst name1 ...).
(display-alg alg-name1 ...)
(display-idpc idpc-name1 ...)
(display-global-assumptions string1 ...)
(display-theorems string1 ...)
```

For types, terms and formulas there is a command `(pp object)` (for pretty-print), which tries to insert useful line breaks. Variants are `(ppc object)` (for pretty-print with case display) and `(pp-subst substitution)` (for pretty-printing substitutions).

`rename-variables` renames bound variables in terms, formulas and comprehension terms.

C.3.2. Goals.

(1) `(set-goal formula)` where *formula* needs to be closed (if it not so, then universal quantifiers will be inserted automatically).
(2) `(normalize-goal . ng-info)` (abbreviated `ng`) takes optional arguments `ng-info`. If there are none, the goal formula and all hypotheses are normalized. Otherwise exactly those among the hypotheses and the goal formula are normalized whose numbers (or names, or just `#t` for the goal formula) are listed as additional arguments.
(3) `(display-current-goal)` (abbr. `dcg`).

C.3.3. Generating interactive proofs.

Implication

`(assume x1...)`

moves the antecedent of a goal in implication form to the hypotheses. The hypotheses, *x1*..., should be identified by numbers or strings.

`(use x)`
where x is

- a number or string identifying a hypothesis from the context,
- the string "Truth",
- the name of a theorem or global assumption.
- a closed proof,
- a formula with free variables from the context, generating a new goal.

Conjunction

`(split)`
expects a conjunction $A \wedge B$ as goal and splits it into two new goals, A and B.

`(use x . elab-path)`
where x is as in the description of the `use` command for implication and *elab-path* consists of `'left` or `'right`.

Universal Quantifier

`(assume x1...)`
moves universally quantified variables into the context. The variables need to be named (by using previously declared names of the appropriate types).

`(use x . terms)`
where x is as in the case of implication and the optional *terms* is here a list of terms. When pattern unification succeeds in finding appropriate instances for the quantifiers in the goal, then these instances will be automatically inserted. However, one needs to explicitly provide terms for those variables that cannot be automatically instantiated by pattern unification.

Existential Quantifier

`(ex-intro term)`
by this command the user provides a term to be used for the present (existential) goal.

`(ex-elim x)`,
where x is

- a number or string identifying an existential hypothesis from the context,
- the name of an existential global assumption or theorem,
- a closed proof on an existential formula,
- an existential formula with free variables from the context, generating a new goal.

Classical Existential Quantifier

`(exc-intro terms)`
this command is analogous to `(ex-intro)`, but it is used in the case of a classical existential goal.

`(exc-elim` *x*`)`
this corresponds to `(ex-elim)` and applies to a classical existential quantifier.

C.3.4. Other general commands. `(use-with` *x* `.` *x-list*`)`
is a more verbose form of `use`, where the terms are not inferred via unification, but have to be given explicitly. Here x is as in `use`, and *x-list* is a list consisting of

- a number or string identifying a hypothesis form the context,
- the name of a theorem or global assumption,
- a closed proof,
- the string "?" generating a new goal,
- `'left` or `'right`,
- a term, whose free variables are added to the context.

`(inst-with` *x* `.` *x-list*`)`
does for forward chaining the same as `use-with` for backward chaining. It adds a new hypothesis which is an instance of a selected hypothesis or of a theorem. Here x and *x-list* are as in `use-with`.

`(inst-with-to` *x* `.` *x-list* `name-hyp)`
expects a string as its last argument, to name the newly introduced instantiated hypothesis.

`(cut` *A*`)`
replaces the goal B by the two new goals A and $A \to B$, with $A \to B$ to be proved first. Note that the same effect can also be produced by means of the `use` command.

`(assert` *A*`)`
replaces the goal B by the two new goals A and $A \to B$, with A to be proved first.

`(ind)`
expects a goal $\forall_{x^\rho} A$ with ρ an algebra. If $c_1, \dots, c_n$ are the constructors of the algebra ρ, then `(ind)` will generate n new goals:

$$\forall_{\vec{x}_i}(A[x := x_{1i}] \to \dots \to A[x := x_{ki}] \to A[x := c_i\vec{x}_i]).$$

`(simind` *all-formula1*`...)`
expects a goal $\forall_{x^\rho} A$ with ρ an algebra. The user provides other formulas to be proved simultaneously with the given one.

`(cases)`
expects a goal $\forall_{x^\rho} A$ with ρ an algebra. Assume that $c_1, \dots, c_n$ are the constructors of the algebra ρ. Then n new (simplified) goals $\forall_{\vec{x}_i} A[x := c_i\vec{x}_i]$ are generated.

`(simp x)`
expects a known fact of the form $r^{\mathbf{B}}$, $\neg r^{\mathbf{B}}$, $t = s$ or $t \approx s$. In case $r^{\mathbf{B}}$, the

boolean term r in the goal is replaced by T, and in case $\neg r^{\mathbf{B}}$ it is replaced by F. If $t = s$ (resp. $t \approx s$), the goal is written in the form $A[x := t]$. Using Compat-Rev (i.e. $\forall_{x,y}(x = y \to Py \to Px)$) (resp. Eq-Compat-Rev (i.e. $\forall_{x,y}(x \approx y \to Py \to Px)$)) the goal $A[x := t]$ is replaced by $A[x := s]$, where P is $\{\, x \mid A \,\}$, x is t and y is s. Here x is

- a number or string identifying a hypothesis form the context,
- the name of a theorem or global assumption, or
- a closed proof,
- a formula with free variables from the context, generating a new goal.

`(name-hyp` *`i x1`*`)`
expects an index i and a string. Then a new goal is created, which differs from the previous one only in display aspects: the string names the ith hypothesis.

`(drop .  x-list)`,
hides (but does not erase) the hypothesis listed in `x-list`. If `x-list` is empty, all hypotheses are hidden.

`(by-assume` *`x yu`*`)`
is used when proving a goal G from an existential hypothesis $ExHyp$: $\exists y A$. It corresponds to saying "by $ExHyp$ assume we have a y satisfying A". Here x identifies an existential hypothesis, and we assume the variable y and the kernel A (with label u). This command corresponds to the sequence `(ex-elim` *`x`*`)`, `(assume` *`y u`*`)`, `(drop` *`x`*`)`.

`(intro i . terms)`
expects as goal an inductively defined predicate. The i-th introduction axiom for this predicate is applied, via `use` (hence `terms` may have to be provided).

`(elim` *`idhyp`*`)`
Recall that $I\vec{r}$ provides (i) a type substitution, (ii) a predicate instantiation, and (iii) the list $\vec{r}$ of argument terms. In `(elim` *`idhyp`*`)` *idhyp* is, with an inductively defined predicate I,

- a number or string identifying a hypothesis $I\vec{r}$ form the context
- the name of a global assumption or theorem $I\vec{r}$;
- a closed proof of a formula $I\vec{r}$;
- a formula $I\vec{r}$ with free variables from the context, generating a new goal.

Then the (strengthened) elimination axiom is used with $\vec{r}$ for $\vec{x}$ and *idhyp* for $I\vec{r}$ to prove the goal $A(\vec{r})$, leaving the instantiated (with $\{\, \vec{x} \mid A(\vec{x}\,) \,\}$) clauses as new goals.

`(elim)`
expects a goal $I\vec{r} \to A(\vec{r})$. Then the (strengthened) clauses are generated as new goals, via `use-with`.

`(undo)` or `(undo` *n*`)`
has the effect of cancelling the last step in a proof, or the last n steps, respectively.

C.3.5. Automation and search.

`(strip)`
moves all universally quantified variables and hypotheses of the current goal into the context.

`(strip n)`
does the same as `(strip)` but only for n variables or hypotheses.

`(proceed)`
automatically refines the goal as far as possible as long as there is a unique proof. When the proof is not unique, it prompts us with the new refined goal, and allows us to proceed in an interactive way.

`(prop)`
searches for a proof of the stated goal. It is devised for propositional logic only.

`(search` *m (name1 m1)* `...)`
expects for m a default value of multiplicity (i.e. a positive integer stating how often the assumptions are to be used). Here *name1* ... are

- numbers or names of hypotheses from the present context or
- names of theorems or global assumptions,

and *m1* ... indicate the multiplicities of the specific *name1* .... To exclude a hypothesis one can list it with multiplicity 0.

`(auto` *m (name1 m1)* `...)`
It can be convenient to automate (the easy cases of an) interactive proof development by iterating `search` as long as it is successful in finding a proof. Then the first goal where it failed is presented as the new goal. `auto` takes the same arguments as `search`.

C.3.6. Displaying proofs objects. There are many ways to display a proof. We often use `display-proof` for a linear representation, showing the formulas and the rules used. We also provide a readable type-free lambda expression via `proof-to-expr`, and we can add useful information with either `proof-to-expr-with-formulas` or `proof-to-expr-with-aconsts`. In case the optional proof argument is not present, the current proof is taken instead.

`(display-proof .` *opt-proof*`)` abbreviated `dp`, `(display-normalized-proof . `*`opt-proof`*`)` abbreviated `dnp`,

`(proof-to-expr . `*`opt-proof`*`)`,

`(proof-to-expr-with-formulas . `*`opt-proof`*`)`,

`(proof-to-expr-with-aconsts . `*`opt-proof`*`)`.

Here `display-normalized-proof` normalizes the proof first. When in addition one wants to check the correctness of the proof, use

`(check-and-display-proof . `*`opt-proof-and-ignore-deco-flag`*`)`

abbreviated `cdp`. `ignore-deco-flag` is set to true as soon as the present proof argument proves a formula of nulltype.

C.3.7. Searching for theorems. It is a practical problem to find existing theorems or global assumptions relevant for the situation at hand. To help searching for those we provide

`(search-about `*`symbol-or-string`*` . `*`opt-strings`*`)`.

It searches in `THEOREMS` and `GLOBAL-ASSUMPTIONS` for all items whose name contains each of the strings given, excluding the strings Total Partial CompRule RewRule Sound. It one wants to list all these as well, take the symbol 'all as first argument.

Bibliography

Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer Verlag, Berlin, Heidelberg, New York, 1993.

Ulrich Berger. Uniform Heyting arithmetic. *Annals of Pure and Applied Logic*, 133:125–148, 2005.

Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

Victor P. Chernov. Constructive operators of finite types. *Journal of Mathematical Science*, 6:465–470, 1976. Translated from *Zapiski Nauch. Sem. Leningrad*, vol. *32*, pp. 140–147 (1972).

Albert Dragalin. New kinds of realizability. In *Abstracts of the 6th International Congress of Logic, Methodology and Philosophy of Sciences*, pages 20–24, Hannover, Germany, 1979.

Roy Dyckhoff. Contraction–free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57:793–807, 1992.

Yuri L. Ershov. Model C of partial continuous functionals. In R. Gandy and M. Hyland, editors, *Logic Colloquium 1976*, pages 455–467. North-Holland, Amsterdam, 1977.

Harvey Friedman. Classically and intuitionistically provably recursive functions. In D.S. Scott and G.H. Müller, editors, *Higher Set Theory*, volume 669 of *Lecture Notes in Mathematics*, pages 21–28. Springer Verlag, Berlin, Heidelberg, New York, 1978.

Gerhard Gentzen. Untersuchungen über das logische Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

David Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–190, 1925.

Jörg Hudelmaier. *Bounds for Cut Elimination in Intuitionistic Propositional Logic*. PhD thesis, Mathematische Fakultät, Eberhard–Karls–Universität Tübingen, 1989.

Hajime Ishihara. A note on the Gödel–Gentzen translation. *Mathematical Logic Quarterly*, 46:135–137, 2000.

Andrey N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Math. Zeitschr.*, 35:58–65, 1932.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

Dale Miller. A logic programming language with lambda–abstraction, function variables and simple unification. *Journal of Logic and Computation*, 2(4):497–536, 1991.

Helmut Schwichtenberg. Proof search in minimal logic. In B. Buchberger and J.A. Campbell, editors, *Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 2004, Proceedings*, volume 3249 of *LNAI*, pages 15–25. Springer Verlag, Berlin, Heidelberg, New York, 2004.

Helmut Schwichtenberg and Stanley S. Wainer. *Proofs and Computations*. Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press, 2012.

Dana Scott. Outline of a mathematical theory of computation. Technical Monograph PRG–2, Oxford University Computing Laboratory, 1970.

Viggo Stoltenberg-Hansen, Edward Griffor, and Ingrid Lindström. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

Index

CA, 31
CV, 31
$\tilde{\exists}, \tilde{\vee}$, 7
$\tilde{\wedge}$, 7

`add-algs`, 99
`add-computation-rule`, 100
`add-global-assumption`, 100
`add-predconst-name`, 100
`add-program-constant`, 100
`add-rewrite-rule`, 100
`add-tvar-name`, 99
`add-pvar-name`, 100
`add-var-name`, 100
`aga`, 100
AndU, 32
`assert`, 102
`assume`, 100, 101
`auto`, 104

Berger, 27
`by-assume`, 103

`cases`, 102
`cdp`, 105
`check-and-display-proof`, 105
Chernov, 15
clause, 32
conjunction, 4
Constable, 47
continuation passing style, 53
conversion
  permutative, 6
`cut`, 102

`dcg`, 100
decoration
  algorithm, 48
  of proofs, 48
derivable
  classically, 8
  intuitionistically, 8
disjunction, 3
`display-alg`, 100
`display-current-goal`, 100
`display-global-assumptions`, 100
`display-idpc`, 100
`display-normalized-proof`, 105
`display-pconst`, 100
`display-proof`, 104
`display-theorems`, 100
`dnp`, 105
`dp`, 104
drinker formula, 11
`drop`, 103

`elim`, 103, 104
elimination axiom, 32
EqD, 32
equality
  Leibniz, 30
Ershov, 15
even number, 29
`ex-elim`, 101
ex-falso-quodlibet, 7
`ex-intro`, 101
`exc-elim`, 102
`exc-intro`, 101
existential quantifier, ix, 4
ExU, 32

formula, 28
  computationally relevant (c.r.), 28

decorated, 28
non-computational (n.c.), 28, 31

Gödel, 16
`GLOBAL-ASSUMPTIONS`, 105
Gödel-Gentzen translation, 12

Harrop formula, 27, 28
Harrop predicate, 28
Hilbert, 16

`ignore-deco-flag`, 105
`ind`, 102
`inst-with`, 102
`inst-with-to`, 102
`intro`, 103
introduction axiom, 32

least-fixed-point operator, 28
Leibniz equality, 30, 32

Martin-Löf, 15

n.c., 27
`name-hyp`, 103
`ng`, 100
`normalize-goal`, 100
nulltype, 28

one-clause-nc, 32

permutative conversion, 6
`pp`, 65, 100
`pp-subst`, 100
`ppc`, 100
predicate, 28
computationally relevant (c.r.), 28
decorated, 28
non-computational (n.c.), 28
`proceed`, 104
proof
pattern, 47
`proof-to-expr`, 105
`proof-to-expr-with-aconsts`, 105
`proof-to-expr-with-formulas`, 105
`prop`, 104

realizability, 34
`remove-predconst-name`, 100
`rename-variables`, 100

Scott, 15
`search`, 104
`search-about`, 105
`set-goal`, 100
`simind`, 102
`simp`, 102
soundness theorem
for realizability, 36
`split`, 101
stability, 8
strictly positive, 28
`strip`, 104

term
extracted, 35
`THEOREMS`, 105

`undo`, 104
uninstantiated formula
of an axiom, 48
`use`, 101
`use-with`, 102

variable
computational, 31
non-computational, 31
variable condition, vii, 4

witnessing predicate, 35