Minlog — An Interactive Prover

Helmut Schwichtenberg

December 2, 1997

Contents

1 Introduction **4**

1.1 Proofs about programs and normalization 4
1.2 Proofs as programs 4
1.3 Extracted programs 5
1.4 The MINLOG object hierarchy 6
1.5 Acknowledgements 6

2 Object language and semantics **7**

2.1 Types 7
2.1.1 Ground types 7
2.1.2 Composing types 8
2.1.3 Accessing types 9
2.1.4 Internals 9
2.2 Denotational semantics for simple type theory 9
2.3 Terms 12
2.3.1 Variables 12
2.3.2 Program constants 13
2.3.3 Function symbols 16
2.4 Composing terms 17
2.4.1 The term grammar 17
2.4.2 Auxiliary construction functions 18
2.4.3 The type and t-degree of a term 18
2.4.4 Free variables and substitution 18
2.4.5 Parsing and displaying 19
2.4.6 Notations 20
2.5 Operational semantics: The reduction relation 20
2.5.1 λ-calculus: $\to_{\beta\eta\uparrow}$ $-reduction$ 20
2.5.2 Defined constants 21
2.5.3 Properties of normal forms 21
2.5.4 Front end 21

3 Formulas **22**

3.1 Atomic formulas 22
3.1.1 Constructors 22
3.1.2 Accessing atomic formulas 22

3.2 Composing formulas . . . 22
3.3 Free and bound variables and substitution . . . 23
3.4 How to display, check and parse formulas . . . 24
3.5 Internals . . . 24

4 Proofs **25**
4.1 Constructing proofs . . . 25
4.2 Free variables and substitution in proof terms . . . 27
4.3 Global assumptions . . . 27
4.3.1 Motivation . . . 27
4.3.2 Administrating global assumptions . . . 28
4.3.3 Accessing global assumptions . . . 28
4.3.4 Standard global assumptions . . . 28
4.3.5 Internals . . . 28
4.4 Axioms and axiom-scheme-forms . . . 29
4.4.1 Axioms . . . 29
4.4.2 Axiom-scheme-forms . . . 29
4.4.3 MINLOG's standard axioms . . . 29
4.4.4 Internals . . . 30
4.5 Normalisation . . . 30
4.6 Accessing proofs . . . 32
4.6.1 Decomposing proofs . . . 32
4.6.2 Check and display of proofs . . . 32

5 Interactive theorem proving **33**
5.1 Administrating goals . . . 33
5.2 Standard proof commands . . . 33
5.2.1 `set-goal` . . . 33
5.2.2 `display-current-goal` . . . 34
5.2.3 `normalize-goal` . . . 34
5.2.4 `assume` . . . 34
5.2.5 `strip` . . . 34
5.2.6 `unstrip` . . . 34
5.2.7 `drop` . . . 34
5.2.8 `split` . . . 34
5.2.9 `split-hyp` . . . 34
5.2.10 `use` . . . 35
5.2.11 `use-with` . . . 35
5.2.12 `cut` . . . 35
5.2.13 `inst` . . . 35
5.2.14 `prop` . . . 36
5.2.15 `search` . . . 36
5.3 Using the axiom-scheme-forms . . . 36
5.3.1 `ind` . . . 36

5.3.2 `cases` . . . 36
5.3.3 `cases-all^` . . . 37
5.3.4 `ex-intro` . . . 37
5.3.5 `ex-elim` . . . 37
5.4 Rearranging the partial proof . . . 37
5.4.1 `undo` . . . 38
5.4.2 `get` . . . 38
5.4.3 `pproof` . . . 38
5.5 Internals . . . 38

6 Extracting programs from proofs **39**
6.1 Modified realisability . . . 39
6.2 Front end . . . 41
6.3 Program extraction from classical proofs . . . 41
6.3.1 `classical-proof-to-constr-proof` . . . 41
6.3.2 `classical-proof-to-program` . . . 41

7 Normalization by evaluation **42**
7.1 Abstract normalization-by-evaluation . . . 42
7.2 Implementation . . . 42
7.2.1 The model . . . 42
7.2.2 Interpretation . . . 42
7.2.3 Quote and unquote . . . 43
7.2.4 Animation of program-constants and function-symbols . . . 43
7.2.5 Normalization-by-evaluation . . . 43
7.2.6 Normalization-by-evaluation for proof terms . . . 44
7.2.7 Internals . . . 44

8 The TEX-output **45**
8.1 How to output . . . 45
8.2 How to modify the output of types, terms and formulas . . . 46
8.2.1 `add-ground-type-information` . . . 46
8.2.2 `add-program-constant-information` . . . 46
8.2.3 `add-variable-information` and `variable-output-flags` . . . 49
8.2.4 `add-function-symbol-information` . . . 49
8.2.5 `add-program-scheme-form-information` . . . 50
8.3 How to modify the output of proofs . . . 50
8.4 Internals . . . 52

Chapter 1

Introduction

MINLOG is an interactive prover based on minimal logic. Its aims are

- to build formal proofs about programs,
- to provide a framework for using proofs as programs and manipulating proofs and
- to provide machine support for program development, e.g. adaption of programs to particular situations.

Programs are understood here as functional programs. Hence we choose a simply typed language as our basis; this seems to suffice for most applications. Since functional programs denote higher type computable functionals, we have to base our development on the theory of partial continuous functionals, for — as shown by Scott [17] and Ershov — these are the mathematically correct domains for computable functionals. Therefore our (typed) variables are always understood to range over the partial continuous functionals. Since this view is not particularly well known, we shortly review the history of computability in higher types in an appendix.

1.1 Proofs about programs and normalization

Normalization is central for to achieve the aims stated above. Why is that so? Programs can be written directly in SCHEME. The name for a program then is a constant (program constant) in the language. It is quite clear that closed terms involving program constants should evaluate to (e.g.) a number. It is also clear that we can use SCHEME evaluation to achieve that. Here our foremost concern is to carry out proofs about programs. Hence we need terms with free variables involving program constants. We want to simplify also such terms, using the same evaluation mechanism as before. This is possible; see [4].

We also want to normalize full proofs. This is possible by the very same mechanism if proofs are written in natural deduction style.

Normalization of proofs is an extremely useful tool. It can be thought of as a device that eliminates detours in proofs, and hence presents the proof in a pure, expanded form suitable for further analysis. This can e.g. be used to carry out the proofs–as–programs paradigm in our setting.

1.2 Proofs as programs

The proofs–as–programs paradigm has been stressed by Constable [2] as providing a suitable interface between mathematics and computer science. This is usually carried out in a setting where the underlying logic is intuitionistic. However, using this logic for mathematical proofs is still somewhat unusual, since some familiar logical laws do not hold any more (e.g. $(\forall x A(x) \to B) \to \exists x(A(x) \to B)$. It is one of our aims here to demonstrate that the proofs–as–programs paradigm is also useful in the intersection of classical and intuitionistic logic, or more precisely in minimal logic in the language determined by $\wedge, \to, \forall$.

However, pure logic clearly is not sufficient, since we want to reason about computable functionals (i.e. programs). So we immediately have to add arithmetical features. How should that be done? First of all we need induction axioms (more precisely: schemata) for the basic inductively generated data types like those of the natural numbers or the booleans. Another essential observation is the following: we only want to deal with decidable atomic formulas, e.g. equations between ground type objects. Hence it is convenient to represent an atom by a term of type boole. In particular we then have an atom given by the boolean constant "false", written `(atom false)` or else $\mathtt{F}$. Now we can add, for any atom P, the ex–falso–quodlibet axiom $\mathtt{F} \to P$ to our axioms. This gives falsity $\mathtt{F}$ a special status; without such an axiom it would behave in minimal logic just as any other propositional variable.

Note that the convention to only deal with decidable atoms excludes the use of inductively defined relations (like Kleene's $\mathcal{O}$). However, if one adheres to the idea that in a formal treatment of mathematics the objects to be studied should be made explicit anyway, then it is natural to accompany any instance of an inductively defined relation with a witness of its generation.

Having $\mathtt{F}$ in our language we can define negation, disjunction and the existential quantifier as usual. Moreover, since we take all atomic formulas to be decidable (given by a boolean term), we can easily prove that any formula A in our $\wedge, \to, \forall$–language satisfies the principle of indirect proof (stability axiom) $\neg\neg A \to A$ (by induction on A, using a case distinction if A is atomic). Therefore we have classical logic available, in spite of the fact that our basic logical rules are only those of minimal logic.

Why is it useful to have the minimal logical rules as a basis? If derivations are written in Gentzen's natural deduction style, then these rules amount to the familiar rules to build terms in the simply typed lambda–calculus: forming abstractions $\lambda x t$ and applications ts (for $\to, \forall$), or pairing and projections (for $\wedge$). Hence proofs are terms in the simply typed lambda–calculus.

Now consider a normal proof of a closed formula $\tilde{\exists} y A(y)$ with $A(y)$ quantifier–free and $\tilde{\exists}$ the classical existential quantifier, i.e. of $[\forall y(A(y) \to \mathtt{F})] \to \mathtt{F}$. This can be viewed as a proof of falsity $\mathtt{F}$ from a (false) assumption $u{:}\,\forall y(A(y) \to \mathtt{F})$. Each occurrence of this assumption u in the proof must be in a context

$$\dfrac{\dfrac{u{:}\,\forall y(A(y) \to \mathtt{F}) \qquad n}{A(n) \to \mathtt{F}} \qquad \begin{array}{c} | \\ A(n) \end{array}}{\mathtt{F}}$$

with a numeral n. Now it is easily seen that at least one of those $A(n)$ must be true (note that $A(n)$ is closed and quantifier–free, hence its truth can be checked). Searching for the "first" such $A(n)$ yields a "direct method" to use our given proof of $\tilde{\exists} y A(y)$ as a program to compute an instance. Note that the essential engine involved in "running" the instantiated proof as a program is normalization.

1.3 Extracted programs

However, there is one drawback in this approach. Proofs are rather long and usually contain many parts which are not relevant for the computational behaviour, e.g. subproofs of purely universal lemmata. Therefore it is desirable to also have a mechanism to extract a (hopefully short and perspicuous) program from a proof. In order to do that, we have to add the "strong" or intuitionistic existential quantifier $\exists^*$ to the language (as opposed to the classical quantifier $\exists$ defined by $\neg\forall\neg$). Then the well–known modified realizability interpretation can be applied to a proof of $\forall x \exists^* y A(x, y)$, yielding an "extracted term", called et, such that $\forall x A(x, \mathrm{et}(x))$ can be derived as well. Such devices are implemented e.g. in Nuprl [6] and in Coq [15].

Note that it is possible to deal with $\exists^*$ (and similarly with $\vee^*$) without changing our logical rules, i.e. those of minimal logic for $\wedge, \to, \forall$. We only have to add some axioms corresponding to the introduction and elimination rules for $\exists^*$ and $\vee^*$ (see the third example in [19]).

1.4 The Minlog object hierarchy

The Minlog object hierarchy consists of

- (Object) terms, having types,
- Formulas, involving terms,
- Proofs, deriving formulas,
- partial proofs, representing incomplete proofs that are constructed interactively.

1.5 Acknowledgements

The following people have been involved in the development of the system and the underlying theory: Holger Benl, Ulrich Berger, Michael Bopp, Matthias Eberl, Felix Joachimski, Karl–Heinz Niggl, Gunnar Reitel, Helmut Schwichtenberg, Monika Seisenberger, Anton Setzer, Robert Stärk, Michael Stoll, Klaus Weich, Wolfgang Zuber.

Chapter 2

Object language and semantics

Our object world is represented by a simply typed λ-calculus with pairing, constants and function symbols.

2.1 Types

are generated from a set of *basic* (or *ground*) *types* using the function and product type constructors $\to, *$.

2.1.1 Ground types

By default the set of basic types includes `boole` and `nat`. Further ground types can be added by

`(add-ground-type '`*type-name*`)` abbreviated `(agt '`*type-name*`)`.

Besides adding the *type-name* to the internal list of ground types the routine also induces a cascade of program-constant- and function-symbol-definitions that will be further documented below. For instance the call `(agt 'real)` provokes the following output.

```
;ok, ground type real added
;ok, self-evaluating program constant undef_real:real
;of t-degree 0 to be displayed undef_real added
;ok, function symbol def-real:(functionality real boole)
;of t-degree 2 to be displayed def added
;ok, program constant =-strict-real:real->real->boole
;of t-degree 1 to be displayed = added
;ok, function symbol eq_real:(functionality real real boole)
;of t-degree 2 to be displayed eq_real added
;ok, program constant if-real:boole->real->real->real
;of t-degree 1 to be displayed [...] added
```

Additionally, `add-ground-type` allows to introduce variable declarations (which otherwise would have to be formulated by `add-variable`) by simply adding the variable symbols as arguments. Thus `(agt 'real 'r 's)` adds the variables `r` and `s` with type `real`:

```
;ok, ground type real added
;ok, variable r:real added
;ok, variable s:real added
```

```
;ok, self-evaluating program constant undef_real:real
;of t-degree 0 to be displayed undef_real added
;ok, function symbol def-real:(functionality real boole)
;of t-degree 2 to be displayed def added
;ok, program constant =-strict-real:real->real->boole
;of t-degree 1 to be displayed = added
;ok, function symbol eq_real:(functionality real real boole)
;of t-degree 2 to be displayed eq_real added
;ok, program constant if-real:boole->real->real->real
;of t-degree 1 to be displayed [...] added
```

Ground types can be removed by `(remove-ground-type '`*type-name*`)` which may be abbreviated to `rgt` and also deletes all variables, program-constants and function-symbols the type of which contain the ground type in regard:

```
==> (rgt 'real)
;ok, variable r is removed
;ok, variable s is removed
;ok, program constant undef_real is removed
;ok, program constant = is removed
;ok, program constant xx is removed
;ok, function symbol def is removed
;ok, function symbol eq_real is removed
;ok, ground type real is removed
```

For technical reasons to be explained below MINLOG's standard list of ground types also includes `existential`, `atomic` and `nulltype`, which (in contrast to all other ground types) are called *non-object-types*.

2.1.2 Composing types

is done using the function type constructor $\to$ and the pair type constructor $*$.

For ease in reading MINLOG adopts the following notational conventions: $*$ associates to the left while $\to$ associates to the right. When both appear $*$ takes precedence over $\to$.

$$\begin{aligned} type_1 \to (type_2 \to type_3) &\equiv type_1 \to type_2 \to type_3, \\ (type_1 * type_2) * type_3 &\equiv type_1 * type_2 * type_3, \\ (type_1 * type_2) \to type_3 &\equiv type_1 * type_2 \to type_3. \end{aligned}$$

Constructor functions.

`(cons-arrow` $type_1\ type_2$`)`	$\Longrightarrow type_1 \to type_2$
`(c-arrow` $type_1\ type_2 \dots type_n\ type$`)`	$\Longrightarrow type_1 \to type_2 \to \dots type_n \to type$
`(cons-star` $type_1\ type_2$`)`	$\Longrightarrow type_1 * type_2$
`(c-list-type` $type_1\ type_2 \dots type_n$`)`	$\Longrightarrow type_1 * type_2 * \dots * type_n *$ `nulltype`.

Thus `nulltype` serves as the type of the empty list `nil`.

Destructor functions.

$$
\begin{array}{ll}
\texttt{(arg-type (cons-arrow } \mathit{type}_1\ \mathit{type}_2\texttt{))} & \Longrightarrow \mathit{type}_1, \\
\texttt{(val-type (cons-arrow } \mathit{type}_1\ \mathit{type}_2\texttt{))} & \Longrightarrow \mathit{type}_2, \\
\texttt{(arg-types (c-arrow } \mathit{type}_1\ \mathit{type}_2 \ldots \mathit{type}_n\ \mathit{type}\texttt{) } k\texttt{)} & \\
 & \Longrightarrow \texttt{(}\mathit{type}_1\ \mathit{type}_2 \ldots \mathit{type}_k\texttt{)}, \qquad k \leq n, \\
\texttt{(left-type (cons-star } \mathit{type}_1\ \mathit{type}_2\texttt{))} & \Longrightarrow \mathit{type}_1, \\
\texttt{(right-type (cons-star } \mathit{type}_1\ \mathit{type}_2\texttt{))} & \Longrightarrow \mathit{type}_2, \\
\texttt{(list-type-to-types (c-list-type (list } \mathit{type}_1\ \mathit{type}_2 \ldots \mathit{type}_n\texttt{)))} & \\
 & \Longrightarrow \texttt{(}\mathit{type}_1\ \mathit{type}_2 \ldots \mathit{type}_n\texttt{)}, \\
\texttt{(type-to-ground-types } \mathit{type}\texttt{)} & \Longrightarrow \text{a list of all ground types in } \mathit{type}.
\end{array}
$$

2.1.3 Accessing types

Test functions are `ground-type?`, `arrow-form?`, `star-form?`, `object-type?`, `list-type?` In contrast to terms and formulas, types cannot be displayed or parsed directly. Nevertheless, a function `type-to-string` exists which uses `->` to display the arrow and `*` for $*$, respecting the notational conventions.

2.1.4 Internals

Ground types are internally represented by their symbols. They are gathered in a list in the global scheme variable `GROUND-TYPES`[1]. Thus testing a type on being ground (`ground-type?`) amounts to testing membership (e.g., `(memq '`*type-name* `GROUND-TYPES)`).

Composed types are of the form `(arrow` *type*$_1$ *type*$_2$`)` or `(star` *type*$_1$ *type*$_2$`)`.[2]

2.2 Denotational semantics for simple type theory

As we want to allow reasoning about programs terminating with an error the intended semantics provides for partial computable functions and undefined values. The appendix outlines how Scott-domains provide for a suitable environment to model the related concepts. Once having accepted them as models for our typed lambda-calculus we may reflect part of their structure in our syntax by adding for instance constants $\bot_\iota$ for the undefined value at type ι.

Here we do not want to go into much detail but summarise only basic facts about Scott-domains. Thorough expositions of the field can be obtained from [12] or [16].

- A *partial order* is a pair $(D, \sqsubseteq)$ of a set D and a reflexive, transitive relation $\sqsubseteq$ on D.
- A partially ordered set D is *directed* if each finite subset M has an upper bound in D.
- A partial order $(D, \sqsubseteq)$ is directed-complete (a *dcpo*) if it has a least element $\bot$ and every directed subset M has a least upper bound $\bigsqcup M$. For the rest of this section let D, E, F denote dcpos. A *flat* dcpo $D_\bot$ is a set D together with a new element $\bot_D$ such that

$$d \sqsubseteq_{D_\bot} e \Leftrightarrow d = e \vee d = \bot_D.$$

[1] By notational convention global variables of MINLOG are written uppercase.

[2] As can be seen from the source code, MINLOG could be extended by further type constructors without too many problems: An association list `TYPE-INFIX-PRECEDENCES` serves to organise precedences of the type-constructors.

- A function $f : D \to E$ is *monotone* if

$$d \sqsubseteq_D d' \Rightarrow f(d) \sqsubseteq_E f(d').$$

- A monotone function $f : D \to E$ is *continuous* if for each directed subset $M \subset D$

$$f(\bigsqcup M) = \bigsqcup f(M).$$

  The set of continuous functions from D to E is denoted by $[D \to E]$. By Tarski's fixed point theorem any continuous function $f : D \to D$ on a dcpo D has a least fixed point given by $\bigsqcup_{n \in \mathbb{N}} f^n(\bot)$ (and the fixed point operator $\mathrm{fix} : [D \to D] \to D$ with $\mathrm{fix}(f) := \bigsqcup_{n \in \mathbb{N}} f^n(\bot)$ is again continuous).

- A function $f : D \to E$ is called *strict* if it preserves the bottom element, i.e., $f(\bot_D) = \bot_E$.

- A dcpo is called *lub-complete* if every directed subset contains its least upper bound. Of course, flat dcpos are lub-complete. A function $f : D \to E$ on a lub-complete dcpo is continuous already if it is monotonic.

  PROOF. Assume $M \subseteq D$ is directed with $M \ni m := \bigsqcup M$. Then $f(\bigsqcup M) = f(m) \sqsubseteq \bigsqcup_{m \in M} f(m) = \bigsqcup f(M)$. By monotonicity $\bigsqcup f(M) \sqsubseteq f(\bigsqcup M)$ holds.

- An element $k \in D$ is *compact* (or *finite*) if in every directed subset $M \subseteq D$ with $k \sqsubseteq \bigsqcup M$ there is a $y \in M$ such that $k \sqsubseteq y$. The set of compact elements of a dcpo D is called its *basis* and denoted by $\mathcal{K}(D)$.

- A dcpo $(D, \sqsubseteq)$ is *algebraic* if for each element $d \in D$ the set $\mathcal{K}(d) := \{e \in \mathcal{K}(D) \mid e \sqsubseteq d\}$ is directed and $d = \bigsqcup \mathcal{K}(d)$. It is called *$\omega$-algebraic* if the basis is countable. Important examples of algebraic dcpos are flat dcpos.

  [Algebraic dcpos allow for an easy ϵ–δ-characterisation of continuity: A monotone function $f : D \to E$ is continuous iff for all $d \in D, e' \in \mathcal{K}(f(d))$ there is an $e \in \mathcal{K}(d)$ with $f(e) \sqsupseteq e'$.]

- A dcpo $(D, \sqsubseteq)$ is *bounded complete* if every bounded subset $E \subset D$ (i.e., $E \sqsubseteq e$ for an $e \in D$) has a least upper bound.

- A *(Scott-)domain* is a bounded-complete algebraic dcpo.

  Both the category $\mathcal{DCPO}$ of dcpos and the category $\mathcal{DOM}$ of domains (objects are dcpos or domains, respectively, morphisms are continuous functions) are cartesian closed.[3] The essential ingredients for the proof of these facts are given by the following two domain constructions.

  - Given two dcpos $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ we can form the *product dcpo* $(D \times E, \sqsubseteq_{D \times E})$ by

  $$\langle d, e \rangle \sqsubseteq_{D \times E} \langle d', e' \rangle :\Leftrightarrow d \sqsubseteq_D d' \wedge e \sqsubseteq_E e'$$

    which again forms a Scott-domain if D and E were such. Pairing and projection are always continuous. The least element $\bot_{D \times E}$ is $\langle \bot_D, \bot_E \rangle$.

[3] In contrast to that of algebraic cpos.

– Given two dcpos (Scott-domains) $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ we can form $[D \to E]$, the *continuous function space*, which equipped with the pointwise order relation

$$f \sqsubseteq_{[D\to E]} g :\Leftrightarrow \forall d \in D. f(d) \sqsubseteq_E g(d)$$

is again a dcpo (Scott-domain). The least upper bounds of it's directed sets are calculated pointwise and it has the constant function $\perp_{[D\to E]} := \lambda d{\in}D.\perp_E$ as least element. The evaluation and curry functionals

$$\begin{array}{rcl} \mathrm{eval} : [D \to E] \times D & \to & E, \\ \langle f, x\rangle & \mapsto & f(x), \end{array} \qquad \begin{array}{rcl} \mathrm{curry} : [D \times E \to F] & \to & [D \to [E \to F]], \\ f & \mapsto & (d \mapsto (e \mapsto f(d,e))). \end{array}$$

are continuous.

If one considers the categories $\mathcal{DCPO}_\perp$ and $\mathcal{DOM}_\perp$ with strict continuous functions rather than just continuous ones as morphisms, the respective product construction is: Given two dcpos $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ we form the *smash product* dcpo $(D \otimes E, \sqsubseteq_{D\otimes E})$ by defining

$$\begin{array}{rcl} D \otimes E & := & \{\, x \in D \times E \mid x = \perp \vee (x = \langle d, e\rangle \wedge d \neq \perp \neq e\,\}, \\ x \sqsubseteq_{D\otimes E} y & :\Leftrightarrow & x \sqsubseteq_{D\times E} y, \end{array}$$

i.e., we identify all pairs with $\perp$ in one component. The canonical conversion function

$$\mathrm{smash} : D \times E \to D \otimes E, \qquad \mathrm{smash}(\langle d, e\rangle) : \begin{cases} \perp & \text{if } d = \perp \text{ or } e = \perp \\ \langle d, e\rangle & \text{otherwise,} \end{cases}$$

is strict as well as continuous. Smash product formation preserves both dcpos and domains.

Assuming a flat dcpo D_ι for every basic type we can construct a simple model of our type theory by taking

$$D_{\rho\to\sigma} := D_\rho \to D_\sigma$$

to interprete function types and

$$D_{\rho\times\sigma} := D_\rho \times D_\sigma$$

for product types. However, since continuity of currying and evaluation is strictly necessary for interpreting λ-abstraction and application, this model is not restrictive enough. Therefore we embed the Scott-model obtained by restricting to the continuous functions $D'_{\rho\to\sigma} := [D_\rho \to D_\sigma]$ for the arrow case $\rho \to \sigma$ on which we may safely use λ-abstraction.[4] For the interpretation it is vitally important that we do not use abstraction for objects in $D_{\rho\to\sigma} \setminus D'_{\rho\to\sigma}$!

Notice that continuous functions defined on D_ι are simply monotonic functions, since the D_ι are flat and therefore lub-complete.

- The sets $T_\rho \subseteq D_\rho$ of *total* objects are defined inductively by

$$\begin{array}{rcl} T_\iota & := & D_\iota \setminus \{\perp_\iota\}, \\ T_{\rho\to\sigma} & := & \{\, f \in D_{\rho\to\sigma}, \mid \forall d \in T_\rho. f(d) \in T_\sigma\,\}, \\ T_{\rho\times\sigma} & := & \{\, d \in D_{\rho\times\sigma} \mid \pi_0(d) \in T_\rho \wedge \pi_1(d) \in T_\sigma\,\}. \end{array}$$

[4]Remark: Since flat dcpos are Scott-domains and the category of Scott-domains is cartesian closed, we actually get a Scott-domain D'_ρ for each type ρ. This is important because the theory of compact elements of $D'_{\rho\to\sigma}$ is quite well understood in the case of algebraic dcpos [12].

- A functional $f \in D_{\rho \to \sigma}$ is *strongly total* if for arbitrary arguments $d \in D_\rho$ the value $f(d)$ is total.
- A functional $f \in D_{\rho \to \sigma}$ is *supertotal* if for arbitrary arguments $d \in D_\rho$ the value $f(d)$ is strongly total.

Given a functional $f \in D_{\rho \to \sigma}$ we define its totality degree (t-degree) by

$$\text{t-deg}(f) := \begin{cases} 3 & \text{if } f \text{ is supertotal,} \\ 2 & \text{if } f \text{ is strongly total, but not supertotal,} \\ 1 & \text{if } f \text{ is total, but not supertotal,} \\ 0 & \text{otherwise, i.e., if } f \text{ is partial.} \end{cases}$$

Note that curry can raise the totality degree of its argument! For example

$$\text{t-deg}(\underbrace{\text{curry} \circ \cdots \circ \text{curry}}_{n \text{ times}}(\pi_1)) = n + 1.$$

Another example with non-constant functions: Let $D = \{\bot, \top\}$ and $\max\colon D^m \to D$ be the maximum function on D. Then for $n < m$

$$\text{t-deg}(\underbrace{\text{curry} \circ \cdots \circ \text{curry}}_{n \text{ times}}(\max)) = n + 1.$$

2.3 Terms

Object terms are constructed from variables and program constants, using λ-abstraction, application, pairing, projection and application of function-symbols. Since all atomic objects and function-symbols are typed, the type of each term is uniquely determined.

2.3.1 Variables

Syntax

Variables are explicitly typed using variable declarations

`(add-variable '`*name type*`)`   abbreviated   `(av '`*name type*`)`

where *name* should consist of letters only. In order to have infinitely many variables available we also allow indices. Thus

```
(av 't 'nat)
```

declares e.g. the variables `t,t2,t345,t0034` as of type `nat`. As described above we distinguish between variables representing either total or partial objects. The latter are denoted by variable symbols with ^ affixed. Hence the above declaration also introduces `t^,t2^,t33^` for type `nat`.

For simplicity, variable declarations can already be given when introducing new ground types by simply adding a list of symbols:

`(agt '`*type-name* `'`*variable-name*$_1$ `'`*variable-name*$_2$ `...)`

Variable declarations are deleted by

`(remove-variable '`*name*$_1$ `'`*name*$_2$ `...)`   abbreviated   `(rv ...)`.

By default MINLOG contains the variable declarations

`k,l,m,n:nat    p,q:boole.`

Semantics

A variable is interpreted as a continuous functional of its type. Depending on its declaration it represents

- a partial object (totality-degree 0) if its name terminates with ^ and
- a total object (totality-degree 1) otherwise.

Accessing variables

`variable?` tests if its argument has been declared as a variable before. The type of a previously defined variable can be obtained by `variable-to-type`. In case we don't want to decide on a name, the function `(type-to-new-partial-variable` *type*`)` gives back a symbol (with ^) for a variable of the given type. For instance `(type-to-new-partial-variable (c-arrow 'boole 'nat 'nat 'real))` produces a variable of the form `boole=>nat=>nat=>real^001`.

Internals

Variable declarations are gathered in `VARIABLES` which is an association list of the form

```
((m nat) (n nat)...)
```

In order to test if a given symbol is a variable the function `variable?` has to ignore a potential ^ and indices. This is done by the functions `remove-final-^` and `remove-final-numeric-chars` respectively. Since the test is used rather often MINLOG remembers variable symbols together with their types and totality-degree in an additional property list `VAR-PLISTS` of the form

```
((p^   ((type boole) (t-deg 0)))
 (p    ((type boole) (t-deg 1)))
 (p01  ((type boole) (t-deg 1)))
 (n    ((type nat)   (t-deg 1)))
 (n45^ ((type nat)   (t-deg 0)))
 ...)
```

to make the functions `variable-to-t-degree` and `variable-to-type` more efficient.

2.3.2 Program constants

Semantics

Program constants are interpreted as computable and in particular continuous functionals at a fixed type (in contrast to function symbols which are interpreted as not necessarily continuous objects). When introducing new program constants the user is therefore supposed to supply both type and totality-degree.

Syntax

Program constants are introduced by

`(add-program-constant '`*name type* `[`*t-degree*`] [`*string*`])` or `(apc ...)`, where

- *name* underlies the usual SCHEME-restrictions for constructing symbols,
- *type* is the type of the program-constant,
- *t-degree* is in $\{0, 1, 2\}$; if it is omitted, the totality degree 0 is taken by default;
- *string* is the string used for both displaying and parsing the program-constant. In order to avoid problems in the parsing routine, one should avoid the symbols `-,+,*,<,=` and use the underscore `_` for connecting subsymbols. In case *string* is omitted, the name of the symbol is used by default.

Program constants can be removed by

`(remove-program-constant '`*name*`)` abbreviated `(rpc '`*name*`)`.

Program-scheme-forms

MINLOG also knows a simple form of polymorphic program-constants in order to realise constants whose type is parametric in another one. The canonical example is the recursion operator for natural numbers $R_\rho^{\texttt{nat}}$, having type $\rho \to (\rho \to \rho) \to \texttt{nat} \to \rho$. Such constructs are implemented as *program-scheme-forms*. Their names usually end in `at` (e.g. `(nat-rec-at 'boole)`; see below).

Currently there are no routines for adding or removing program-scheme-forms, so the ambitious user has to consult the internals and the source code.

Accessing program constants

The functions `program-constant?` and `program-scheme-form?` test on being a program constant or program-scheme-form respectively. Several functions can be used to extract type, totality degree and display information for program constants: `program-constant-to-type`, `program-scheme-form-to-type`, `program-constant-to-t-deg`[5], `program-constant-to-string`, `program-scheme-form-to-string`.

Applications of program-constants are parsed and displayed in an uncurried form. For instance, after having introduced

```
(apc 'test (c-arrow 'nat 'nat 'boole))
```

the term `((test 0) 0)` is displayed `test 0 0` and parsed the same.

Furthermore MINLOG uses infix notation and organises parentheses correctly for some of the predefined program-constants.

Program-scheme-forms are currently displayed without type-parameter. E.g., the *nat*--recursion operator $R_\rho^{\texttt{nat}}$ prints as `nat-rec`. This prohibits parsing of program-scheme-forms and forces to use the internal representation, which comes as application of its symbol to a quoted type-expression.[6] For instance the recursion operator $R_{\texttt{boole}}^{\texttt{nat}}$ is internally represented by

```
(nat-rec-at 'boole)
```

[5]All program-scheme-forms are automatically supposed to have totality degree 1.

[6]As in axiom-scheme-forms the quotation is necessary to block evaluation throughout normalisation.

The routine `program-scheme-form-to-string` turns this expression simply into `nat-rec`, cutting off both `at` and the type-information.

Since the type of each program-scheme-form *name* depends on its type-parameter, there is a procedure (`type-of-`*name type*) to compute it.

```
(type-of-nat-rec-at 'boole) ⟹
(arrow boole (arrow (arrow nat (arrow boole boole)) (arrow nat boole)))
≡ boole→ (nat→boole→boole) →nat→boole.
```

Animating program constants

After having added a program constant symbol to the system, you are supposed to provide its interpretation by defining a SCHEME-procedure for it that operates on internal term representations. In doing so remind that program-constants usually come in curried form and therefore require in their definitions nested `lambda`-abstractions such as

```
(lambda (a)
   (lambda (b)
      (lambda (c) kernel)))
```

instead of SCHEME's

```
(lambda (a b c) kernel)
```

Self-evaluating program-constants. The easiest way a program constant can act is to do nothing. As an example we might want our above-defined `test`-program-constant to behave as follows:

```
(test (+ 0 1)) (+ 5 2) ⟹ (test 1) 7.
```

This is called *evaluating-to-itself* and may be achieved by adding the program-constant using `add-self-evaluating-program-constant` instead of `add-program-constant` (both get the same arguments). This function automatically defines a SCHEME-function with the same name as the program constant that simply takes arguments according to the type information and reproduces itself.

Internals

All program-constants are collected in an association list `PROGRAM-CONSTANTS` of the form

```
((nil nulltype 2 "nil")
 (undef_boole boole 0 "undef_boole")
 (true boole 2 "true")
 (false boole 2 "false")
 (=-strict-boole (arrow boole (arrow boole boole)) 1 "=")
 ...)
```

In terms, program-constants appear simply as symbols. Thus the program-constant application obtained from `(parse-term "0 = 1")` is `((=-strict-nat 0) 1)`. The special role of `-,=,*,+` throughout the parsing routine forbids their use in symbols to parse.

The internal representation of program-scheme-forms has already been discussed above. All program-scheme-form-symbols are collected in `PROGRAM-SCHEME-SYMBOLS`. Let us remark one trivial fact on their animation: The respective SCHEME-procedures have to expect a type as first argument, since, e.g., in evaluation of `(nat-rec-at 'boole)` the function `nat-rec-at` is supplied the quoted type-information first.

Generic program constants

For each basic type ι there exists

`undef_`ι of type ι and totality degree 0 (self-evaluating) representing the least element of the modelling domain; for higher types the least element is produced by the program-scheme-form `(undef-at` ρ`)`;

`=-strict-real` of type $\iota \to \iota \to$ `boole` for testing syntactic equality monotonously and totally (in contrast to the non-continuous function `eq_`ι explained below), displaying infix by `=`;

`if-`ι of type `boole`$\to \iota \to \iota \to \iota$ (total) for implementing case distinction; applications display quite unexpected (reducing parentheses):

`(display-term '(((if-nat true) 0) 1))` $\Longrightarrow$ `[if true then 0 else 1]`

For higher types ρ there is an analogous program-scheme-form named `(if-at` ρ`)`.

2.3.3 Function symbols

Semantics

It is exactly the function symbols that are interpreted as possibly non-continuous functionals living in $(D_\rho \to D_\sigma) \setminus D'_{\rho\to\sigma}$. The most important examples are non-strict equality `eq_`ι and the definedness-predicate `def_`ι for basic types ι:

```
(def-nat 'undef_nat) ==> false    (def-nat 0) ==> true
```

Repeating the warning remarks above we forbid λ-abstraction over variables under function symbols, because currying is not available. To stress this, function symbols are assigned a typed functionality of the form $\rho_1, \ldots, \rho_n \to \sigma$ $(n \geq 1)$ rather than a type.

Syntax

A *functionality* is constructed from a list `(`ρ_1 ρ_2 ... ρ_n σ`)` (for a function assigning arguments of types $\rho_1, \rho_2, \ldots, \rho_n$ an object of type σ) by

`(c-fcty` ρ_1 ρ_2 ... ρ_n σ`)`

One can add new function symbols using

`(add-function-symbol '`*name functionality t-degree string*`)` or `(afs ...)`

where *t-degree* is declared as for program constants and *string* is used for parsing and displaying.

Function symbols are removed by

`(remove-function-symbol '`*name*`)`.

Accessing function symbols

First of all a functionality can be decomposed into the list of argument types and the target type using the functions `fcty-to-val` and `fcty-to-args`. To display a functionality use `d-fcty`. `function-symbol?` tests on being a function-symbol whereas `function-symbol-to-fcty` and `function-symbol-to-t-deg` yield the expected information about function symbols.

Generic function symbols

For each basic type ι there exists

`eq_`ι of functionality $\iota, \iota \to$ *boole* for non-strict equality; it has totality degree 0 and displays as its symbol.

`def-`ι of functionality ι $\to$`boole` for testing on being undefined supertotally; it is defined in the obvious way and displays and parses without type-information: `def`.

Animating function-symbols

As program-constants each function-symbol has to be implemented by a SCHEME-function of the same name. To define it, one can use the function `define-by` with option `'uncurry` which implements the functionality correctly: A function-symbol of functionality $\rho_1, \rho_2, \rho_3 \to \sigma$ is a function with head of the form

```
(lambda (a b c) ...)
```

Internals

The global variable `FUNCTION-SYMBOLS` is a list which contains one entry for each function-symbol of the system. As expected, those have the form

(*fs-name functionality t-degree string*)

2.4 Composing terms

2.4.1 The term grammar

The abstract syntax for terms is given by the grammar

$r, s := x$ `|` c `| (lambda (`x`^) `r`) | (`r s`) | (`f $r_1 \ldots r_n$`)`
`| (cons `r s`) | (car `r`) | (cdr `r`),`

where application, function-symbol-application and projections have to be well-typed according to the following typing rules.

- Variables and constants are typed through declaration.
- If $x\hat{}$ has type ρ and r has type σ then `(lambda (`$x\hat{}$`) `r`)` has type $\rho \to \sigma$.
- If r has type $\rho \to \sigma$ and s has type ρ then `(`r s`)` has type σ.
- If r has type ρ and s has type σ then `(cons `r s`)` has type $\rho * \sigma$.
- If r has type $\rho * \sigma$ then `(car `r`)` has type ρ and `(cdr `r`)` has type σ.
- If $r_1, \ldots, r_n$ have types $\rho_1, \ldots, \rho_n$ respectively and f is of functionality $\rho_1, \ldots, \rho_n \to \sigma$ then `(`f r_1 `...` r_n`)` is of type σ.

In order to guarantee correctness of our semantics λ-abstraction underlies the additional restriction that the abstracted variable $x\hat{}$ does not occur under a function symbol.

NOTATION. We use r, s, t for terms in this manual. Throughout MINLOG application associates to the left. Hence

$$rs_1 \dots s_n \equiv (\dots((rs_1)s_2)\dots s_n).$$

In this manual we use the vector notation $\vec{s}$ to denote iterated $\to$- and $*$-eliminations. Thus $r\vec{s}$ could be for instance

$$(\text{car } (\text{cdr } rs_1))s_4.$$

2.4.2 Auxiliary construction functions

As obvious from the definition, terms are built up by ordinary SCHEME-pairing and abstraction operations. However, some auxiliary functions have been implemented to simplify term construction.

(c-app *op* arg_1 ... arg_n) yields a nested application according to the above notational convention:

(...(*op* arg_1)...arg_n).

In case *op* were a function symbol, it produces a function-symbol application instead.

(c-abst var_1 ... var_n r) constructs a nested abstraction:

(lambda (var_1) ... (lambda (var_n) r))

2.4.3 The type and t-degree of a term

Any MINLOG-term — although untyped in nature — can be typed by the typing rules given above. This is done by the function `term-to-type`:

```
(term-to-type (pt "[m,p][if p then m else 0]"))
   ==>   (arrow nat (arrow boole nat))
```

(`[m,p]` denotes λ-abstraction, cf. 2.4.5)

The *totality-degree* of a term can be derived from the totality degree of its atomic constituents:

```
(term-to-t-deg (pt "[m,p][if p then m else 0]")) ==> 1
```

In order to test if a term is total, i.e., has positive totality degree, one can also use the function `synt-total?`.

2.4.4 Free variables and substitution

The *free variables* of a term are gathered as a list by the function `term-to-free`. In this manual we will use the function fv instead.

A *substitution* is a SCHEME-list

((var_1 r_1) ... (var_n r_n)).

It will be written $[r_1, \dots, r_n/var_1, \dots, var_n]$ and denoted by σ throughout this manual. We write $r\sigma$ for the result of executing the simultaneous substitution σ on r as is done by the function `term-substitute` described below.

The empty substitution is contained in `empty-subst`. Adding the binding $x \mapsto r$ to σ results in

`(extend` σ x r`).`

A *generalised substitution* is a SCHEME-list

`((`r_1 s_1`) ... (`r_n s_n`)).`

MINLOG provides the following substitution functions.

`(term-substitute` r σ`)` executes standard simultaneous substitution, renaming bound variables wherever necessary.

`(term-subst` r x s`)` is the shortcut to the singular substitution $r[s/x]$.

`(term-gen-substitute` r σ`)` allows generalised substitutions σ.

`(term-gen-subst` r x s`)` abbreviates `(term-gen-substitute` r `'((`x s`)))`.

2.4.5 Parsing and displaying

Using the string information provided for all atomic constituents, MINLOG can both display term-representations in string form by `display-term` (`dt` for short, invokes `term-to-string`) and reparse them by `parse-term` (abbreviated `pt`). However, there are some limitations as to the parsability of certain program-constants, function-symbols and program-scheme-forms as described above.

- (Nested) λ-abstractions are denoted by writing the abstracted variable(s) in square brackets before the kernel:

```
(dt '(lambda (m^) (lambda (n^) m^))) ==> [m^,n^]m^
```

- As already noted, applications are displayed and parsed left-associatively. Thus program-constant-applications and function-symbol-applications display similarly (if not infix is used), although their internal representations are totally different:

```
(dt '((and_strict p) q))  ==>  and_strict p q
(dt '(eq_boole p q))      ==>  eq_boole p q
```

- Pairs are written with an infixed `#`-symbol. Projections use a prefixed `p_1` or `p_2`.

```
(dt '(car (cons 0 (cons 1 2))))  ==>  p_1(0#1#2)
```

2.4.6 Notations

In some chapters of this manual we will adopt the following abbreviations.

$$\begin{aligned} \lambda x t &:= \texttt{(lambda (}x\texttt{) }t\texttt{)}, \\ \langle r, s\rangle &:= \texttt{(cons }r\ s\texttt{)}, \\ r1 &:= \texttt{(car }r\texttt{)}, \\ r2 &:= \texttt{(cdr }r\texttt{)}. \end{aligned}$$

Furthermore we will use the vector notation $r\vec{r}$ to denote iterated eliminations, i.e., applications and projections.

2.5 Operational semantics: The reduction relation

By defining an evaluation mechanism for transforming terms into normal forms we give an operational semantics for our term language that is correct with respect to Scott-domain-semantics. For normal closed terms of ground type it is also complete in the sense that each object in the respective domain can be assigned a uniquely determined normal closed term.

The operational semantics is presented as a rewriting semantics. To this end we define the rewrite relation $\to$ as the union of $\to_{\beta\eta\uparrow}$ and $\to_R$, both to be defined below.

Definition. Given a binary relation $\mapsto$ between terms we define the *term closure* $\to$ inductively by

- If $r \mapsto r'$ then $r \to r'$.
- If $r \to r'$ then also

$$sr \to sr', \quad \texttt{(lambda (}x\texttt{) }r\texttt{)} \to \texttt{(lambda (}x\texttt{) }r'\texttt{)},$$

$$\texttt{(cons }r\ s\texttt{)} \to \texttt{(cons }r'\ s\texttt{)}, \quad \texttt{(cons }s\ r\texttt{)} \to \texttt{(cons }s\ r'\texttt{)}, \quad f(\vec{r}r\vec{r}')\vec{s} \to f(\vec{r}r'\vec{r}')\vec{s}.$$

2.5.1 λ-calculus: $\to_{\beta\eta\uparrow}$ *−reduction*

The relations $\to_\beta, \to_\eta$ and $\to_{\eta\uparrow}$ are given as the term closure of the following conversion relations.

$$\begin{aligned} \texttt{(lambda (}x\texttt{) }r\texttt{) }s\vec{s} &\mapsto_\beta r[s/x]\vec{s}, \\ \texttt{(car (cons }r\ s\texttt{))}\vec{s} &\mapsto_\beta r\vec{s}, \\ \texttt{(cdr (cons }r\ s\texttt{))}\vec{s} &\mapsto_\beta s\vec{s}, \\ \texttt{(lambda (}x\texttt{) (}r\ x\texttt{))} &\mapsto_\eta r, \quad \texttt{if } x \notin \texttt{fv}(r), \\ \texttt{(cons (car }r\texttt{) (cdr }r\texttt{))} &\mapsto_\eta r, \\ r &\mapsto_{\eta\uparrow} \texttt{(lambda (}x\texttt{) (}r\ x\texttt{))} \\ &\qquad \text{if } r : \rho \to \sigma \text{ is not a } \texttt{lambda}\text{-term and } x : \rho \text{ is new}, \\ r &\mapsto_{\eta\uparrow} \texttt{(cons (car }r\texttt{) (cdr }r\texttt{))} \\ &\qquad \text{if } r : \rho * \sigma \text{ is not a } \texttt{cons}\text{-term}. \end{aligned}$$

2.5.2 Defined constants

We assume further that we have a conversion relation $\mapsto_R$ which rewrites only terms of the form $c\vec{r}$ and $f(\vec{r})\vec{s}$ with $\vec{r}$ and $\vec{s}$ in $\to_{\beta\eta\uparrow R}$-normal-form to terms in $\to_{\eta\uparrow}$-normal-form. Let $\to_R$ be its term closure.

More requirements for $\to_R$ are listed in chapter 7, where normalization-by-evaluation is discussed.

2.5.3 Properties of normal forms

Without proof we state the following inductive characterizations of the set Terms, the set NF_γ of normal forms with respect to $\to_\gamma$ ($\gamma \in \{\beta, \eta\uparrow, \beta\eta\uparrow\}$) and the set LNF of $\to := \to_{\beta\eta\uparrow R}$-normal-forms.

$$\begin{array}{rcccl}
\text{Terms} & \ni & r,s & ::= & x\vec{r} \mid c\vec{r} \mid f(\vec{r})\vec{s} \mid \lambda xr \mid \langle r,s\rangle \mid (\lambda xr)s\vec{s} \mid \langle r,s\rangle i\vec{s}, \\
\mathrm{NF}_{\eta\uparrow} & \ni & r,s & ::= & (x\vec{r})^\iota \mid (c\vec{r})^\iota \mid (f(\vec{r})\vec{s})^\iota \mid \lambda xr \mid \langle r,s\rangle \mid ((\lambda xr)s\vec{s})^\iota \mid (\langle r,s\rangle i\vec{s})^\iota, \\
\mathrm{NF}_\beta & \ni & r,s & ::= & x\vec{r} \mid c\vec{r} \mid f(\vec{r})\vec{s} \mid \lambda xr \mid \langle r,s\rangle, \\
\mathrm{LNF} & \ni & r,s & ::= & (x\vec{r})^\iota \mid (c\vec{r})^{\iota\ (\ddagger)} \mid (f(\vec{r})\vec{s})^{\iota\ (\ddagger)} \mid \lambda xr \mid \langle r,s\rangle,
\end{array}$$

where ($\ddagger$) is the restriction that $c\vec{r}$ and $f(\vec{r})\vec{s}$ are not $\to_R$-reducible.

2.5.4 Front end

Basic assumptions about $\to$ and some facts about normalization-by-evaluation are formulated in 7. The user usually can assume that $\to$ yields unique normal forms which are described in detail in the next section.

To obtain normal forms of a term r use

`(term-to-eta-nf` r`)` for the $\to_\eta$-normal form (provided r is in LNF),

`(normalize-term` r`)` or `(nt` r`)` for its $\to_{\beta\eta R}$-normal form.

Chapter 3

Formulas

3.1 Atomic formulas

Any atomic formula is built from a boolean term r by forming `(c-atom` r`)`. At this point our basic assumption that we only deal with decidable atoms becomes visible.

NOTATION. We use $A, B, C, \dots$ for arbitrary and P, Q for atomic formulas in this manual.

3.1.1 Constructors

Some *standard construction-functions* have been implemented to construct atomic formulas with particular predicates.

`(c-def` r`)` builds the atomic formula `def` r for any term r of ground type.

`(c-=` r s`)`, `(c-<` r s`)`, `(c-<=` r s`)` construct the atomic formulas r`=`s, r `<` s, r `<=`s for strict equality and inequalities `<` and `<=` if r and s are of the same basic type and of type `nat` in case of `<` and `<=`.

`truth` and `falsity` yield the atomic formulas `T` and `F`.

3.1.2 Accessing atomic formulas

`(atom-form?` f`)` tests on being an atomic formula. The underlying boolean term of an atomic formula can be extracted by `atom-form-to-kernel`.

3.2 Composing formulas

Formulas are composed by implication, conjunction, universal and existential quantification. For all connectives there is a constructor version beginning with `cons-`*name* for one application of the connective and one iterated variant named `c-`*short-name*.

Implication. `(cons-implication` A B`)` generates the formula $A \to B$ which is displayed and parsed A`->`B. Like the arrow type constructor $\to$ associates to the right. This is reflected in the definition of the iterated implication constructor —

`(c-imp` A_1 A_2 $\dots A_n$ B`)`
`==>` $A_1 \to A_2 \dots \to A_n \to B$
$\equiv A_1 \to (A_2 \to \dots (A_n \to B) \dots)$.

`imp-form?` tests on being an implication. The destructor functions are called `premise` and `conclusion`, i.e.,

(premise (cons-implication A B)) ==> A,
(conclusion (cons-implication A B)) ==> B.

Negation. As usual in minimal and intuitionistic logic *negation* $\neg A$ is defined by $A \to$`F` which can be constructed using (cons-negation A). In order to negate a list of formulas use

(c-neg A_1 ... A_n) ==> A_1 -> ... -> A_n -> F.

Conjunction. (cons-conjunction A B) constructs the formula $A \wedge B$ which is parsed and displayed A&B. Conjunction associates to the left, hence

(c-and A_1 A_2 ... A_n) ==> A_1 & ... & $A_n \equiv$ (A_1 & A_2) & ... & A_n.

`and-form?` tests on conjunctions, the destructors are `left-conjunct` and `right-conjunct`.

Universal quantification. (cons-all-formula x A) constructs the formula $\forall x A$ which is parsed and displayed `all x` A. Iterated universal quantifications are produced by

(c-all x_1 ... x_n A) ==> all $x_1, \ldots, x_n A$

Both the parser and the display-routine support the point notation to reduce the need of parentheses: all $\vec{x}$.*expression* $:\equiv$ all $\vec{x}$(*expression*).

`all-form?` tests on universal quantification; the destructors are `all-form-to-symbol` and `all-form-to-kernel`.

Existential quantification. This underlies the same syntactic conventions as universal quantification. The respective constructor, test and destructor functions are `cons-ex-formula`, `c-ex`, `ex-form?`, `ex-form-to-symbol` and `ex-form-to-kernel`.

What about disjunction? In the presence of the ground types `nat` (or `boole`) the disjunction $A \vee B$ can be defined with the help of existential quantification.

$$A \vee B := \exists m.(m = 0 \to A) \ \& \ (m \neq 0 \to B).$$

It turns out that the natural-deduction derivation rules for $\vee$ can be derived from those of the other connectives even in intuitionistic logic.

3.3 Free and bound variables and substitution

- The set of **free variables** of a formula A is computed as a list by (formula-to-free A). In this manual we will be using the abbreviation $\mathrm{fv}(A)$ (as well as $\mathrm{fv}(r)$) for the list of free variables.
- (normalize-formula A) normalizes the terms in the atoms of A. It can be abbreviated `nf`.

- `(alpha-equal-formulas?` A B `'() '())` tests if A and B are identical modulo names of bound variables, regardless whether the binding is a quantification or a λ-abstraction.
- `(alpha-=-nfs?` A B`)` tests α-equality of the normal forms of A and B.
- `(formula-substitute` A σ`)` executes simultaneous substitution, renaming bound variables where necessary. For substitution of only one single variable x, use the function `(formula-subst` A x t`)`.
- Analogous to the function `term-gen-substitute` there exists also a function for substituting expressions, called `formula-gen-substitute`. For details cf. 2.4.4.

3.4 How to display, check and parse formulas

- `(display-formula` A`)` outputs a reparsable[1] representation of A using the function `formula-to-string`. `display-formula` is abbreviated `df`.

  Remark. Although atomic formulas are essentially boolean terms they are internally represented differently. Hence, it makes a difference whether a boolean term is parsed as a term or a formula:

  ```
  (pt "p")     ==>     p      while
  (pf "p")     ==>     (atom p).
  ```

- The (more or less) inverse function is `parse-formula`, abbreviated `pf`.
- `(formula-form?` *object*`)` uses the test functions for the various formula-connectives and `atom-form?` in order to find out if *object* is a formula.
- `(check-and-display-formula` A`)` tries to output a representation of A as a formula. Since this amounts to analysing the structure of A recursively, it also functions as a test on syntactical correctness. It can be abbreviated `cdf`.

3.5 Internals

Atoms r (with r a term of type `boole`) are internally represented as `(atom` r`)` and thus syntactically distinguished from the term r. The connectives are translated as follows

```
(pf "A -> B")      ==>     (imp A B)
(pf "A & B")       ==>     (and A B)
(pf "all x A")     ==>     (all x A)
(pf "ex x A")      ==>     (ex x A)
```

(Note that spaces around `&` are needed!)

All functions concerning formulas can be found in `formula.scm`.

[1]Some restrictions apply as to parsing certain program-scheme-forms and –constants as described above.

Chapter 4

Proofs

MINLOG's logical calculus is a first-order natural deduction system enhanced by axioms for arithmetic and further data structures.

A proof $\vec{u} : \vec{A} \vdash d : B$ in natural deduction is a derivation of one goal formula B from a list of assumptions $\vec{A}$ which are assigned labels $\vec{u}$. The rules used in the derivation can be linearly denoted by a proof term d. If the context is clear or not important we often denote $\Gamma \vdash d : B$ by d^B. Also we mark assumption labels by their formula instead of giving the context.

- The list `((`u_1 A_1`)` ...`(`u_n A_n`))` forms the *context* of the proof. By default, MINLOG allows u, v for assumption labels. In this manual we use Γ to denote contexts.

- Proof terms (denoted by d, e) are generated by the following grammar.

  $$d, e ::= a \mid u \mid g \mid \texttt{(lambda (}u\texttt{)}\ d\texttt{)} \mid de \mid \texttt{(cons}\ d\ e\texttt{)} \mid \texttt{(car}\ d\texttt{)} \mid \texttt{(cdr}\ d\texttt{)},$$

  where a denotes certain constants (and axiom-scheme-forms, see below) and g stands for global assumptions.

In MINLOG you can generate the proof $\Gamma \vdash d : A$ using

`(c-proof` Γ A d`)`.

Its components are regained by `proof-to-context`, `proof-to-formula` and `proof-to-pterm`. `proof-form?` tests on being a proof.

4.1 Constructing proofs

The following rules define when $\Gamma \vdash d : A$ is a derivation in our natural deduction calculus. We list both the graphical and the proof term representation of all inference rules of our natural deduction calculus.

- Every axiom and global assumption C gets assigned a representing constant which forms the proof $\emptyset \vdash c : C$. In case c is a declared global assumption the proof is constructed by `(global-assumption` c`)`. For the general treatment of axioms and global assumptions see sections 4.4 and 4.3.

- Under the assumption A, labeled u, we have a derivation of A: $u : A \vdash u : A$. The proof is constructed by `(make-assumption` u A`)`.

- $\to$-introduction: If under the assumption $u : A$ we have a proof term d for B then we can form a proof term `(lambda (`u`)` d`)` for $A \to B$ by cancelling the assumption u:

$$\dfrac{\begin{array}{c}[u : A] \\ | \\ B\end{array}}{A \to B}\, u \qquad\qquad \dfrac{\Gamma \vdash d : B}{\Gamma' \vdash \texttt{(lambda (}u\texttt{)}\ d\texttt{)} : A \to B}\ \Gamma' = \Gamma \setminus u : A.$$

The proof $\Gamma' \vdash$`(lambda (`u`)` d`)`$: A \to B$ is constructed by `(imp-intro` $\Gamma \vdash d : B\ \ u\ \ A$`)`.

- $\to$-elimination (modus ponens): If we have both a proof term d for $A \to B$ and a proof term e for A then we can form the proof term de proving B.

$$\dfrac{\begin{array}{c} | \\ A \to B\end{array} \qquad \begin{array}{c} | \\ A\end{array}}{B} \qquad\qquad \dfrac{\Gamma \vdash d : A \to B \qquad \Gamma' \vdash e : A}{\Gamma, \Gamma' \vdash de : B}$$

(Here Γ, Γ' denotes the concatenation of the two lists with duplicates removed.) The proof of $\Gamma, \Gamma' \vdash de : B$ is constructed by `(elim` $\Gamma \vdash d : A \to B\ \ \Gamma' \vdash e : A$`)`.

- $\forall$-introduction: If we have a proof term d for B where x does not occur free in any of the free assumptions $\vec{A}$ then `(lambda (`x`)` d`)` is a proof term for $\forall x B$.

$$\dfrac{\begin{array}{c} | \\ B\end{array}}{\forall x B} \qquad\qquad \dfrac{\vec{u} : \vec{A} \vdash d : B}{\vec{u} : \vec{A} \vdash \texttt{(lambda (}x\texttt{)}\ d\texttt{)} : \forall x B}$$

The proof $\Gamma \vdash$`(lambda (`x`)` d`)`$: \forall x B$ is constructed by `(all-intro` $\Gamma \vdash d : B\ \ x$`)`.

- $\forall$-elimination: If we have a proof term d for $\forall x B$ then we immediately obtain a proof term dt for $B_x[t]$ for any object term t of the same type as x:

$$\dfrac{\begin{array}{c} | \\ \forall x B\end{array} \qquad t}{B_x[t]} \qquad\qquad \dfrac{\Gamma \vdash d : \forall x B}{\Gamma \vdash dt : B_x[t]}$$

The proof $\Gamma \vdash dt : B_x[t]$ is constructed by `(elim` $\Gamma \vdash d : \forall x B\ \ t$`)`.

- &-introduction: Two proofs $\Gamma \vdash d : B$ and $\Gamma' \vdash e : C$ can be joined to a proof term `(cons` $d\ e$`)` of $B \& C$:

$$\dfrac{\begin{array}{c} | \\ B\end{array} \qquad \begin{array}{c} | \\ C\end{array}}{B \& C} \qquad\qquad \dfrac{\Gamma \vdash d : B \qquad \Gamma' \vdash e : C}{\Gamma, \Gamma' \vdash (\mathrm{cons}\ d\ e) : B \& C}$$

This is done by `(and-intro` $\Gamma \vdash d : B\ \ \Gamma' \vdash e : C$`)`.

- &-elimination: A proof term d for $B_1 \& B_2$ yields both a derivation `(car` d`)` of B_1 and `(cdr` d`)` of B_2.

$$\dfrac{\begin{array}{c} | \\ B_1 \& B_2\end{array}}{B_i} \qquad\qquad \dfrac{\Gamma \vdash d : B_1 \& B_2}{\Gamma \vdash (\mathrm{c}{}^{a}_{d}\mathrm{r}\ d) : B_{\substack{1 \\ 2}}}$$

`(and-elim-left` $\Gamma \vdash d : B \& C$`)` and `(and-elim-right` $\Gamma \vdash d : B \& C$`)` help to do this.

Note that — in contrast to most natural deduction systems — we do not have rules for ∃-introduction or –elimination, since they are implemented as axioms.[1]

The construction function `elim` may in fact be used for iterated eliminations also. Thus its general syntax is

(`elim` *proof* $arg_1 \ldots arg_n$)

where the arg_k are either proofs or object terms. Similarly there exists a general `intro`-function

(`intro` *proof* $arg_1 \ldots arg_n$),

but here arg_k is either a list (u A) (resulting in →-introduction) or an object variable x for ∀-introduction.

Finally there is a function (`close-proof` *proof*) that binds all free assumptions in the proof to close it.

4.2 Free variables and substitution in proof terms

Proof terms very much resemble object terms in that they also contain λ-abstraction, application, pairing, projection and even object terms as subterms. Therefore we speak of *t-expressions* to comprise both object terms and proof terms and denote them by r and s. The reason is that — with respect to variables and substitution — the two kinds can and must be treated simultaneously.

(`t-expr-to-free` r) yields the list of free variables of r. This may contain both object variables (from the object terms of ∀-eliminations) and assumption variables.

(`t-expr-substitute` r σ) executes the simultaneous substitution σ (which may also contain bindings for assumption variables now) on the term r.

(`t-expr-subst` r *var* s) substitutes s for the variable *var*.

(`t-expr-gen-substitute` r σ) applies the generalised substitution (again in the extended sense for t-expressions) on r.

(`t-expr-gen-subst` r s s') is the shortcut to the replacement of s' for s.

(`t-expr-replace` r σ) is a "mindless" version of `t-expr-substitute` that may provoke variable collisions, since it does not rename bound variables.

(`t-expr-repl` r *var* s) is the expected analogon for replacing *var* by s.

4.3 Global assumptions

4.3.1 Motivation

MINLOG supports a modular approach for proving theorems in that it allows to leave certain subproofs as global assumptions that play the role of lemmata. Often their use is unavoidable

[1]This rather strange implementation has its origin in the basic design of MINLOG as logical framework for **minimal logic**, which usually lives in the →&∀-fragment.

when one wants to maintain short proof terms after normalisation.[2] A global assumption need not have a MINLOG-proof itself although it is recommended that you assure yourself (and the reader of your proofs) of their provability, so as a rule you should include the MINLOG-proofs of all used global assumptions into your tactic file.

4.3.2 Administrating global assumptions

A global assumption can be added to MINLOG using

`(add-global-assumption` *symbol formula*`)` abbreviated `(aga ...)`.

It is removed by `(remove-global-assumption` *symbol*`)`, abbreviated `(rga ...)`.

4.3.3 Accessing global assumptions

`(global-assumption-variable?` *symbol*`)` tests if *symbol* is registered as the symbol for a global assumption. In that case `(global-assumption-variable-to-formula` *symbol*`)` returns its formula.

`(dga)` displays a list of all global assumptions.

4.3.4 Standard global assumptions

By default MINLOG contains the global assumptions

`efq` for *ex-falso-quodlibet*: $\forall p\hat{}.\mathtt{F} \to p\hat{}$ (this formula can be accessed as `efq-formula`).

`stab` for *stability*: $\forall p\hat{}.\neg\neg p\hat{} \to p\hat{}$ (this formula can be accessed as `stab-formula`).

(Recall that in $\mathtt{F} \to p\hat{}$ and in $\neg\neg p\hat{} \to p\hat{}$ the symbol $p\hat{}$ denotes an atom, cf. 3.4)

Notice that the two global assumptions provide the respective laws for atomic formulas only. A proof of *ex-falso-quodlibet* for arbitrary formulas A can be obtained by `(proof-of-efq-at` A`)` and `(pterm-of-efq-at` A`)`, respectively. As for stability, the general law $\neg\neg A \to A$ can be derived from `stab` only for $\exists$-free formulas, where the functions `(pterm-of-stab-at` A`)` and `(stab-at` A`)` do the work.

In the presence of boolean induction (which is one of our standard axiom scheme forms, see below) both *ex-falso-quodlibet* and *stability* can be derived. The respective proofs and proof terms are saved in `efq-pterm`, `efq-proof`, `stab-pterm` and `stab-proof`. There is also a version of `(stab-at` A`)` where all stability assumptions have been replaced by their proofs which is called `(stab-arith-at` A`)`.

4.3.5 Internals

All global assumptions are gathered in the association list `GLOBAL-ASSUMPTIONS` which in virgin MINLOG reads

```
((efq (all p^ (imp (atom false) (atom p^))))
 (stab (all p^ (imp (imp (imp (atom p^) (atom false))
                        (atom false)) (atom p^))))).
```

[2]Since MINLOG can only work with normalised proofs in program extraction and display it is of utmost importance to pack subproofs into global assumptions wherever possible.

4.4 Axioms and axiom-scheme-forms

To administrate axiom systems, MINLOG allows the introduction of axioms and so-called axiom-scheme-forms which are axioms parametric in one or more formulas.

4.4.1 Axioms

Axioms are treated more or less as global assumptions. A new axiom can be added by `(add-axiom-constant` *symbol formula*`)` (`aac` for short, *formula* must be closed) and removed by `(remove-axiom-constant` *symbol*`)` (`rac` for short). The function `axiom-constant?` tests whether its argument is an axiom-constant in which case `axiom-constant-to-formula` produces the formula.

4.4.2 Axiom-scheme-forms

typically serve to implement induction-axioms and inference rules turned into axioms. Their parametricity is expressed in their symbols which usually end in `-at`. The most prominent example is the axiom-scheme-form `(nat-ind-at '`A`)`[3] stating induction as axiom.

A new axiom-scheme-form can be added by `(add-axiom-scheme-form` *symbol*`)`, although this certainly does not suffice. As MINLOG does for all its standard axiom-scheme-forms the user also has to provide for

- a SCHEME-procedure named `formula-of-`*symbol* which takes as many arguments as the axiom-scheme-form itself is supposed to and returns the axiom formula. Remark here that the formula of an axiom-scheme-form should always be closed and therefore all free variables have to be quantified in `formula-of-`*symbol*.

- a SCHEME-procedure named *symbol*`-axiom-at` that constructs the proof of the axiom parametric in the formula.

- a SCHEME-procedure named *symbol* that animates the axiom-scheme-form to enable normalisation of proofs (see below). This is the tricky part in adding new axiom-scheme-forms and the reason why only advanced MINLOGicians should dare to.

- Finally, when adding an axiom-scheme-form (or an axiom) for a formula which is non-harrop, the user should also provide for an extracted program.

`axiom-scheme-form?` tests whether its argument is an axiom-scheme-form, i.e., of the form `(`*axiom-scheme-symbol* `'`$formula_1$ `'`$formula_2$ `...)` in which case `axiom-scheme-form-to-op` yields *axiom-scheme-symbol* and `axiom-scheme-form-to-args` yields the list of the formulas (which in most cases contains only one element). `axiom-scheme-form-to-formula` computes the axiom by calling the function `formula-of-`*axiom-scheme-symbol*.

4.4.3 MINLOG's standard axioms

- The *axiom of truth* simply states that `T` is actually true. Its axiom-constant-symbol is `truth-axiom-symbol`; its proof is contained in `truth-axiom`.

- Boolean induction

[3] To allow normalisation-by-evaluation the formula arguments of axiom-scheme-forms need always be quoted.

$\forall \mathrm{fv}(A).A[\mathtt{true}/p] \to A[\mathtt{false}/p] \to \forall p A$

and boolean induction for partial variables

$\forall \mathrm{fv}(A).A[\mathtt{undef\_boole}/\hat{p}] \to A[\mathtt{true}/\hat{p}] \to A[\mathtt{false}/\hat{p}] \to \forall \hat{p} A$

are implemented by `(boole-ind-at` $\forall pA$`)` and `(boole^-ind-at` $\forall \hat{p}A$`)`.

- Induction on (partial) natural numbers

  $\forall \mathrm{fv}(A).A[0/n] \to (\forall n.A \to A[n+1/n]) \to \forall n A$ and
  $\forall \mathrm{fv}(A).A[\mathtt{undef\_nat}/\hat{n}] \to A[0/\hat{n}] \to (\forall \hat{n}.A \to A[\hat{n}+1/\hat{n}]) \to \forall \hat{n} A$

  is implemented as `(nat-ind-at` $\forall nA$`)` and `(nat^-ind-at` $\forall \hat{n}A$`)`.

- Proof by cases comes in three flavours: For total variables, for partial variables and independent formulas and for partial variables with possibly dependent formulas.

  $\forall \mathrm{fv}(A), p.(p \to A[\mathtt{true}/p]) \to (\neg p \to A[\mathtt{false}/p]) \to A$,
  $\forall \mathrm{fv}(A), \hat{p}.(\hat{p} \to A) \to (\neg \hat{p} \to A) \to A$ and
  $\forall \mathrm{fv}(A), \hat{p}.(\neg \mathtt{def}(\hat{p}) \to A[\mathtt{undef\_boole}/\hat{p}]) \to (\hat{p} \to A[\mathtt{true}/\hat{p}]) \to (\neg \hat{p} \to A[\mathtt{false}/\hat{p}]) \to A$.

  The respective axiom-scheme-forms are `(cases-at` $\forall pA$`)` and `(cases^-at` $\forall \hat{p}A$`)` and `(cases-with-undef-at` $\forall \hat{p}A$`)`.

- To relate quantification over total and partial variables there exists an axiom-scheme-form `(all-all^-at` $\forall \hat{x}^\iota A$`)` proving

  $$\forall \mathrm{fv}(A).A[\mathtt{undef\_}\iota/\hat{x}] \to \forall x A[x/\hat{x}] \to \forall \hat{x} A.$$

- The role of the existential quantifier is axiomatised by the rules of
  $\exists$-introduction: `(ex-intro-at` $\exists xA$`)` proves $\forall \mathrm{fv}(A), x.A \to \exists x A$.
  $\exists$-elimination: `(ex-elim-at` $\exists xA$ B`)` proves $\forall \mathrm{fv}(A).\exists x A \to (\forall x.A \to B) \to B$, if x does not occur free in B.

4.4.4 Internals

Axiom-constants and the symbols of axiom-scheme-forms are stored in the global SCHEME-variables `AXIOM-CONSTANTS` and `AXIOM-SCHEME-SYMBOLS`.

4.5 Normalisation

Proofs often contain detours that essentially stem from a modular approach in the construction process. For instance we might prove B by using a given derivation $A \to B$ and showing A. However, if the former derivation itself was built by $\to$ introduction from a proof of B under the assumption A we surely constructed a detour:

$$\dfrac{\dfrac{\begin{matrix}[u\colon A]\\ \vert\\ B\end{matrix}}{A \to B}\,u \qquad \begin{matrix}\vert\\ A\end{matrix}}{B}$$

We can eliminate it by filling in the proof of A at every place we used $u\colon A$ in the proof of B on the left. Let us look at proof terms now: There the situation reads

$$\dfrac{\dfrac{\Gamma, u\colon A \vdash d\colon B}{\Gamma \vdash \texttt{(lambda (}u\texttt{)}\ d\texttt{)}\colon A \to B} \qquad \Gamma' \vdash e\colon A}{\Gamma, \Gamma' \vdash \texttt{(lambda (}u\texttt{)}\ d\texttt{)}e\colon B}$$

which gets simplified to $\Gamma, \Gamma' \vdash d[e/u]\colon B$.

There are similar so-called β-conversions for the other connectives:

$$\begin{aligned} \texttt{(lambda (}x\texttt{)}\ d\texttt{)}t &\to d[t/x], \\ \texttt{(car (cons}\ d\ e\texttt{))} &\to d, \\ \texttt{(cdr (cons}\ d\ e\texttt{))} &\to e. \end{aligned}$$

Another form of detours — η-redexes — arise from applying an introduction-rule directly after elimination:

$$\dfrac{\dfrac{\begin{array}{c} \vdots \\ A \to B \end{array} \qquad [u\colon A]}{B}}{A \to B}\, u$$

The respective η-conversions are

$$\begin{aligned} \texttt{(lambda (}u\texttt{) (}d\ u\texttt{))} &\to d && \text{if } u \text{ is not free in } d, \\ \texttt{(cons (car}\ d\texttt{) (cdr}\ d\texttt{))} &\to d, \\ \texttt{(lambda (}x\texttt{) (}d\ x\texttt{))} &\to d && \text{if } x \text{ is not free in } d. \end{aligned}$$

Furthermore we have conversions for each of the axiom-scheme-forms (for simplicity without parameters).

$$\begin{aligned}
\texttt{(boole-ind-at}\ \forall pA\texttt{)}d^{A[\texttt{true}]}e^{A[\texttt{false}]}\,\mathrm{true} &\to d, \\
\texttt{(boole-ind-at}\ \forall pA\texttt{)}d^{A[\texttt{true}]}e^{A[\texttt{false}]}\,\mathrm{false} &\to e, \\
\texttt{(boole\^{}-ind-at}\ \forall \hat{p}A\texttt{)}d_0^{A[\texttt{undef_boole}]}d_1^{A[\texttt{true}]}d_2^{A[\texttt{false}]}\texttt{undef_boole} &\to d_0, \\
\texttt{(boole\^{}-ind-at}\ \forall \hat{p}A\texttt{)}d_0^{A[\texttt{undef_boole}]}d_1^{A[\texttt{true}]}d_2^{A[\texttt{false}]}\texttt{true} &\to d_1, \\
\texttt{(boole\^{}-ind-at}\ \forall \hat{p}A\texttt{)}d_0^{A[\texttt{undef_boole}]}d_1^{A[\texttt{true}]}d_2^{A[\texttt{false}]}\texttt{false} &\to d_2, \\
\texttt{(nat-ind-at}\ \forall nA\texttt{)}d^{A[0]}e^{\forall n.A \to A[n+1]}0 &\to d, \\
\texttt{(nat-ind-at}\ \forall nA\texttt{)}d^{A[0]}e^{\forall n.A \to A[n+1]}(t+1) &\to et[\texttt{(nat-ind-at}\ \forall nA\texttt{)}det], \\
\texttt{(nat\^{}-ind-at}\ \forall \hat{n}A\texttt{)}d_0^{A[\texttt{undef_nat}]}d_1^{A[0]}d_2^{\forall \hat{n}.A \to A[\hat{n}+1]}\texttt{undef_nat} &\to d_0, \\
\texttt{(nat\^{}-ind-at}\ \forall \hat{n}A\texttt{)}d_0^{A[\texttt{undef_nat}]}d_1^{A[0]}d_2^{\forall \hat{n}.A \to A[\hat{n}+1]}0 &\to d_1, \\
\texttt{(nat\^{}-ind-at}\ \forall \hat{n}A\texttt{)}d_0^{A[\texttt{undef_nat}]}d_1^{A[0]}d_2^{\forall \hat{n}.A \to A[\hat{n}+1]}(t+1) &\to d_2t[\texttt{(nat\^{}-ind-at}\ \forall \hat{n}A\texttt{)}d_0d_1d_2t], \\
\texttt{(cases-at}\ \forall pA\texttt{)true}d^{\mathbf{T} \to A[\texttt{true}]}e^{\mathbf{F} \to A[\texttt{false}]} &\to d\texttt{truth-axiom-symbol}, \\
\texttt{(cases-at}\ \forall pA\texttt{)false}d^{\mathbf{T} \to A[\texttt{true}]}e^{\mathbf{F} \to A[\texttt{false}]} &\to e\texttt{(lambda (}u\texttt{)}\ u\texttt{)}, \\
\texttt{(cases\^{}-at}\ \forall \hat{p}A\texttt{)true}d^{\mathbf{T} \to A}e^{\mathbf{F} \to A} &\to d\texttt{truth-axiom-symbol}, \\
\texttt{(cases\^{}-at}\ \forall \hat{p}A\texttt{)}\begin{array}{c}\texttt{undef_boole} \\ \texttt{false}\end{array} d^{\mathbf{T} \to A}e^{\mathbf{F} \to A} &\to e\texttt{(lambda (}u\texttt{)}\ u\texttt{)}, \\
\texttt{(cases-with-undef-at}\ \forall \hat{p}A\texttt{)undef_boole}d_0d_1d_2 &\to d_0\texttt{(lambda (}u\texttt{)}\ u\texttt{)}, \\
\texttt{(cases-with-undef-at}\ \forall \hat{p}A\texttt{)true}d_0d_1d_2 &\to d_1\texttt{truth-axiom-symbol}, \\
\texttt{(cases-with-undef-at}\ \forall \hat{p}A\texttt{)false}d_0d_1d_2 &\to d_2\texttt{(lambda (}u\texttt{)}\ u\texttt{)}, \\
\texttt{(all-all\^{}-at}\ \forall \hat{x}A\texttt{)}d_0^{A[\texttt{undef}]}d_1^{\forall x\, A[x]}\texttt{undef} &\to d_0, \\
\texttt{(all-all\^{}-at}\ \forall \hat{x}A\texttt{)}d_0^{A[\texttt{undef}]}d_1^{\forall x\, A[x]}t &\to d_1t, \qquad \text{if } t \text{ is defined}, \\
\texttt{(ex-elim-at}\ \exists xA\ B\texttt{)[(ex-intro-at}\ \exists xA\texttt{)}'td^A\texttt{]}e^{\forall x.A \to B} &\to etd.
\end{aligned}$$

[...]

(normalize-proof *proof*) computes the normal-form of *proof* and may be abbreviated (np *proof*).

4.6 Accessing proofs

The access to properties of proofs is severely restricted by the form of representation: Since we do not register the formulas of assumptions that bound variables stand for, we cannot reconstruct subproofs of non-normal proofs. Let us illustrate this: The (non-normal) derivation

$$\dfrac{\dfrac{u1\colon q}{(A \to A) \to q}\,u2 \qquad \dfrac{[u3\colon A]}{A \to A}\,u3}{q}$$

is internally represented by

```
#(((u1 (atom q)))
  (atom q)
  ((lambda (u2) u1) (lambda (u3) u3))).
```

Now despite its simplicity we cannot extract the subproof

$$\dfrac{[u3\colon A]}{A \to A}\,u3$$

from the whole proof represented since we cannot infer the formula it derives. Due to these restrictions most of the decomposition functions often cannot work correctly for non-normal proofs.

4.6.1 Decomposing proofs

(elim-proof-to-main *proof*) extracts the main derivation of an elimination proof.

(elim-proof-to-side *proof*) yields the side derivation of a $\to$-elimination proof or the term argument of a $\forall$-elimination proof.

(intro-proof-to-main *proof*) produces the main part of a $\forall$- or $\to$-introduction proof.

(and-intro-proof-to-left *proof*) and (and-intro-proof-to-right *proof*) project the respective subproofs.

4.6.2 Check and display of proofs

(display-proof *proof*) produces a readable screen-display of normal proofs and may be abbreviated (dp *proof*).

(check-and-display-proof *proof*) ((cdp *proof*) for short) involves some additional testing operations, such as check-and-display-context, check-and-display-t-expr and check-and-display-formula.

(check-and-display-context Γ) tests if all assumption variables in Γ have been declared and if all formulas are correct.

(check-and-display-t-texpr r) is the respective test functions for both object and proof terms.

Chapter 5

Interactive theorem proving

Interactive theorem proving consists in studying a list of goals (the *goal-stack*) in a given context (made up from object variables and hypotheses) and suggesting to the machine how the proof of them should proceed. The first goal in the goal-stack is the *current goal*.

5.1 Administrating goals

For MINLOG a goal $\Delta \vdash S\colon B$ consists of

- the formula B to be shown;
- an *ordered context* Δ consisting of a list of free object variables and assumptions (i.e., 2-element-lists (u A) for the assumption A with label u) describing the environment under which the formula B is to be proven;
- a symbol S serving to identify the goal uniquely. Goal symbols usually start with `?` and get extended by MINLOG through each proof step.

Given these ingredients (`c-goal` S Δ B) produces the goal; the functions `goal-to-formula`, `goal-to-ordered-context` and `goal-to-symbol` serve to regain its constituents.

(`display-goal` $\Delta \vdash S\colon B$)

outputs a readable version of its argument. It may be abbreviated to `(dg ...)`.

5.2 Standard proof commands

5.2.1 `set-goal`

An interactive proof is started by defining the goal formula and naming a symbol for it. This is done by

(`set-goal` '*goal-symbol goal-formula*)

required that *goal-symbol* is sufficiently new to the system. If *goal-formula* is not closed, MINLOG automatically uses the universal closure.

5.2.2 display-current-goal

`(display-current-goal)` repeats the output of the current goal that occurs after each interactive proof step. It can be abbreviated to `(dcg)`.

5.2.3 normalize-goal

`(normalize-goal)` applies formula normalization (See sections 3.3 and 2.5!) to both the context of the current goal and the goal formula itself. It may be abbreviated to `(ng)`. Note that only the object terms in the atoms of the formulas are normalized. Hence e.g. $A \rightarrow \mathtt{T}$ is *not* "normalized" to `T`.

5.2.4 assume

`(assume` $symbol_1$ ... $symbol_n$`)` allows to eliminate ∀-quantifiers and implications in the current goal formula by introducing new object variables and assumptions labels, $symbol_k$ into the ordered context.

5.2.5 strip

If an automatic device for naming the variables and hypotheses suffices then one may use `(strip)` instead of `assume` to unwrap all leading ∀-quantifiers and implications of the goal formula. `(strip` n`)` only moves the first n assumptions/∀-quantifiers into the ordered context. `(strip)` is typically used, when the names are not addressed in the remainder of the interaction, e.g. if the current goal can be proven by a following `(prop)`.

5.2.6 unstrip

`(unstrip)` removes the ordered context of the current goal and extends the goal formula accordingly. `(unstrip` n`)` does the same for the first n variables/hypotheses of the ordered context.

5.2.7 drop

`(drop` u_1 ... u_n`)` drops the assumptions $u_1, \ldots, u_n$ from the ordered context (without changing the goal formula). It helps keeping clearity. Object variables are *not* dropped.

5.2.8 split

`(split)` expects a conjunction as the current goal formula and produces two new goals with the respective subformulas. `(split-iterated)` applies this iteratively, `(split-iterated` n`)` does it n times.

5.2.9 split-hyp

`(split-hyp` u`)` splits the assumption with label u of the ordered-context producing two new assumptions $u1$ and $u2$.

5.2.10 `use`

(`use` *obj*) tries to prove the goal "using" *obj*. Here *obj* can be one of the following.

- An assumption variable from the ordered context, or
- a global assumption, or
- a closed proof.

It is required that the formula associated with *obj* coincides with the goal formula.

5.2.11 `use-with`

(`use-with` *obj* arg_1 ... arg_n) tries to prove the goal "using" *obj* applied to arg_1 ... arg_n. Here *obj* can be one of the following.

- An assumption variable from the ordered context, or
- a global assumption, or
- a closed proof.

Each argument arg_k can be

- an assumption variable from the ordered context, or
- a global assumption, or
- a closed proof, or
- a new goal symbol (in this case a new goal with the same ordered context is generated), or
- one of the symbols `left` or `right`, or
- a term with free variables among the given ordered context.

It is required that the formula associated with *obj* after being stripped according to the arguments coincides with the goal formula.

5.2.12 `cut`

(`cut` A) generates for the given goal formula B the two new goals A and $A \to B$ with the same ordered context.

5.2.13 `inst`

(`inst-with` *obj* arg_1 ... arg_n) works analogously to `use-with` except that it does not try to prove the goal formula but adds the instantiated formula to the ordered context.

5.2.14 `prop`

`(prop)` searches for a proof of the current goal in minimal propositional logic. In particular it provides easy access to the axiom of truth for proving `T` and to *ex-falso-quodlibet* and *proof-by-contradiction*. The search mechanism is based on work of Hudelmaier [10, 11] and Dyckhoff [7]. If the search does not succeed, the same is done for intuitionistic propositional logic (by adding ex-falso-quodlibet assumptions to the context). If it does not succeed again, it does the same for classical propositional logic (by adding stability assumptions to the context).

5.2.15 `search`

This command activates an automatic proof search in the style of higher order logic programming.

`(search` m `'(`u_1 m_1`) (`u_2 m_2`) ...)` expects for m any natural number ≥ 1 and for u_k either

- assumption variables from the ordered context, or else
- symbols for global assumptions;

the corresponding m_k should again be a natural number ≥ 1. m, m_k denotes the multiplicities of use of the respective assumptions. Here m is the general bound for the multiplicity of uses of hypotheses in the ordered context. It is overridden by explicit mention of some (u_i m_i). `search` without any argument takes for m a default value, presently 2. A search for a proof of the current goal is carried out in minimal quantifier logic, where the hypotheses (and if mentioned explicitely some global assumptions) can only be used the given number of times. (A predecessor called `auto` is still available. However, `auto` only recognizes $\to\forall$-formulas.)

5.3 Using the axiom-scheme-forms

5.3.1 `ind`

`(ind)` initiates an inductive proof of the current goal, which must be a universally quantified formula, e.g., $\forall n A$. It uses the type of the quantified variable to launch induction and produces new goals for the respective proof steps accordingly. In case of a partial variable the axiom for induction over partial variables is used.

REMARK. `(ind)` simply uses the display-string of the type of the universally quantified variable, appends the string `ind` or `ind^` and tries to call the resulting function name. Thus `(ind)` cannot work correctly if no such function exists, as is the case for non-ground-type variables or new ground types for which no induction axiom-scheme has been declared. By default, MINLOGonly knows the functions `(nat-ind)`, `(nat-ind^)`, `(boole-ind)` and `(boole-ind^)`.

5.3.2 `cases`

`(cases` *test*`)` replaces the current goal A by two or — if *test* is not syntactically total and actually appears in A — by three new goals:

- If *test* is syntactically total:

$$test \to A[\texttt{true}/test],$$
$$\neg test \to A[\texttt{false}/test].$$

- If *test* does not occur in A:

$$test \to A,$$
$$\neg test \to A.$$

- If *test* is not syntactically total and occurs in A:

$$\neg(\texttt{def}\ \ test) \to A[\texttt{undef\_boole}/test],$$
$$test \to A[\texttt{true}/test],$$
$$\neg test \to A[\texttt{false}/test].$$

`(cases)` without an argument expects an atomic goal and searches for an if–term within the boolean term making up that goal. With the test *test* of the first if–term found it then calls `(cases` *test*`)`.

5.3.3 `cases-all^`

`(cases-all^)` expects a (not necessarily closed) goal $\forall\hat{x}A(\hat{x})$ with $\hat{x}$ of a ground type. It generates the two new goals $A[\texttt{undef}/\hat{x}]$ and $\forall x A(x)$, using the `all-all^`-axiom.

5.3.4 `ex-intro`

`(ex-intro` r`)` expects an existential goal $\exists x A$ or $\exists\hat{x}A$ and generates the new goal $A[r/x]$ or $A[r/\hat{x}]$. In the first case $\exists x A$ the term r needs to be total.

5.3.5 `ex-elim`

`(ex-elim` *obj*`)` can be applied to any goal A, provided *obj* is associated with an existential formula $\exists\hat{x}B$ ($\hat{x}$ may be partial here) as described below, such that $\hat{x}$ is not free in A. In any case a new goal $\forall\hat{x}.B \to A$ is generated. *obj* must be one of the following.

- An assumption variable for an existential formula from the ordered context of the goal, or
- a global assumption of an existential formula, or
- a closed proof of an existential formula, or
- an existential formula; in this case an additional new goal with this existential formula and the same ordered context is generated, or
- a list consisting of a new goal symbol and an existential formula; in this case again an additional new goal with this existential formula and the same ordered context is generated.

5.4 Rearranging the partial proof

While having a theorem proved interactively, MINLOG constructs a partial proof named `pproof` internally. A *partial proof* is essentially a proof containing free assumption variables with some information how the user intends to remove them. More precisely, it consists of

- a list of goals,
- the closed formula C the user wanted to prove originally (i.e., the original goal formula),
- a proof term d containing free assumption variables only among the goal formulas.

Given a list of goals l, a closed formula C and a proof term d as described, one obtains the related partial proof by

`(c-pproof` l C d`)`

The functions `pproof-to-goals`, `pproof-to-formula` and `pproof-to-pterm` extract the respective components of a partial proof.

5.4.1 `undo`

Each step in the construction of `pproof` is saved in the `pproof-history` in order to allow reversing erroneous proof steps:

`(undo)`	takes back the last proof step,
`(undo` n`)`	takes back the last n steps.

5.4.2 `get`

`(get` *goal-symbol*`)`

moves the goal labelled *goal-symbol* on top of the goal-stack and thus makes it the current goal.

5.4.3 `pproof`

Once having completed an interactive proof the partial proof saved in `pproof` is in fact a complete Minlog-proof. Thus you can normalise, display and output it as well as extract a subproof or program from it.

5.5 Internals

Goals and proofs are internally stored in vectors. The partial proof history is a regular Scheme-list `pproof-history` containing pproofs at each position.

Chapter 6

Extracting programs from proofs

6.1 Modified realisability

The Curry-Howard-Isomorphism shows how intuitionistic proofs in natural deduction style can be understood as programs of a typed lambda-calculus and vice versa. However, a program from a proof carries along a lot of unnecessary information. If, e.g., we have shown $\forall m \exists n . m = \sqrt{n}$ then the proof provides a function which assigns to any m an n with the desired property and furthermore, a proof that this n in fact fulfills it.

Realisability interpretations allow to extract exactly the computational content of a proof by reinterpreting formulas and transforming proofs accordingly. *Modified realisability* chooses the language of type theory to do so.

6.1.1. Definition. The *realiser type* $\tau(A)$ of a formula is defined by induction on the complexity of A.

$$
\begin{array}{rcl}
\tau(P) & := & \texttt{nulltype}, \\
\tau(\exists x^\rho A) & := & \begin{cases} \rho & \text{if } \tau(A) = \texttt{nulltype}, \\ \rho * \tau(A) & \text{otherwise.} \end{cases} \\
\tau(\forall x^\rho A) & := & \begin{cases} \texttt{nulltype} & \text{if } \tau(A) = \texttt{nulltype}, \\ \rho \to \tau(A) & \text{otherwise.} \end{cases} \\
\tau(A \& B) & := & \begin{cases} \texttt{nulltype} & \text{if both } \tau(A) \text{ and } \tau(B) = \texttt{nulltype}, \\ \tau(B) & \text{if just } \tau(A) = \texttt{nulltype}, \\ \tau(A) & \text{if just } \tau(B) = \texttt{nulltype}, \\ \tau(A) * \tau(B) & \text{otherwise.} \end{cases} \\
\tau(A \to B) & := & \begin{cases} \tau(B) & \text{if } \tau(A) = \texttt{nulltype}, \\ \texttt{nulltype} & \text{if } \tau(B) = \texttt{nulltype}, \\ \tau(A) \to \tau(B) & \text{otherwise.} \end{cases}
\end{array}
$$

A formula A is a *Harrop formula*, if $\tau(A) = \texttt{nulltype}$. Note that negative formulas are Harrop formulas. Non-Harrop formulas are called *computationally meaningful*.

6.1.2. DEFINITION. The *extracted term* of a proof $\vec{u}\colon \vec{A} \vdash d\colon B$ is defined relative to an assignment $\vec{x}_{\vec{u}}\colon \tau(\vec{A})$ of object variables for the assumptions:[1]

$$
\begin{array}{rcl}
\mathrm{et}_{\vec{x}_{\vec{u}}}(d^A) & := & \mathtt{nil} \quad \text{if } \tau(A) = \mathtt{nulltype}, \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(u) & := & x_u, \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\lambda u^A d) & := & \begin{cases} \mathrm{et}_{\vec{x}_{\vec{u}}}(d) & \text{if } \tau(A) = \mathtt{nulltype}, \\ \lambda x_u^{\tau(A)} \mathrm{et}_{\vec{x}_{\vec{u}}, x_u}(d) & \text{otherwise.} \end{cases} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(d^{A \to B} e) & := & \begin{cases} \mathrm{et}_{\vec{x}_{\vec{u}}}(d) & \text{if } \tau(A) = \mathtt{nulltype}, \\ \mathrm{et}_{\vec{x}_{\vec{u}}}(d)\mathrm{et}_{\vec{x}_{\vec{u}}}(e) & \text{otherwise.} \end{cases} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{cons}\ d^A\ e^B) & := & \begin{cases} \mathrm{et}_{\vec{x}_{\vec{u}}}(d) & \text{if } \tau(B) = \mathtt{nulltype}, \\ \mathrm{et}_{\vec{x}_{\vec{u}}}(e) & \text{if } \tau(A) = \mathtt{nulltype}, \\ (\mathtt{cons}\ \mathrm{et}_{\vec{x}_{\vec{u}}}(d)\ \mathrm{et}_{\vec{x}_{\vec{u}}}(e)) & \text{otherwise.} \end{cases} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{car}\ d^{A \& B}) & := & \begin{cases} \mathrm{et}_{\vec{x}_{\vec{u}}}(d) & \text{if } \tau(B) = \mathtt{nulltype} \\ (\mathtt{car}\ \mathrm{et}_{\vec{x}_{\vec{u}}}(d)) & \text{otherwise.} \end{cases} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{cdr}\ d^{A \& B}) & := & \begin{cases} \mathrm{et}_{\vec{x}_{\vec{u}}}(d) & \text{if } \tau(A) = \mathtt{nulltype} \\ (\mathtt{cdr}\ \mathrm{et}_{\vec{x}_{\vec{u}}}(d)) & \text{otherwise.} \end{cases} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\lambda x^\rho d) & := & \lambda x^\rho \mathrm{et}_{\vec{x}_{\vec{u}}}(d), \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(dt) & := & \mathrm{et}_{\vec{x}_{\vec{u}}}(d)t, \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{ex\text{-}intro\text{-}at}\ \exists x A) & := & \lambda \mathrm{fv}(A), x. \begin{cases} x & \text{if } \tau(A) = \mathtt{nulltype}, \\ \lambda y.(\mathtt{cons}\ x\ y) & \text{otherwise.} \end{cases} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{ex\text{-}elim\text{-}at}\ \exists x^\rho A\ B) & := & \\
\multicolumn{3}{l}{\lambda \mathrm{fv}(\exists x A, B) \begin{cases} \lambda y^\rho, z^{\rho \to \tau(B)}.zy & \text{if } \tau(A) = \mathtt{nulltype}, \\ \lambda y^{\tau(\exists x A)}, z^{\tau(\forall x^\rho.A \to B)}.z(\mathtt{car}\ y)(\mathtt{cdr}\ y) & \text{otherwise.} \end{cases}} \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{nat\text{-}ind\text{-}at}\ \forall n A) & := & \lambda \mathrm{fv}(\forall n A).(\mathtt{nat\text{-}rec\text{-}at}\ \tau(A)), \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{boole\text{-}ind\text{-}at}\ \forall p A) & := & \lambda \mathrm{fv}(\forall p A), x^{\tau(A)}, y^{\tau(A)}, p.(\mathtt{if\text{-}at}\ \tau(A))pxy, \\
\mathrm{et}_{\vec{x}_{\vec{u}}}(\mathtt{cases\text{-}at}\ \forall p A) & := & \lambda \mathrm{fv}(\forall p A)(\mathtt{if\text{-}at}\ \tau(A)).
\end{array}
$$

6.1.3. DEFINITION. Given a formula A and a term r of type $\tau(A)$ we now define the formula $r \mathrel{\mathrm{mr}} A$ (*r (modified) realises A*.

$$
\begin{array}{rcl}
\mathtt{nil} \mathrel{\mathrm{mr}} P & := & P \\
r \mathrel{\mathrm{mr}} \exists x A & := & \begin{cases} \mathtt{nil} \mathrel{\mathrm{mr}} A[r/x] & \text{if } \tau(A) = \mathtt{nulltype}, \\ (\mathtt{cdr}\ r) \mathrel{\mathrm{mr}} A[(\mathtt{car}\ r)/x] & \text{otherwise.} \end{cases} \\
r \mathrel{\mathrm{mr}} \forall x A & := & \begin{cases} \forall x.\mathtt{nil} \mathrel{\mathrm{mr}} A & \text{if } \tau(A) = \mathtt{nulltype}, \\ \forall x.rx \mathrel{\mathrm{mr}} A & \text{otherwise.} \end{cases} \\
r \mathrel{\mathrm{mr}} A \to B & := & \begin{cases} \mathtt{nil} \mathrel{\mathrm{mr}} A \to r \mathrel{\mathrm{mr}} B & \text{if } \tau(A) = \mathtt{nulltype}, \\ \forall x.x \mathrel{\mathrm{mr}} A \to \mathtt{nil} \mathrel{\mathrm{mr}} B & \text{if } \tau(A) \neq \mathtt{nulltype} = \tau(B), \\ \forall x.x \mathrel{\mathrm{mr}} A \to rx \mathrel{\mathrm{mr}} B & \text{otherwise.} \end{cases} \\
r \mathrel{\mathrm{mr}} A \& B & := & \begin{cases} (r \mathrel{\mathrm{mr}} A) \,\&\, (\mathtt{nil} \mathrel{\mathrm{mr}} B) & \text{if } \tau(B) = \mathtt{nulltype}, \\ (\mathtt{nil} \mathrel{\mathrm{mr}} A) \,\&\, (r \mathrel{\mathrm{mr}} B) & \text{if } \tau(A) = \mathtt{nulltype}, \\ (\mathtt{car}\ r) \mathrel{\mathrm{mr}} A \,\&\, (\mathtt{cdr}\ r) \mathrel{\mathrm{mr}} B & \text{otherwise.} \end{cases}
\end{array}
$$

6.1.4. THEOREM. *(Soundness). For each derivation $\vec{u}\colon \vec{A} \vdash d\colon B$ we can derive $\mathrm{et}_{\vec{x}_{\vec{u}}}(d) \mathrel{\mathrm{mr}} B$.*

[1] Up to now program extraction is defined and implemented only for proofs not involving the axiom-scheme-forms `all-all^`, `nat^-ind`, `boole^-ind`, `cases^`, `cases-with-undef` or any global assumptions or axiom constants whose formula is non-Harrop.

Proof by induction on d. The proof can be found in [13, 8, 18]. See also [3, 5].

Clearly if B is negative then `nil` mr $B = B$. Hence for formulas of the form $\forall x^\rho \exists y^\sigma B(x, y)$ with negative $B(x, y)$ we have $\tau(\forall x \exists y B(x, y)) = \rho \to \sigma$ and

$$r \text{ mr } \forall x \exists y B(x, y) = \forall x B(x, rx).$$

Therefore we obtain as a corollary to the soundness theorem the

6.1.5. Theorem. *(Extraction). From a derivation* $\vec{u}{:}\,\vec{A} \vdash d{:}\,\forall x^\rho \exists y^\sigma B(x, y)$, *with* $\vec{A}, B$ *negative, one can extract a closed term* $\mathrm{et}_{\vec{x}\,\vec{u}}(d)^{\rho\to\sigma}$ *such that the formula* $\forall x B(x, \mathrm{et}_{\vec{x}\,\vec{u}}(d)x)$ *is provable from* $\vec{A}$.

Usually $\vec{A}$ are lemmas, i.e. true formulas which we could but do not prove in order to keep the derivation d short. The theorem says that this abbreviation does not affect program extraction.

6.2 Front end

The realiser type $\tau(A)$ is computed by `(formula-to-et-type` A`)`. Some examples are

```
(formula-to-et-type (pf "all m ex n.m<n"))
                                  ==> (arrow nat nat)
(formula-to-et-type (pf "all l ex m ex n.m+n=k"))
                                  ==> (arrow nat (star nat nat))
(formula-to-et-type (pf "(all l ex m. l<m) -> ex n.0<n"))
                                  ==> (arrow (arrow nat nat) nat)
```

The extracted term of a **normal proof** $\vec{A} \vdash d{:}\,B$ can be obtained by `(et` $\vec{A} \vdash d{:}\,B$`)`.

6.3 Program extraction from classical proofs

A classical, i.e. non-constructive proof of an existential statement can under certain circumstances be translated into a constructive proof, and hence yields an algorithm. For background we refer to [5] where a refinement of the A-translation going back to work of Friedman [9] and Leivant [14] is described.

6.3.1 `classical-proof-to-constr-proof`

`(classical-proof-to-constr-proof` *proof*`)` transforms a classical proof of an $\forall\exists$−statement to a constructive one. The user is allowed to use free or global Π^0_1−assumptions, i.e. the goal should be of the form

$$\forall \vec{x}_1 .\, A_1 \to \ldots \to \forall \vec{x}_{m-1} .\, A_{m-1} \to \forall \vec{x}_m .\, (\forall \vec{y} .\, B_0 \to \ldots \to B_n \to \bot) \to \bot$$

with conjunction free Π^0_1−formulas A_j, $0 \le j < m$ and quantifier and conjunction free formulas $B_j \equiv \mathbb{B}_j[\vec{x}_1, \ldots, \vec{x}_m, \vec{y}]$, $0 \le j \le n$. (The implementation will be extended to a formalization with conjunctions, as soon as possible.) The constructive proof can be used to extract a program.

6.3.2 `classical-proof-to-program`

`(classical-proof-to-program` *proof*`)` yields a program at once. Since internally programs of the A-translated assumptions are calculated separately, this procedure is more efficient.

Chapter 7

Normalization by evaluation

In 2.5 we described the operational semantics of the MINLOG object language in terms of the rewrite system $\rightarrow_{\beta\eta\uparrow R}$ although we did not mention how reduction is actually implemented. In this chapter we outline the basic concept of normalization-by-evaluation and also give a correctness proof.

The fundamental idea of normalization-by-evaluation is to use the evaluation mechanism of a functional metalanguage (such as SCHEME) in order to compute the normal forms of terms without executing the reduction steps and substitution (which is notoriously inefficient in standard implementations) in particular.

7.1 Abstract normalization-by-evaluation

For the theoretical background we refer to [4]; a new formalization of normalization-by-evaluation, extended by term rewriting for constants, will be included, soon.

7.2 Implementation

7.2.1 The model

MINLOG is implemented in the untyped[1] functional programming language SCHEME. Objects of a function space $D_\rho \to D_\sigma$ are SCHEME-functions which could be obtained by evaluating a SCHEME-expression like `(lambda (a) ...)`. The product space is formed with SCHEME's primitive pairing function `cons` and the projections `car` and `cdr`. So we only remain to realize the domains D_ι which are supposed to contain terms in long normal form: Those are — like all other MINLOG-objects — represented as quoted list-expressions, i.e., SCHEME-lists of symbols. For instance the normal term $\lambda\hat{n}\hat{n}$ for identity on `nat` can be obtained by evaluating `(quote (lambda (n^) n^))`.

7.2.2 Interpretation

The evaluation function $[\![\cdot]\!]_\xi$ is, of course, provided by SCHEME's `eval`-function which animates λ-abstraction and pairing by the internal SCHEME-mechanisms.[2]

[1] More precisely: SCHEME is weakly typed by distinguishing between function objects of certain arities, pairs, characters, natural numbers, symbols and strings.

[2] According to [1] the `eval`-function is not part of the standard SCHEME-repertoire. Nevertheless it is provided by all of the standard implementations.

`eval` is parametric in the current environment which we have called ξ. It contains all active procedure definitions and the bindings of evaluated λ-instantiations. Some implementations even allow to feed ξ as an extra argument to `eval`.[3]

7.2.3 Quote and unquote

In MINLOG, the function ϕ_ρ is accessible as

(`object-to-long-normal-form` *object* ρ)

ψ_ρ (restricted to VNF) is called by

(`normal-app-expression-to-object` *term* ρ)

Remark that it does neither check if *term* is actually of the form $x\vec{r}$ nor if it has the proper type.

7.2.4 Animation of program-constants and function-symbols

The user of MINLOG is responsible to guarantee that his definitions of program-constants and function-symbols actually respect the requirements we formulated above. In particular

- A program constant of type $\rho * \sigma \to \rho'$ expects SCHEME-pairs as first argument.
- A program constant of type $(\rho \to \sigma) \to \rho'$ expects a SCHEME-function as argument.
- If no rewriting is necessary, the program constant animation should implement reproduction using `object-to-long-normal-form` and `normal-app-expression-to-object`. For instance the function `iterator` of type `(nat→nat)→nat` should read somewhat like

```
(define iterator
  (lambda (f)
    (lambda (n)
      (lambda (arg)
        (cond
         ((zero-nat? n) arg)
         ((suc-nat? n)
          (let* ((n-1 (pred-nat n))
                 (prev (((iterator f) n-1) arg)))
            (f prev)))
         (else ; reproduce
          (normal-app-expr-to-obj
           (c-app 'iterator (obj-to-long-normal-form f) n arg))))))))
```

7.2.5 Normalization-by-evaluation

The function *nbe* is named `term-to-eta-nf`. (`normalize` *term*) (abbreviated `nt`) applies syntactic η-normalization after using `term-to-eta-nf`.

[3]In LMU SCHEME, which is MINLOG's favourite dialect, `eval` expects an environment as second argument. The current environment can be obtained by evaluating `(the-environment)`.

7.2.6 Normalization-by-evaluation for proof terms

If one regards the formula of a proof term as its type then one needs a slightly more general notion of types which allows for the most harmless version of type-dependency through $\forall$- and $\exists$-quantification. However, this additional complexity can be removed by embedding formulas of first order predicate-logic into a simple type system containing a canonical base type `atomic` for atomic formulas as follows:

$$\begin{array}{lcl} \mathrm{type}(P) & := & \texttt{atomic}, \\ \mathrm{type}(\exists x^{\rho} A) & := & \texttt{existential}, \\ \mathrm{type}(\forall x^{\rho} A) & := & \rho \to \mathrm{type}(A), \\ \mathrm{type}(A \to B) & := & \mathrm{type}(A) \to \mathrm{type}(B), \\ \mathrm{type}(A \& B) & := & \mathrm{type}(A) * \mathrm{type}(B). \end{array}$$

Using this conversion (which is done by `formula-to-type`) proof terms are typed correctly and thus can be normalized. Axiom-scheme-form reductions form the rewrite relation $\to_R$.

Proof term normalization by `(normalize-proof` *proof*`)` also produces η-contractive normal-forms by executing η-normalization after $\beta\eta\!\uparrow$-normalizing (using `t-expr-to-eta-nf`).

7.2.7 Internals

The new variable that is required in the definition of ϕ is produced by `(gensym` *string*`)` which returns a new symbol beginning with *string*.

Chapter 8

The TEX-output

MINLOG proofs can be output into a TEX-file that only needs a macro package `proofmacros.tex` for compilation.[1] There is also a possibility to convert other MINLOG-objects such as types, terms and formulas into a TEXable string, which then can be exported into LATEX-source-files using simple copy and paste.

8.1 How to output

Due to the inconveniences of non-normal proofs, the TEX-output can only process normalised proofs.

`(out-pproof "`*file*`")` provides the easiest way to get a TEX-output of the current partial proof. The partial proof `(pproof)` is normalised, transformed into a string and written on *file*.

`(out-pproof-as-assertion` *theorem-identifier theorem-name* `"`*file*`"` *handling* `['false])` additionally adds some layout around the proof. It first typesets the string *theorem-identifier* boldfaced, then the string *theorem-name* in italics, then the proven formula. Finally it adds a newline and starts the proof output with a leading italic *Proof.*

The *handling* is a symbol that rules how the file *file* is accessed:

`'once` opens the file, ships out all text and closes it again.

`'open` does all the same, except closing the file again.

`'append` appends the text to the previously opened file.

`'append-and-close` does the same, but closes the file afterwards.

Using the command `(append-to-tex-proof-file` *astring*`)` one can add further text to the file. `(close-tex-proof-file)` closes it.

`(normalized-proof-to-output-string` *proof*`)` simply computes the TEX-output of *proof* and returns its string (which is to be put in TEX-normal-mode).

`(normalized-proof-to-output-string-as-assertion` *theorem-identifier theorem-name proof*`)` produces the analogon to the result of `out-pproof-as-assertion` as a string.

`(formula-to-output-string` *formula*`)` yields a representation of the formula that already contains encapsulating `$`-signs for invoking math-mode.

[1] We have chosen TEX instead of LATEX or LATEX2ε for compatibility reasons, since this format can be read by all three of them.

`(term-to-output-string` *term*`)` produces a string which has to be included in a TEX-math-mode environment.

`(type-to-output-string` *type*`)`. The resulting string also needs a math-environment.

`(global-assumptions-to-output-string)` returns a string that lists all active global assumptions with their formulas.

The following routines allow to work with the TEX-proof-file (the functions `out-pproof` and `out-pproof-as-assertion` also use them):

`(open-tex-proof-file` *file*`)` opens the file and inserts an include line for `proofmacros.tex`.

`(append-to-tex-proof-file` *string*`)` adds the *string* to the TEX-proof-file.

`(close-tex-proof-file)` does the obvious.

8.2 How to modify the output of types, terms and formulas

In order to obtain optimal results one needs to supply some printing information for all newly introduced MINLOG-objects. Therefore there exists a list of functions of the form `add-...-information`. We will start with `add-program-constant-information`, since most other functions have similar and less options.

8.2.1 `add-ground-type-information`

New ground-types are by default printed in roman font. If the user likes to assign a particular output-string to his newly invented basic type, he can use

`(add-ground-type-information` *type-symbol string*`)`.

8.2.2 `add-program-constant-information`

The function has the following syntax

`(add-program-constant-information` *pc-symbol display-string precedence arg-num* [*positioner* [*modifier mod-positioner*]]`)`

display-string is the output string used for the program-constant in **TEX-math-mode-format** with all backslashes `\` changed into `@`.[2] This could be for instance `@cdot` for a $\cdot$-printout. Depending on the *positioner* chosen, *display-string* may also be a list of strings as, e.g., required for the `wrap-around`-positioner (for details see below).

precedence is a natural number that is used in comparison with other terms in order to avoid parentheses. Low value means high precedence, which means the object scarcely needs to be put in brackets.

`'no-brackets` can be used instead of 0 and `'in-brackets` is synonymous for 100.

The precedence is first used when the program constant is applied to argument terms (the number of argument-terms is recognised using *arg-num*) in order to judge whether the arguments have

[2] The TEXnician will realise that `@`'s cat-code is changed into 0 by `proofmacros.tex`.

to be put in brackets. Then also the precedence of the whole application expression is set to the operator's precedence, except when the precedence is `'superior` or `'inferior` in which case the maximal (minimal) precedence of the arguments is used for the application.

In rare cases one wants to use different values for the two occasions where *precedence* is used. This can be achieved by giving a 2-element-list of precedences, where the latter is used for computing the precedence of an operator-application.

arg-num declares the number of arguments the program-constant usually gets. Of course, the routine could extract this information from the type-declaration of the program-constant. However, there are cases when a program-constant might expect fewer arguments than its type might suggest, as e.g., the `curry`-function of type `(nat`×`nat`→`nat)`→`nat`→`nat`→`nat` demonstrates.

positioner is a symbol for a function that is used to typeset the program-constant when applied to its regular number of arguments.

`as-operator` prefixes the *display-string* to an embraced list of the arguments separated by commas. If for instance we have declared

```
(add-program-constant-information 'rel "R" 3 2 'as-operator)
```

we get $R(n+2\,,3\,)$ as TEX-display for `(pt "rel (n+2) 3")`.

Note that the arguments will never be parenthesised with `'as-operator`. Hence, the *precedence* is only used to compute the precedence of the resulting term.

`infix` is used to put the *display-string* of the program-constant between each pair of arguments. Thus `infix` serves even for program-constants of arbitrary arity. E.g., typing

```
(add-program-constant-information 'rel "@verb$_$" 3 4 'infix)
```

produces $(n+2\,)$_0 _$?^{\mathbb{N}}$_1 for `(pt "rel (n+2) 0 undef_nat 1")`. As can be seen from this example, the precedence info is used for determining that $n+2$ has to be put in brackets.

`prefix` and `postfix` simply place the *display-string* before (after) the argument displays which are not separated by any separator string. Therefore the two options will usually only be sensible for one-argument program-constants. If we change the program-constant-information of the last example to `'postfix` we obtain $(n+2\,)0\ ?^{\mathbb{N}}\,1$ _. An even more realistic example is the unary `zero?`-operator:

```
(add-program-constant-information 'zero?  "@,=0?" 2 1 postfix)
```

yields $\hat{n}\ = 0?$ for the term `(pt "zero?  n^")`.

`wrap-around` assumes that the *display-string* in fact is not a string but a list of strings. These will be distributed before, among and after the argument displays. The most prominent example is the `if-nat`-operator whose program-constant-information is provided by

```
(add-program-constant-information
  'if-nat
  '("[@,{@sf if@ }" "@ {@sf then@ }" "{@sf @ else@ }" "@,]")
  '(superior no-brackets) 3 'wrap-around)
```

The term `(pt "[if [if T then F else F] then 2 else 3]")` is output as

$[\,\mathsf{if}\,[\,\mathsf{if}\ \top\ \mathsf{then}\ \perp\ \mathsf{else}\ \perp\,]\ \mathsf{then}\ 2\ \mathsf{else}\ 3\,]$

There is another positioner named `hoch-n` that will be explained below. The subsection on internals shows how new positioners can be written easily to achieve an even more sophisticated output.

modifier mod-positioner. Some operators require additional effort to cover iterated applications. First of all, there are right-associative (e.g., the list operator `cons`), left-associative and left-and-right-associative operators (such as addition and multiplication). Second there are functionals f whose n-fold iteration on the argument a should print out $f^n(a)$ instead of $f(f(\ldots f(a)\ldots))$.

Such extravagancies are signalled by the optional *modifier* with its *mod-positioner*. *modifier* is a symbol for a function that gets as argument the whole application of the operator in question. If it answers different than `#f` then the *mod-positioner* is used for positioning and fed the result of the *modifier*-call. Usually, the user does not have to bother about these internals, since he will need only the following kind of standard *modifier-mod-positioner*-combinations.

- Left-right-associative infix-operators. E.g.,

  ```
  (add-program-constant-information '+-nat "+" 4 2 'infix
  'associative 'infix)
  ```

  produces $2 + 3 * 4 + 1 + 2$ for `(pt "2 + (3 * 4) + (1 + 2)")`[3]

- Right- or left-associative infix operators. E.g.,

  ```
  (add-program-constant-information 'add "::" 3 2 'infix
  'right-associative 'infix )
  ```

  results in the output $2 :: (3 + 4\,) :: eps$ for `(pt "add 2 (add (3+4) eps)")`.

- Iterated applications as exponentiation. E.g.,

  ```
  (add-program-constant-information 'succ "S" 1 1 'prefix
  'count-repeated-op 'hoch-n)
  ```

  yields $S^3 n$" for `(pt "succ (succ (succ n)))` and $S0$ for `(pt "succ 0")`.

More examples can be taken from the list of program-constant-information for the standard MINLOG-program-constants:

```
((nil "@,{@rm nil}@," 0 0)
 (true "@,@top@," 0 0)
 (false "@,@bot@," 0 0)
 (undef_boole "@,?^@B@," 0 0)
 (if-boole ("[@,{@sf if}$ $" "$ {@sf then} $" "$ {@sf else} $" "@,]")
           (superior no-brackets) 3 wrap-around)
 (undef_nat "@,?^@N@," 0 0)
 (zero-nat "@,0@," 0 0)
 (0 "@,0" 0 0)
 (1 "@,1" 0 0)
 (pred-nat "@$" 1 1 prefix count-repeated-op hoch-n)
 (=-strict-boole "=" 6 2 infix)
 (and_strict "@land " 5 2 infix)
 (=-strict-nat "=" 6 2 infix)
```

[3] That $3 * 4$ is not in brackets results from the stronger precedence 3 for $*$ than 4 for $+$.

```
(=-nat "=" 6 2 infix)
(<-strict-nat "<"  6 2 infix)
(<=-strict-nat "@leq " 6 2 infix)
(if-nat ("[@,{@sf if}$ $" "$ {@sf then} $" "$ {@sf else} $" "@,]")
        (superior no-brackets) 3 wrap-around)
(plus-nat "+" 4 2 infix associative infix)
(*-nat "*" 3 2 infix associative infix)
(minus-nat "-" 4 2 infix)
(max_nat "max" 1 2 as-operator)
(min_nat "min" 1 2 as-operator)))
```

If one omits program-constant-information for a new program-constant c then the output routine by default assumes that c is a unary function to be displayed `as-operator`.

8.2.3 `add-variable-information` and `variable-output-flags`

Most of MINLOG's variables shall print out as one is used to from mathematical texts. Anyway, if one tries, e.g., to prove a property about any binary relation `rel` on natural numbers then the resulting theorem should read something like $\forall \texttt{rel}, m, n.\texttt{rel}(m, n) \to \ldots$. This requires that the variable `rel` is treated similar to a program-constant by the output routine. Consequently the syntax of the respective function `add-variable-information` is identical to that of `add-program-constant-information`:

> `(add-variable-information` *pc-symbol display-string precedence arg-num* [*positioner* [*modifier mod-positioner*]]`)`

Whenever for a variable x no variable information is given then x will be displayed according to the `variable-output-flags`. This is a global variable which may contain the following symbols.

`show-partial-sign`. If present this flag ensures that every partial variable gets a "hat", i.e., is displayed $\hat{m}$ rather than m.

`applications-in-brackets`. When this flag is set, applications of variables to variables, such as fm are printed $f(m)$.

`show-number`. In rare cases the number indices of variables may not be necessary for understanding, so they can be left out using this flag.

`type-quantifications`. This flag takes care for λ-abstractions and $\forall\exists$-quantifications where it supplies a type superscript to the variable display. Thus the usual $\forall R,\, m\, R(m, m)$ becomes $\forall R^{\mathbb{N},\mathbb{N}\to\mathbb{B}},\, m^{\mathbb{N}}\, R(m, m)$.

`convert-dummy-variables` transforms the names of MINLOG's gensymmed variables (which sometimes really look awful as the example `nat=>nat=>boole*seq^00124` demonstrates) into a default string, which is produced by the function `(dummy-variable-to-string` *var-symbol type*`)`. This can easily be changed and prints a δ by default.

8.2.4 `add-function-symbol-information`

The syntax of this function is the same as that of `add-program-constant-information` except that, of course, no *arg-num* is required, since it can already be found in `FUNCTION-SYMBOLS`. By default, function-symbols get printed boldfaced `'prefix` with precedence `'inferior`.

MINLOG's standard function symbols have the following default output-information.

```
((eq_boole "=" 3 infix)
 (def-boole "@downarrow" 6 postfix)
 (extends_boole "@sqsubseteq " 3 infix)
 (imp_fsym "@supset " 3 infix)
 (and_fsym "@& " 3 infix)
 (eq_nat "=" 3 infix)
 (less_nat "<" 3 infix)
 (leq_nat "@leq " 3 infix)
 (def-nat "@downarrow" 6 postfix)
 (extends_nat "@sqsubseteq " 3 infix)))
```

8.2.5 `add-program-scheme-form-information`

Essentially, program-scheme-forms are treated just like program-constants, except that their *display-string* is parametric in their argument:[4]

> `(add-program-scheme-form-information` *pc-symbol display-fct precedence arg-num* [*positioner* [*modifier mod-positioner*]]`)`

Here *display-fct* is a **quoted** unary function that gets fed the polymorphic argument of the program-scheme-form (which needs to be evaluated to become a type) and is supposed to produce a *display-string* as described for `add-program-constant-information` (this means in particular that — in case the *positioner* were `wrap-around` — *display-string* should be a list of strings). As an example we show the clause for the recursion operator:

```
(add-program-scheme-form-information
 'nat-rec-at
 '(lambda (arg)
    (string-append "{@cal R}_{"
                   (type-to-output-string (ev arg))
                   "}"))
 3 2 'as-operator)
```

results in the output $\mathcal{R}_{\mathbb{N}}(0\,, \lambda\hat{n}, \hat{m}.\hat{m}+\hat{n})$ for

```
(((nat-rec-at 'nat)
  0)
 (lambda (n^)
    (lambda (m^)
      ((plus-nat m^) n^))))
```

8.3 How to modify the output of proofs

Analogous to `variable-output-informations` there is a global SCHEME-variable, called `proof-display-flags`, that rules how atomic proofs (assumptions, global assumptions, axioms, axiom-scheme-forms) are printed by default. When, e.g., an assumption is used, the output-routine prints out something like "This follows from" and then adds some information on the atomic proof from what "This" actually is derived.

[4] Recall that a program-scheme-form is typically of the form `(...-at (quote` *type*`))`.

`show-formulas-of-assumptions`. This prints out the formula of an assumption each time it is used.

`show-numbers-of-assumptions`. This prints the number of the assumption in square brackets.

By default, both flags are set on and produce, e.g., the left example output. the right part has been obtained without `show-formulas-of-assumptions`.

> This follows from $\bot \to m = 0$ (an instance of *ex-falso-quodlibet*) where $m = 0 \searrow_* \swarrow m + 0 = 0$ using $m < 0$ [1] where $\bot \searrow_* \swarrow m < 0$.

> This follows from $\bot \to m = 0$ (an instance of *ex-falso-quodlibet*) where $m = 0 \searrow_* \swarrow m + 0 = 0$ using [1] where $\bot \searrow_* \swarrow m < 0$.

The next triple of options covers the printout of axioms.

`show-formulas-of-axioms` prints out axioms (similar to `show-formulas-of-assumptions`).

`show-names-of-axioms` prints the name of the axiom (e.g., *truth-axiom*).

`show-axiom-text` adds the word "axiom" to the output, yielding, e.g., *this follows from the axiom* ... instead of just *this follows from.*

The similar flags for global assumptions and axiom-scheme-forms are named `show-formula-of-global-assumptions`, `show-names-of-global-assumptions`, `show-global-assumption-text`, `show-formulas-of-axiom-scheme-forms`, `show-axiom-scheme-form-text` and `show-names-of-axiom-scheme-forms`.

Apart from these default regulations you can assign to each singular global assumption, axiom and axiom-scheme-form individual printout-information by using the functions `add-global-assumption-information`, `add-axiom-information` and `add-axiom-scheme-form-information`. Each of these commands allows for the following syntax:

(`add-...-information` '*symbol string show-formula? show-text? show-the?*)

string is the output string used if *show-text?* is `#t`.

show-formula? is a SCHEME-truth-value. If it is `#t` then the formula of the object is displayed.

show-text? is a SCHEME-truth-value. If it is `#t` then the *string* is displayed.

show-the? is a SCHEME-truth-value. If it is `#t` then the word *the* is prefixed.

As an example we show how the axiom-of-truth is displayed:

```
(add-axiom-information 'truth-axiom-symbol " of truth" #f #t #t)
```

In an example proof output this leads to the following text.

> For $\forall m^{\mathbb{N}}, n^{\mathbb{N}}.m < n \to m + 1 < m + 2$ assume m, n and $m < n$ [1] and use the axiom of truth where $m + 1 < m + 2 \searrow_* \swarrow \top$.

Notice that for each of the standard axiom-scheme-forms there is a specific output procedure that is executed when the axiom-scheme-form is used with sufficiently many arguments. Therefore the `axiom-scheme-form-informations` are scarcely used.

8.4 Internals

Since the TEX-print procedures are rather special and the front end allows quite sophisticated access to the various options, the user normally need not confront himself with the internals except for one detail that allows for some impressive outputs: When processing term applications $oa_1 \ldots a_n$ of operators o with a printing-information (provided by the `add-...-information`-functions) declaring a *positioner* (such as `as-operator`) it calls a function with the name of the positioner as follows:

(*positioner op-display-string operator-prec argument-outputs bracket-pointer*)

The positioner function is supposed to return the *output* of the whole application term, where an output is a 2-element-list consisting of

- a *string* that is used for display and
- a precedence that will be used for displaying the *string* in the context of other term operators.

The arguments of the function are

op-display-string. This is exactly the *display-string* given in the printing information.

operator-precedence. This is the precedence given in the printing information (the first if two are given as a list, see above).

argument-outputs. A list of outputs (i.e., lists (*string prec*)) for the arguments of the application.

bracket-pointer is either `'round` or `'square` and is used by an auxiliary function `embrace` serving to put certain parts of the application in parentheses.

Most of the positioners mainly bother composing the output-string and leave the computation of the resulting precedence to the `refine`-procedure. For instance, the positioner `as-operator` reads

```
(define (as-operator operator
                     operator-prec
                     argument-outputs
                     bracket-pointer)
  (list
   (string-append operator
                  (bracket-pointer-to-open bracket-pointer)
                  (comma-strings "," (map car argument-outputs))
                  (bracket-pointer-to-close bracket-pointer))
   (refine operator-prec argument-outputs)))
```

This results in the following behaviour:

```
(as-operator "op" 3 '(("a" 2) ("b" 3) ("c" 1)) 'square) ==> ("op(a,b,c)" 3)
```

In order to demonstrate how one can use the modularity of the *positioner*-functions to achieve pleasing TEX-outputs we give an example. Suppose we are working in a graph theoretic setting where we have a program-constant `path` that takes a graph g (i.e., a relation on $\mathbb{N} \times \mathbb{N}$ which is represented as object of type `nat→nat→boole`), a vertex i (i.e., a natural number) a list x of vertices (of type `seq` for sequences) and two further vertices j and k and returns a boolean element telling whether x is a path in g between j and k passing no elements less than i.[5] The graph theoretic lecture notes we took the example from suggests that `(pt "path g i x j k")` should be displayed $P_i(x, j, k)$. Notice that the graph-argument is omitted as it will be always constant in proofs. To achieve such an output we introduce a new *positioner* `as-path` and state the program-constant-information of `path` accordingly:

```
(add-program-constant-information 'path "P" 2 5 'as-path)
```

The positioner is implemented as follows:

```
(define (as-path operator
                 operator-prec
                 argument-outputs
                 bracket-pointer)
  (list
   (string-append "P_{"
                  (caadr argument-outputs)
                  "}("
                  (comma-strings "," (map car (cddr argument-outputs)))
                  ")")
   (refine operator-prec argument-outputs)))
```

To complete the picture we quote an excerpt of the proof-display:

> **Theorem** *(Warshall).*
> $\forall R, i, j, k \exists \hat{x}.(\hat{x} = \bot \to \forall y \neg P_i(y, j, k)) \wedge (\neg \hat{x} = \bot \to (P_i(\hat{x}, j, k) \wedge \mathrm{Wf}(\hat{x})))$.
>
> *Proof.* Let R be given. Then we have to show $\forall i, j, k \exists \hat{x}.(\hat{x} = \bot \to \forall y \neg P_i(y, j, k)) \wedge (\neg \hat{x} = \bot \to (P_i(\hat{x}, j, k) \wedge \mathrm{Wf}(\hat{x})))$.
> This is shown by induction on i.
>
> $i = 0$. Assume j and k. Then we have to show $\exists \hat{x}.(\hat{x} = \bot \to \forall y \neg P_0(y, j, k)) \wedge (\neg \hat{x} = \bot \to (P_0(\hat{x}, j, k) \wedge \mathrm{Wf}(\hat{x})))$.
> This is shown by cases on $j = k$. ...

Finally we list some administrable functions that can be used for programming positioners.

(`comma-strings` *comma-string precedence-list*) composes a string of *string-list*, separated by *comma-string*.

(`refine` *operator-prec alist*) computes the precedence of an application composed of an operator with *operator-prec* and precedences of the arguments *precedence-list*.

(`in-brackets` *string bracket-pointer*) puts the string in brackets according to bracket-pointer ($\in$ {`'round`,`'square`}).

(`embrace` *(string prec) op-prec bracket-pointer*) puts the string in the brackets chosen if *prec* compared to *op-prec* advises to do so.

[5] By the complexity of the example one can guess that it is born in practice. In fact, such things arise when proving the correctness of the Warshall-algorithm.

Bibliography

[1] H. Abelson, N.I. Adams, D.H. Bartley, G. Brooks, R.K. Dybvig, D.P. Friedman, R. Halstead, C. Hanson, C.T. Haynes, E. Kohlbecker, D. Oxley, K.M. Pitman, G.J. Rozas, G. Sussman, and M. Wand. Revised report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[2] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[3] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 91–106. Springer Verlag, Berlin, Heidelberg, New York, 1993.

[4] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[5] Ulrich Berger and Helmut Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, IN, USA, October 1994*, volume 960 of *Lecture Notes in Computer Science*, pages 77–97. Springer Verlag, Berlin, Heidelberg, New York, 1995.

[6] R.L. Constable, S.F. Allen, H.M. Bromley, W.R Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice–Hall, New Jersey, 1986.

[7] Roy Dyckhoff. Contraction–free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57:793–807, 1992.

[8] Anne S. Troelstra (editor). *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics.* Springer Verlag, Berlin, Heidelberg, New York, 1973.

[9] 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.

[10] Jörg Hudelmaier. *Bounds for Cut Elimination in Intuitionistic Propositional Logic.* PhD thesis, Mathematische Fakultät, Eberhard–Karls–Universität Tübingen, 1989.

[11] Jörg Hudelmaier. Bounds for cut elimination in intuitionistic propositional logic. *Archive for Mathematical Logic*, 31:331–354, 1992. Lemma 4 is true (and needed) for atomic u only.

[12] Achim Jung. Domain theory. In *Handbook of Logic in Computer Science.* Oxford University Press, 1992.

[13] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North–Holland, Amsterdam, 1959.

[14] Daniel Leivant. Syntactic translations and provably recursive functions. *The Journal of Symbolic Logic*, 50(3):682–688, September 1985.

[15] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. Submitted for publication, April 1992.

[16] Gordon D. Plotkin. Domains. Lecture Notes.

[17] Dana Scott. Domains for denotational semantics. In E. Nielsen and E.M. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer Verlag, Berlin, Heidelberg, New York, 1982. A corrected and expanded version of a paper prepared for ICALP'82, Aarhus, Denmark.

[18] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics. An Introduction*, volume 121, 123 of *Studies in Logic and the Foundations of Mathematics.* North–Holland, Amsterdam, 1988.

[19] Jaco van de Pol and Helmut Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 350–364. Springer Verlag, Berlin, Heidelberg, New York, 1995.