

MINLOG REFERENCE MANUAL

CONTENTS

Acknowledgements	2
1. Introduction	2
1.1. Simultaneous free algebras	3
1.2. Partial continuous functionals	3
1.3. Primitive recursion, computable functionals	4
1.4. Decidable predicates, axioms for predicates	4
1.5. Minimal logic, proof transformation	4
1.6. Comparison with Coq and Isabelle	5
2. Types, with simultaneous free algebras as base types	6
2.1. Generalities for substitutions, type substitutions	6
2.2. Simultaneous free algebras as base types	7
3. Variables	10
4. Constants	12
4.1. Rewrite and computation rules for program constants	13
4.2. Recursion over simultaneous free algebras	14
4.3. Internal representation of constants	16
5. Predicate variables and constants	18
5.1. Predicate variables	18
5.2. Predicate constants	19
5.3. Inductively defined predicate constants	20
6. Terms and objects	21
6.1. Normalization	23
6.2. Substitution	25
7. Formulas and comprehension terms	25
8. Assumption variables and constants	30
8.1. Assumption variables	30
8.2. Axiom constants	32
8.3. Theorems	35
8.4. Global assumptions	37
9. Proofs	37
9.1. Constructors and accessors	37
9.2. Normalization	39
9.3. Substitution	40
9.4. Display	41
9.5. Classical logic	41
10. Interactive theorem proving with partial proofs	42
10.1. Partial proofs	42
10.2. Interactive theorem proving	42
11. Search	47

12. Computational content of classical proofs	49
13. Extracted terms	50
14. Reading formulas in external form	51
14.1. Lexical analysis	51
14.2. Parsing	52
References	55
Index	57

Acknowledgements. The MINLOG system has been under development since around 1990. My sincere thanks go to the many contributors: Holger Benl (Dijkstra algorithm, inductive data types), Ulrich Berger (very many contributions), Michael Bopp (program development by proof transformation), Wilfried Buchholz (translation of classical proof into intuitionistic ones), Laura Crosilla (tutorial), Matthias Eberl (normalization by evaluation), Dan Hernest (functional interpretation), Felix Joachimski (many contributions, in particular translation of classical proofs into intuitionistic ones, producing Tex output, documentation), Ralph Matthes (documentation), Karl-Heinz Niggl (program development by proof transformation), Jaco van de Pol (experiments concerning monotone functionals), Martin Ruckert (many contributions, in particular the MPC tool), Robert Stärk (alpha equivalence), Monika Seisenberger (many contributions, including inductive definitions and translation of classical proofs into intuitionistic ones), Klaus Weich (proof search, the Fibonacci numbers example), Wolfgang Zuber (documentation).

1. INTRODUCTION

MINLOG is intended to reason about computable functionals, using minimal logic. It is an interactive prover with the following features.

- Proofs are treated as first class objects: they can be normalized and then used for reading off an instance if the proven formula is existential, or changed for program development by proof transformation.
- To keep control over the complexity of extracted programs, we follow Kreisel's proposal and aim at a theory with a strong language and weak existence axioms. It should be conservative over (a fragment of) arithmetic.
- MINLOG is based on minimal rather than classical or intuitionistic logic. This more general setting makes it possible to implement program extraction from classical proofs, via a refined A -translation (cf. [3]).
- Constants are intended to denote computable functionals. Since their (mathematically correct) domains are the Scott-Ershov partial continuous functionals, this is the intended range of the quantifiers.
- Variables carry (simple) types, with free algebras as base types. The latter need not be finitary (so we allow e.g. countably branching trees), and can be simultaneously generated. Type parameters (ML style) are allowed, but we keep the theory predicative and disallow type quantification.

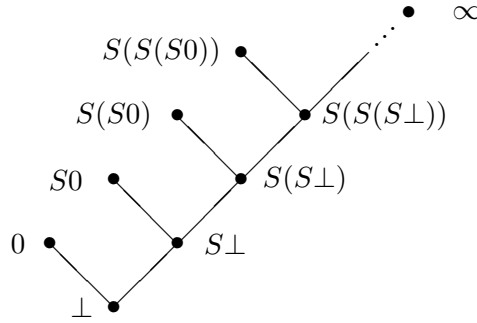


FIGURE 1. The domain of natural numbers

- To simplify equational reasoning, the system identifies terms with the same normal form. A rich collection of rewrite rules is provided, which can be extended by the user. Decidable predicates are implemented via boolean valued functions, hence the rewrite mechanism applies to them as well.

We now describe in more details some of these features.

1.1. Simultaneous free algebras. A free algebra is given by *constructors*, for instance zero and successor for the natural numbers. We want to treat other data types as well, like lists and binary trees. When dealing with inductively defined sets, it will also be useful to explicitly refer to the generation tree. Such trees are quite often countably branching, and hence we allow infinitary free algebras from the outset.

The freeness of the constructors is expressed by requiring that their ranges are disjoint and that they are injective. Moreover, we view the free algebra as a domain and require that its bottom element is not in the range of the constructors. Hence the constructors are total and non-strict. For the notion of totality cf. [12, Chapter 8.3].

In our intended semantics we do not require that every semantic object is the denotation of a closed term, not even for finitary algebras. One reason is that for normalization by evaluation (cf. [4]) we want to allow term families in our semantics.

To make a free algebra into a domain and still have the constructors injective and with disjoint ranges, we model e.g. the natural numbers as shown in Figure 1. Notice that for more complex algebras we usually need many more “infinite” elements; this is a consequence of the closure of domains under suprema. To make dealing with such complex structures less annoying, we will normally restrict attention to the *total* elements of a domain, in this case – as expected – the elements labelled 0 , $S0$, $S(S0)$ etc.

1.2. Partial continuous functionals. As already mentioned, the (mathematically correct) domains of computable functionals have been identified by Scott and Ershov as the partial continuous functionals; cf. [12]. Since we want to deal with computable functionals in our theory, we consider it as mandatory to accommodate their domains. This is also true if one is interested in total functionals only; they have to be treated as particular partial continuous functionals. We will make use of predicate constants Total_ρ with

the total functionals of type ρ as the intended meaning. To make formal arguments with quantifiers relativized to total objects more manageable, we use a special sort of variables intended to range over such objects only. For example, $\mathbf{n}0, \mathbf{n}1, \mathbf{n}2, \dots, \mathbf{m}0, \dots$ range over total natural numbers, and $\mathbf{n}^0, \mathbf{n}^1, \mathbf{n}^2, \dots$ are general variables. This amounts to an abbreviation of

$$\begin{aligned} \forall \hat{x}. \text{Total}_\rho(\hat{x}) \rightarrow A & \quad \text{by} \quad \forall x A, \\ \exists \hat{x}. \text{Total}_\rho(\hat{x}) \wedge A & \quad \text{by} \quad \exists x A. \end{aligned}$$

1.3. Primitive recursion, computable functionals. The elimination constants corresponding to the constructors are called primitive recursion operators \mathcal{R} . They are described in detail in Section 4. In this setup, every closed term reduces to a numeral.

However, we shall also use constants for rather arbitrary computable functionals, and axiomatize them according to their intended meaning by means of rewrite rules. An example is the general fixed point operator fix , which is axiomatized by $\text{fix}F = F(\text{fix}F)$. Clearly then it cannot be true any more that every closed term reduces to a numeral. We may have non-terminating terms, but this just means that not always it is a good idea to try to normalize a term.

An important consequence of admitting non-terminating terms is that our notion of proof is not decidable: when checking e.g. whether two terms are equal we may run into a non-terminating computation. But we still have semi-decidability of proofs, i.e., an algorithm to check the correctness of a proof that can only give correct results, but may not terminate. In practice this is sufficient.

To avoid this somewhat unpleasant undecidability phenomenon, we may also view our proofs as abbreviated forms of full proofs, with certain equality arguments left implicit. If some information sufficient to recover the full proof (e.g. for each node a bound on the number of rewrite steps needed to verify it) is stored as part of the proof, then we retain decidability of proofs.

1.4. Decidable predicates, axioms for predicates. As already mentioned, decidable predicates are viewed via boolean valued functions, hence the rewrite mechanism applies to them as well.

Equality is decidable for finitary algebras only; infinitary algebras are to be treated similarly to arrow types. For infinitary algebras (extensional) equality is a predicate constant, with appropriate axioms. In a finitary algebra equality is a (recursively defined) program constant. Similarly, existence (or totality) is a decidable predicate for finitary algebras, and given by predicate constants Total_ρ for infinitary algebras as well as composed types. The axioms are listed in Subsection 8.2 of Section 8.

1.5. Minimal logic, proof transformation. For generalities about minimal logic cf. [13]. A concise description of the theory behind the present implementation can be found in “Minimal Logic for Computable Functions” which is available on the MINLOG page www.minlog-system.de.

SS:Coq

1.6. Comparison with Coq and Isabelle. COQ (cf. coq.inria.fr) has evolved from a calculus of constructions defined by HUET and COQUAND. It is a constructive, but impredicative system based on type theory. More recently it has been extended by PAULIN-MOHRING to also include inductively defined predicates. Program extraction from proofs has been implemented by PAULIN-MOHRING, FILLIATRE and LETOUZEY, in the sense that OCAML programs are extracted from proofs.

The ISABELLE/HOL system of PAULSON and NIPKOW has its roots in CHURCH's theory of simple types and HILBERT's Epsilon calculus. It is an inherently classical system; however, since many proofs in fact use constructive arguments, in is conceivable that program extraction can be done there as well. This has been explored by BERGHOFER in his thesis [6].

Compared with the MINLOG system, the following points are of interest.

- The fact that in COQ a formula is just a map into the type `Prop` (and in ISABELLE into the type `bool`) can be used to define such a function by what is called *strong elimination*, say by $f(\mathbf{tt}) := A$ and $f(\mathbf{ff}) := B$ with fixed formulas A and B . The problem is that then it is impossible to assign an ordinary type (say in the sense of ML) to a proof. It is not clear how this problem for program extraction can be avoided (in a clean way) for both COQ and ISABELLE. In MINLOG it does not exist due to the separation of terms and formulas.
- The impredicativity (in the sense of quantification over predicate variables) built into COQ and ISABELLE has as a consequence that extracted programs need to abstract over type variables, which is not allowed in program languages of the ML family. Therefore one can only allow outer universal quantification over type and predicate variables in proofs to be used for program extraction; this is done in the MINLOG system from the outset. However, many uses of quantification over predicate variables (like defining the logical connectives apart from \rightarrow and \forall) can be achieved by means of inductively defined predicates. This feature is available in all three systems.
- The distinction between properties with and without computational content seems to be crucial for a reasonable program extraction environment; this feature is available in all three systems. However, it also seems to be necessary to distinguish between universal quantifiers with and without computational content, as in BERGER's [2]. At present this feature is available in the MINLOG system only.
- COQ has records, whose fields may contain proofs and may depend on earlier fields. This can be useful, but does not seem to be really essential. If desired, in MINLOG one can use products for this purpose; however, proof objects have to be introduced explicitly via assumptions.
- MINLOG's automated proof search `search` tool is based on MILLER's [10]; it produces proofs in minimal logic. In addition, COQ has many strong tactics, for instance `Omega` for quantifier free PRESBURGER arithmetic, `Arith` for proving simple arithmetic properties and `Ring`

for proving consequences of the ring axioms. Similar tactics exist in ISABELLE. These tactics tend to produce rather long proofs, which is due to the fact that equality arguments are carried out explicitly. This is avoided in MINLOG by relativizing every proof to a set of rewrite rules, and identifying terms and formulas with the same normal form w.r.t. these rules.

- In ISABELLE as well as in MINLOG the extracted programs are provided as terms within the language, and a soundness proof can be generated automatically. For COQ (and similarly for NUPRL) such a feature could at present only be achieved by means of some form of reflection.

2. TYPES, WITH SIMULTANEOUS FREE ALGEBRAS AS BASE TYPES

S:Types

Generally we consider typed theories only. Types are built from type variables and type constants by algebra type formation (**alg** $\rho_1 \dots \rho_n$), arrow type formation $\rho \rightarrow \sigma$ and product type formation $\rho \times \sigma$ (and possibly other type constructors).

We have type constants **atomic**, **existential**, **prop** and **nulltype**. They will be used to assign types to formulas. E.g. $\forall n \, n = 0$ receives the type **nat** \rightarrow **atomic**, and $\forall n, m \exists k \, n + m = k$ receives the type **nat** \rightarrow **nat** \rightarrow **existential**. The type **prop** is used for predicate variables, e.g. R of arity **nat**, **nat** \rightarrow **prop**. Types of formulas will be necessary for normalization by evaluation of proof terms. The type **nulltype** will be useful when assigning to a formula the type of a program to be extracted from a proof of this formula. Types not involving the types **atomic**, **existential**, **prop** and **nulltype** are called object types.

Type variable names are **alpha**, **beta**...; **alpha** is provided by default. To have infinitely many type variables available, we allow appended indices: **alpha1**, **alpha2**, **alpha3**... will be type variables. The only type constants are **atomic**, **existential**, **prop** and **nulltype**.

SS:GenSubst

2.1. Generalities for substitutions, type substitutions. Generally, a substitution is a list $((x_1 \, t_1) \dots (x_n \, t_n))$ of lists of length two, with distinct variables x_i and such that for each i , x_i is different from t_i . It is understood as simultaneous substitution. The default equality is **equal?**; however, in the versions ending with **-wrt** (for “with respect to”) one can provide special notions of equality. To construct substitutions we have

```
(make-substitution args vals)
(make-substitution-wrt arg-val-equal? args vals)
(make-subst arg val)
(make-subst-wrt arg-val-equal? arg val)
empty-subst
```

Accessing a substitution is done via the usual access operations for association list: **assoc** and **assoc-wrt**. We also provide

```
(restrict-substitution-wrt subst test?)
(restrict-substitution-to-args subst args)
```

```
(substitution-equal? subst1 subst2)
(substitution-equal-wrt? arg-equal? val-equal? subst1 subst2)
(subst-item-equal-wrt? arg-equal? val-equal? item1 item2)
(consistent-substitutions-wrt?
  arg-equal? val-equal? subst1 subst2)
```

Composition $\vartheta\sigma$ of two substitutions

$$\begin{aligned}\vartheta &= ((x_1 \ s_1) \dots (x_m \ s_m)), \\ \sigma &= ((y_1 \ t_1) \dots (y_n \ t_n))\end{aligned}$$

is defined as follows. In the list $((x_1 \ s_1\sigma) \dots (x_m \ s_m\sigma) (y_1 \ t_1) \dots (y_n \ t_n))$ remove all bindings $(x_i \ s_i\sigma)$ with $s_i\sigma = x_i$, and also all bindings $(y_j \ t_j)$ with $y_j \in \{x_1, \dots, x_n\}$. It is easy to see that composition is associative, with the empty substitution as unit. We provide

```
(compose-substitutions-wrt substitution-proc arg-equal?
  arg-val-equal? subst1 subst2)
```

We shall have occasion to use these general substitution procedures for the following kinds of substitutions

for	called	domain equality	arg-val-equality
type variables	tsubst	equal?	equal?
object variables	osubst	equal?	var-term-equal?
predicate variables	psubst	equal?	pvar-cterm-equal?
assumption variables	asubst	avar=?	avar-proof-equal?

The following substitutions will make sense for a

type	tsubst
term	tsubst and osubst
formula	tsubst and osubst and psubst
proof	tsubst and osubst and psubst and asubst

In particular, for *type substitutions* **tsubst** we have

```
(type-substitute type tsubst)
(type-subst type tvar type1)
(compose-t-substitutions tsubst1 tsubst2)
```

A display function for type substitutions is

```
(display-t-substitution tsubst)
```

2.2. Simultaneous free algebras as base types. We allow the formation of inductively generated types $\mu\vec{\alpha}\vec{\kappa}$, where $\vec{\alpha} = \alpha_1, \dots, \alpha_n$ is a list of distinct type variables, and $\vec{\kappa}$ is a list of “constructor types” whose argument types contain $\alpha_1, \dots, \alpha_n$ in strictly positive positions only.

For instance, $\mu\alpha(\alpha, \alpha \rightarrow \alpha)$ is the type of natural numbers; here the list $(\alpha, \alpha \rightarrow \alpha)$ stands for two generation principles: α for “there is a natural number” (the number 0), and $\alpha \rightarrow \alpha$ for “for every natural number there is another one” (its successor).

Let an infinite supply of *type variables* α, β be given. Let $\vec{\alpha} = (\alpha_j)_{j=1,\dots,m}$ be a list of distinct type variables. *Types* $\rho, \sigma, \tau, \mu, \nu \in \mathbf{Types}$ and *constructor types* $\kappa \in \mathbf{KT}(\vec{\alpha})$ are defined inductively as follows.

$$\frac{\vec{\rho}, \vec{\sigma}_1, \dots, \vec{\sigma}_n \in \mathbf{Types}}{\vec{\rho} \rightarrow (\vec{\sigma}_1 \rightarrow \alpha_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \alpha_{j_n}) \rightarrow \alpha_j \in \mathbf{KT}(\vec{\alpha})} \quad (n \geq 0)$$

$$\frac{\kappa_1, \dots, \kappa_n \in \mathbf{KT}(\vec{\alpha})}{(\mu \vec{\alpha} (\kappa_1, \dots, \kappa_n))_j \in \mathbf{Types}} \quad (n \geq 1, j = 1, \dots, m) \quad \frac{\rho, \sigma \in \mathbf{Types}}{\rho \rightarrow \sigma \in \mathbf{Types}}$$

Here $\vec{\rho}$ is short for a list ρ_1, \dots, ρ_k ($k \geq 0$) of types and $\vec{\rho} \rightarrow \sigma$ means $\rho_1 \rightarrow \dots \rightarrow \rho_k \rightarrow \sigma$, associated to the right. We shall use μ, ν for types of the form $(\mu \vec{\alpha} (\kappa_1, \dots, \kappa_n))_j$ only, and for types $\vec{\tau} = (\tau_j)_{j=1,\dots,m}$ and a constructor type $\kappa = \vec{\rho} \rightarrow (\vec{\sigma}_1 \rightarrow \alpha_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \alpha_{j_n}) \rightarrow \alpha_j \in \mathbf{KT}(\vec{\alpha})$ let

$$\kappa[\vec{\tau}] := \vec{\rho} \rightarrow (\vec{\sigma}_1 \rightarrow \tau_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \tau_{j_n}) \rightarrow \tau_j.$$

Examples.

<code>unit</code>	$:= \mu \alpha \alpha,$
<code>boole</code>	$:= \mu \alpha (\alpha, \alpha),$
<code>nat</code>	$:= \mu \alpha (\alpha, \alpha \rightarrow \alpha),$
<code>ytensor</code>	$(\alpha_1)(\alpha_2) := \mu \alpha. \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha,$
<code>ypair</code>	$(\alpha_1)(\alpha_2) := \mu \alpha. (\text{unit} \rightarrow \alpha_1) \rightarrow (\text{unit} \rightarrow \alpha_2) \rightarrow \text{unit} \rightarrow \alpha,$
<code>yplus</code>	$(\alpha_1)(\alpha_2) := \mu \alpha. (\alpha_1 \rightarrow \alpha, \alpha_2 \rightarrow \alpha),$
<code>list</code>	$(\alpha_1) := \mu \alpha (\alpha, \alpha_1 \rightarrow \alpha \rightarrow \alpha),$
<code>(tree, tlist)</code>	$:= \mu (\alpha, \beta) (\alpha, \beta \rightarrow \alpha, \beta, \alpha \rightarrow \beta \rightarrow \beta),$
<code>btree</code>	$:= \mu \alpha (\alpha, \alpha \rightarrow \alpha \rightarrow \alpha),$
<code>\mathcal{O}</code>	$:= \mu \alpha (\alpha, \alpha \rightarrow \alpha, (\text{nat} \rightarrow \alpha) \rightarrow \alpha),$
<code>\mathcal{T}_0</code>	$:= \text{nat},$
<code>\mathcal{T}_{n+1}</code>	$:= \mu \alpha (\alpha, (\mathcal{T}_n \rightarrow \alpha) \rightarrow \alpha).$

Note that we could have defined our primitive $\rho \times \sigma$ by $\mu \alpha. \rho \rightarrow \sigma \rightarrow \alpha$. However, this may lead to complex terms when it comes to extract programs from proofs. Therefore we stick to using $\rho \times \sigma$ as a primitive.

To add and remove names for type variables, we use

```
(add-tvar-name name1 ...)
(remove-tvar-name name1 ...)
```

We need a constructor, accessors and a test for type variables.

```
(make-tvar index name)  constructor
(tvar-to-index tvar)    accessor
(tvar-to-name tvar)     accessor
(tvar? x) .
```

To generate new type variables we use

```
(new-tvar type)
```


To introduce simultaneous free algebras we use

`add-algebras-with-parameters`, abbreviated `add-param-algs`.

An example is

```
(add-param-algs
 (list "labtree" "labtlist") 'alg-typeop 2
 '("LabLeaf" "alpha1=>labtree")
 '("LabBranch" "labtlist=>alpha2=>labtree")
 '("LabEmpty" "labtlist")
 '("LabTcons" "labtree=>labtlist=>labtlist" pairscheme-op))
```

This simultaneously introduces the two free algebras `labtree` and `labtlist`, both finitary, whose constructors are `LabLeaf`, `LabBranch`, `LabEmpty` and `LabTcons` (written as an infix pair operator, hence right associative). The constructors are introduced as “self-evaluating” constants; they play a special role in our semantics for normalization by evaluation.

In case there are no parameters we use `add-algs`, and in case there is no need for a simultaneous definition we use `add-alg` or `add-param-alg`.

For already introduced algebras we need constructors and accessors

```
(make-alg name type1 ...)
(alg-form-to-name alg)
(alg-form-to-types alg)
(alg-name-to-simalg-names alg-name)
(alg-name-to-token-types alg-name)
(alg-name-to-typed-constr-names alg-name)
(alg-name-to-tvars alg-name)
(alg-name-to-arity alg-name)
```

We also provide the tests

```
(alg-form? x)      incomplete test
(alg? x)           complete test
(finalg? type)     incomplete test
(ground-type? x)   incomplete test
```

We require that there is at least one nullary constructor in every free algebra; hence, it has a “canonical inhabitant”. For arbitrary types this need not be the case, but occasionally (e.g. for general logical problems, like to prove the drinker formula) it is useful. Therefore

```
(make-inhabited type term1 ...)
```

marks the optional term as the canonical inhabitant if it is provided, and otherwise creates a new constant of that type, which is taken to be the canonical inhabitant. We also have

```
(type-to-canonical-inhabitant type),
```

which returns the canonical inhabitant; it is an error to apply this procedure to a non-inhabited type. We do allow non-inhabited types to be able to implement some aspects of [7, 1]

To remove names for algebras we use

```
(remove-alg-name name1 ...)
```

Examples. Standard examples for finitary free algebras are the type `nat` of unary natural numbers, and the type `btree` of binary trees. The domain \mathcal{I}_{nat} of unary natural numbers is defined (as in [4]) as a solution to a domain equation.

We always provide the finitary free algebra `unit` consisting of exactly one element, and `bool` of booleans; objects of the latter type are (cf. loc. cit.) `true`, `false` and families of terms of this type, and in addition the bottom object of type `bool`.

Tests:

```
(arrow-form? type)
(star-form? type)
(object-type? type)
```

We also need constructors and accessors for arrow types

```
(make-arrow arg-type val-type)      constructor
(arrow-form-to-arg-type arrow-type)  accessor
(arrow-form-to-val-type arrow-type)  accessor
```

and star types

```
(make-star type1 type2)             constructor
(star-form-to-left-type star-type)   accessor
(star-form-to-right-type star-type)  accessor.
```

For convenience we also have

```
(mk-arrow type1 ... type)
(arrow-form-to-arg-types type <n>)  all (first n) argument types
(arrow-form-to-final-val-type type)  type of final value.
```

To check and to display a type we have

```
(type? x)
(type-to-string type).
```

Implementation. Type variables are implemented as lists:

```
(tvar index name).
```

3. VARIABLES

A variable of an object type is interpreted by a continuous functional (object) of that type. We use the word “variable” and not “program variable”, since continuous functionals are not necessarily computable. For readable in- and output, and also for ease in parsing, we may reserve certain strings as names for variables of a given type, e.g. `n,m` for variables of type `nat`. Then also `n0,n1,n2,...,m0,...` can be used for the same purpose.

In most cases we need to argue about existing (i.e. total) objects only. For the notion of totality we have to refer to [12, Chapter 8.3]; particularly

relevant here is exercise 8.5.7. To make formal arguments with quantifiers relativized to total objects more manageable, we use a special sort of variables intended to range over such objects only. For example, $n_0, n_1, n_2, \dots, m_0, \dots$ range over total natural numbers, and n^0, n^1, n^2, \dots are general variables. We say that the *degree of totality* for the former is 1, and for the latter 0.

n, m for variables of type `nat`), we use

```
(add-var-name name1 ... type)
(remove-var-name name1 ... type)
(default-var-name type).
```

The first variable name added for any given type becomes the default variable name. If the system creates new variables of this type, they will carry that name. For complex types it sometimes is necessary to talk about variables of a certain type without using a specific name. In this case one can use the empty string to create a so called numerated variable (see below). The parser is able to produce this kind of canonical variables from type expressions.

We need a constructor, accessors and tests for variables.

<code>(make-var type index t-deg name)</code>	constructor
<code>(var-to-type var)</code>	accessor
<code>(var-to-index var)</code>	accessor
<code>(var-to-t-deg var)</code>	accessor
<code>(var-to-name var)</code>	accessor
<code>(var-form? x)</code>	incomplete test
<code>(var? x).</code>	complete test

It is guaranteed that `equal?` is a valid test for equality of variables. Moreover, it is guaranteed that parsing a displayed variable reproduces the variable; the converse need not be the case (we may want to convert it into some canonical form).

For convenience we have the function

```
(mk-var type <index> <t-deg> <name>).
```

The type is a required argument; however, the remaining arguments are optional. The default for the name string is the value returned by

```
(default-var-name type)
```

If there is no default name, a numerated variable is created. The default for the totality is “total”.

Using the empty string as the name, we can create so called numerated variables. We further require that we can test whether a given variable belongs to those special ones, and that from every numerated variable we can compute its index:

```
(numerated-var? var)
(numerated-var-to-index numerated-var).
```

It is guaranteed that `make-var` used with the empty name string is a bijection

$$\text{Types} \times \mathbb{N} \times \text{TDegs} \rightarrow \text{NumVars}$$

with inverses `var-to-type`, `numerated-var-to-index` and `var-to-t-deg`.

Although these functions look like an ad hoc extension of the interface that is convenient for normalization by evaluation, there is also a deeper background: these functions can be seen as the “computational content” of the well-known phrase “we assume that there are infinitely many variables of every type”. Giving a constructive proof for this statement would require to give infinitely many examples of variables for every type. This of course can only be done by specifying a function (for every type) that enumerates these examples. To make the specification finite we require the examples to be given in a uniform way, i.e. by a function of two arguments. To make sure that all these examples are in fact different, we would have to require `make-var` to be injective. Instead, we require (classically equivalent) `make-var` to be a bijection on its image, as again, this can be turned into a computational statement by requiring that a witness (i.e. an inverse function) is given.

Finally, as often the exact knowledge of infinitely many variables of every type is not needed we require that, either by using the above functions or by some other form of definition, functions

`(type-to-new-var type)`

`(type-to-new-partial-var type)`

are defined that return a (total or partial) variable of the requested type, different from all variables that have ever been returned by any of the specified functions so far.

Occasionally we may want to create a new variable with the same name (and degree of totality) as a given one. This is useful e.g. for bound renaming. Therefore we supply

`(var-to-new-var var).`

Implementation. Variables are implemented as lists:

`(var type index t-deg name).`

4. CONSTANTS

Every constant (or more precisely, object constant) has a type and denotes a computable (hence continuous) functional of that type. We have the following three kinds of constants:

- constructors, kind `constr`,
- constants with user defined rules (also called program(mable) constant, or `pconst`), kind `pconst`,
- constants whose rules are fixed, kind `fixed-rules`.

The latter are built into the system: recursion operators for arbitrary algebras, equality and existence operators for finitary algebras, and existence elimination. They are typed in parametrized form, with the actual type (or formula) given by a type (or type and formula) substitution that is also part of the constant. For instance, equality is typed by $\alpha \rightarrow \alpha \rightarrow \text{boole}$ and a

type substitution $\alpha \mapsto \rho$. This is done for clarity (and brevity, e.g. for large ρ in the example above), since one should think of the type of a constant in this way.

For constructors and for constants with fixed rules, by efficiency reasons we want to keep the object denoted by the constant (as needed for normalization by evaluation) as part of it. It depends on the type of the constant, hence must be updated in a given proof whenever the type changes by a type substitution.

SS:RewCompRules

4.1. Rewrite and computation rules for program constants. For every program constant c^ρ we assume that some rewrite rules of the form $c\vec{K} \mapsto N$ are given, where $\text{FV}(N) \subseteq \text{FV}(\vec{K})$ and $c\vec{K}, N$ have the same type (not necessarily a ground type). Moreover, for any two rules $c\vec{K} \mapsto N$ and $c\vec{K}' \mapsto N'$ we require that \vec{K} and \vec{K}' are of the same length, called the *arity* of c . The rules are divided into *computation rules* and proper *rewrite rules*. They must satisfy the requirements listed in [4]. The idea is that a computation rule can be understood as a description of a computation in a suitable *semantical* model, provided the syntactic constructors correspond to semantic ones in the model, whereas the other rules describe *syntactic* transformations.

There a more general approach was used: one may enter into components of products. Then instead of one arity one needs several “type informations” $\vec{\rho} \rightarrow \sigma$ with $\vec{\rho}$ a list of types, 0’s and 1’s indicating the left or right part of a product type. For example, if c is of type $\tau \rightarrow (\tau \rightarrow \tau \rightarrow \tau) \times (\tau \rightarrow \tau)$, then the rules $cy0xx \mapsto a$ and $cy1 \mapsto b$ are admitted, and c comes with the type informations $(\tau, 0, \tau, \tau \rightarrow \tau) \rightarrow \tau$ and $(\tau, 1) \rightarrow (\tau \rightarrow \tau)$. – However, for simplicity we only deal with a single arity here.

Given a set of rewrite rules, we want to treat some rules - which we call *computation rules* - in a different, more efficient way. The idea is that a computation rule can be understood as a description of a computation in a suitable *semantical model*, provided the syntactic constructors correspond to semantic ones in the model, whereas the other rules describe *syntactic* transformations.

In order to define what we mean by computation rules, we need the notion of a *constructor pattern*. These are special terms defined inductively as follows.

- Every variable is a constructor pattern.
- If c is a constructor and P_1, \dots, P_n are constructor patterns (or projection markers 0 or 1), such that $c\vec{P}$ is of ground type, then $c\vec{P}$ is a constructor pattern.

From the given set of rewrite rules we choose a subset COMP with the following properties.

- If $c\vec{P} \mapsto Q \in \text{COMP}$, then P_1, \dots, P_n are constructor patterns or projection markers.
- The rules are left-linear, i.e. if $c\vec{P} \mapsto Q \in \text{COMP}$, then every variable in $c\vec{P}$ occurs only once in $c\vec{P}$.
- The rules are non-overlapping, i.e. for different rules $c\vec{K} \mapsto M$ and $c\vec{L} \mapsto N$ in COMP the left hand sides $c\vec{K}$ and $c\vec{L}$ are non-unifiable.

We write $c\vec{M} \mapsto_{\text{comp}} Q$ to indicate that the rule is in COMP. All other rules will be called (proper) rewrite rules, written $c\vec{M} \mapsto_{\text{rew}} K$.

In our reduction strategy computation rules will always be applied first, and since they are non-overlapping, this part of the reduction is unique. However, since we allowed almost arbitrary rewrite rules, it may happen that in case no computation rule applies a term may be rewritten by different rules \notin COMP. In order to obtain a deterministic procedure we then select the first applicable rewrite rule (This is a slight simplification of [4], where special functions sel_c were used for this purpose).

SS:RecSFA

4.2. Recursion over simultaneous free algebras. We now explain what we mean by recursion over simultaneous free algebras. The inductive structure of the types $\vec{\mu} = \mu \vec{\alpha} \vec{\kappa}$ corresponds to two sorts of constants. With the *constructors* $\text{constr}_i^{\vec{\mu}}: \kappa_i[\vec{\mu}]$ we can construct elements of a type μ_j , and with the *recursion operators* $\mathcal{R}_{\mu_j}^{\vec{\mu}, \vec{\tau}}$ we can construct mappings from μ_j to τ_j by recursion on the structure of $\vec{\mu}$. So in (Rec arrow-types), **arrow-types** is a list $\mu_1 \rightarrow \tau_1, \dots, \mu_k \rightarrow \tau_k$. Here μ_1, \dots, μ_k are the algebras defined simultaneously and τ_1, \dots, τ_k are the result types.

For convenience in our later treatment of proofs (when we want to normalize a proof by (1) translating it into a term, (2) normalizing this term and (3) translating the normal term back into a proof), we also allow all formulas $\forall x_1^{\mu_1} A_1, \dots, \forall x_k^{\mu_k} A_k$ instead of **arrow-types**: they are treated as $\mu_1 \rightarrow \tau(A_1), \dots, \mu_k \rightarrow \tau(A_k)$ with $\tau(A_j)$ the type of A_j .

Recall the definition of types and constructor types in Section 2, and the examples given there. In order to define the type of the recursion operators w.r.t. $\vec{\mu} = \mu \vec{\alpha} \vec{\kappa}$ and result types $\vec{\tau}$, we first define for

$$\kappa_i = \vec{\rho} \rightarrow (\vec{\sigma}_1 \rightarrow \alpha_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \alpha_{j_n}) \rightarrow \alpha_j \in \text{KT}(\vec{\alpha})$$

the *step type*

$$\delta_i^{\vec{\mu}, \vec{\tau}} := \vec{\rho} \rightarrow (\vec{\sigma}_1 \rightarrow \mu_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \mu_{j_n}) \rightarrow (\vec{\sigma}_1 \rightarrow \tau_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \tau_{j_n}) \rightarrow \tau_j.$$

Here $\vec{\rho}, (\vec{\sigma}_1 \rightarrow \mu_{j_1}), \dots, (\vec{\sigma}_n \rightarrow \mu_{j_n})$ correspond to the *components* of the object of type μ_j under consideration, and $(\vec{\sigma}_1 \rightarrow \tau_{j_1}), \dots, (\vec{\sigma}_n \rightarrow \tau_{j_n})$ to the previously defined values. The recursion operator $\mathcal{R}_{\mu_j}^{\vec{\mu}, \vec{\tau}}$ has type

$$\mathcal{R}_{\mu_j}^{\vec{\mu}, \vec{\tau}}: \delta_1^{\vec{\mu}, \vec{\tau}} \rightarrow \dots \rightarrow \delta_k^{\vec{\mu}, \vec{\tau}} \rightarrow \mu_j \rightarrow \tau_j.$$

We will often write $\mathcal{R}_j^{\vec{\mu}, \vec{\tau}}$ for $\mathcal{R}_{\mu_j}^{\vec{\mu}, \vec{\tau}}$, and omit the upper indices $\vec{\mu}, \vec{\tau}$ when they are clear from the context. In case of a non-simultaneous free algebra, i.e. of type $\mu \alpha(\kappa)$, for $\mathcal{R}_\mu^{\mu, \tau}$ we normally write \mathcal{R}_μ^τ .

A simple example for simultaneous free algebras is

$$(\text{tree}, \text{tlist}) := \mu(\alpha, \beta) (\alpha, \beta \rightarrow \alpha, \beta, \alpha \rightarrow \beta \rightarrow \beta).$$

The constructors are

$$\begin{aligned} \text{Leaf}^{\text{tree}} &:= \text{constr}_1^{(\text{tree}, \text{tlist})}, \\ \text{Branch}^{\text{tlist} \rightarrow \text{tree}} &:= \text{constr}_2^{(\text{tree}, \text{tlist})}, \\ \text{Empty}^{\text{tlist}} &:= \text{constr}_3^{(\text{tree}, \text{tlist})}, \end{aligned}$$

$$\mathsf{Tcons}^{\mathsf{tree} \rightarrow \mathsf{tlist} \rightarrow \mathsf{tlist}} := \mathsf{constr}_4^{(\mathsf{tree}, \mathsf{tlist})}.$$

An example for a recursion constant is

$$\begin{aligned} &(\mathsf{const} \ \mathsf{Rec} \ \delta_1 \rightarrow \delta_2 \rightarrow \delta_3 \rightarrow \delta_4 \rightarrow \mathsf{tree} \rightarrow \alpha_1 \\ &\quad (\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2)) \end{aligned}$$

with

$$\begin{aligned} \delta_1 &:= \alpha_1, \\ \delta_2 &:= \mathsf{tlist} \rightarrow \alpha_2 \rightarrow \alpha_1, \\ \delta_3 &:= \alpha_2, \\ \delta_4 &:= \mathsf{tree} \rightarrow \mathsf{tlist} \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_2. \end{aligned}$$

Here the fact that we deal with a simultaneous recursion (over tree and tlist), and that we define a constant of type $\mathsf{tree} \rightarrow \dots$, can all be inferred from what is given: the type $\mathsf{tree} \rightarrow \dots$ is right there, and for tlist we can look up the simultaneously defined algebras.

For the external representation (i.e. display) we use the shorter notation

$$(\mathsf{Rec} \ \mathsf{tree} \rightarrow \tau_1 \ \mathsf{tlist} \rightarrow \tau_2).$$

As already mentioned, it is also possible that the object constant Rec comes with formulas instead of types, as the assumption constant Ind below. This is due to our desire not to duplicate code when normalizing proofs, but rather translate the proof into a term first, normalize the term and then translate back into a proof. For the last step we must have the original formulas of the induction operator available.

To see a concrete example, let us recursively define addition $+$: $\mathsf{tree} \rightarrow \mathsf{tree} \rightarrow \mathsf{tree}$ and \oplus : $\mathsf{tlist} \rightarrow \mathsf{tree} \rightarrow \mathsf{tlist}$. The recursion equations to be satisfied are

$$\begin{aligned} + \ \mathsf{Leaf} &= \lambda a a, \\ + \ (\mathsf{Branch} \ bs) &= \lambda a. \mathsf{Branch}(\oplus \ bs \ a), \\ \oplus \ \mathsf{Empty} &= \lambda a \ \mathsf{Empty}, \\ \oplus \ (\mathsf{Tcons} \ b \ bs) &= \lambda a. \mathsf{Tcons}(+ \ b \ a)(\oplus \ bs \ a). \end{aligned}$$

We define $+$ and \oplus by means of the recursion operators $\mathcal{R}_{\mathsf{tree}}$ and $\mathcal{R}_{\mathsf{tlist}}$ with result types

$$\begin{aligned} \tau_1 &:= \mathsf{tree} \rightarrow \mathsf{tree}, \\ \tau_2 &:= \mathsf{tree} \rightarrow \mathsf{tlist}. \end{aligned}$$

The step terms are

$$\begin{aligned} M_1 &:= \lambda a a, \\ M_2 &:= \lambda bs \lambda g^{\tau_2} \lambda a. \mathsf{Branch}(g \ a), \\ M_3 &:= \lambda a \ \mathsf{Empty}, \\ M_4 &:= \lambda b \lambda bs \lambda f^{\tau_1} \lambda g^{\tau_2} \lambda a. \mathsf{Tcons}(f \ a)(g \ a). \end{aligned}$$

Then

$$+ := \mathcal{R}_{\mathsf{tree}} \vec{M} : \mathsf{tree} \rightarrow \mathsf{tree} \rightarrow \mathsf{tree},$$

$$\oplus := \mathcal{R}_{\text{tlist}} \vec{M} : \text{tlist} \rightarrow \text{tree} \rightarrow \text{tlist}.$$

To explain the *conversion relation*, it will be useful to employ the following notation. Let $\vec{\mu} = \mu \vec{\alpha} \vec{\kappa}$,

$$\kappa_i = \rho_1 \rightarrow \dots \rightarrow \rho_m \rightarrow (\vec{\sigma}_1 \rightarrow \alpha_{j_1}) \rightarrow \dots \rightarrow (\vec{\sigma}_n \rightarrow \alpha_{j_n}) \rightarrow \alpha_j \in \text{KT}(\vec{\alpha}),$$

and consider $\text{constr}_i^{\vec{\mu}} \vec{N}$. Then we write $\vec{N}^P = N_1^P, \dots, N_m^P$ for the *parameter arguments* $N_1^{\rho_1}, \dots, N_m^{\rho_m}$ and $\vec{N}^R = N_1^R, \dots, N_n^R$ for the *recursive arguments* $N_{m+1}^{\vec{\sigma}_1 \rightarrow \mu_{j_1}}, \dots, N_{m+n}^{\vec{\sigma}_n \rightarrow \mu_{j_n}}$, and n^R for the number n of recursive arguments.

We define a *conversion relation* \mapsto_ρ between terms of type ρ by

- (1) $(\lambda x M) N \mapsto M[x := N]$
- (2) $\lambda x. M x \mapsto M$ if $x \notin \text{FV}(M)$, M not an abstraction
- (3) $(\mathcal{R}_j^{\vec{\mu}, \vec{\tau}} \vec{M})^{\mu_j \rightarrow \tau_j} (\text{constr}_i^{\vec{\mu}} \vec{N}) \mapsto M_i \vec{N} ((\mathcal{R}_{j_1}^{\vec{\mu}, \vec{\tau}} \vec{M}) \circ N_1^R) \dots ((\mathcal{R}_{j_n}^{\vec{\mu}, \vec{\tau}} \vec{M}) \circ N_n^R)$

Here we have written $\mathcal{R}_j^{\vec{\mu}, \vec{\tau}}$ for $\mathcal{R}_{\mu_j}^{\vec{\mu}, \vec{\tau}}$, and \circ means composition.

4.3. Internal representation of constants. Every object constant has the internal representation

$$(\text{const } \textit{object-or-arity name uninstant-type-or-formula subst} \\ \textit{t-deg token-type arrow-types-or-repro-formulas}),$$

where *subst* may have type, object and assumption variables in its domain. The type of the constant is the result of carrying out this substitution in *uninstant-type-or-formula* (if this is a type; otherwise first substitute and then convert the formula into a type); free type variables may again occur in this type. Note that a formula will occur if *name* is **Ex-Intro** or **Ex-Elim**, and may occur if it is **Rec**. Examples for object constants are

$$\begin{aligned} &(\text{const } \text{Compose } (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma \ (\alpha \mapsto \rho, \beta \mapsto \sigma, \gamma \mapsto \tau) \ \dots) \\ &(\text{const } \text{Eq } \alpha \rightarrow \alpha \rightarrow \text{boole } (\alpha \mapsto \text{finalg}) \ \dots) \\ &(\text{const } \text{E } \alpha \rightarrow \text{boole } (\alpha \mapsto \text{finalg} \dots)) \\ &(\text{const } \text{Ex-Elim } \exists x^\alpha P(x) \rightarrow (\forall x^\alpha. P(x) \rightarrow Q) \rightarrow Q \\ &\quad (\alpha \mapsto \tau, P^{(\alpha)} \mapsto \{z^\tau \mid A\}, Q \mapsto \{ \mid B \}) \ \dots) \end{aligned}$$

object-or-arity is an object if this object cannot be changed, e.g. by allowing user defined rules for the constant; otherwise, the associated object needs to be updated whenever a new rule is added, and we have the arity of those rules instead. The rules are of crucial importance for the correctness of a proof, and should not be invisibly buried in the denoted object taken as part of the constant (hence of any term involving it). Therefore we keep the rules of a program constant and also its denoted objects (depending on type substitutions) at a central place, a global variable **PROGRAM-CONSTANTS** which assigns to every name of such a constant the constant itself (with uninstantiated type), the rules presently chosen for it and also its denoted objects (as association list with type substitutions as keys). When a new rule has been added, the new objects for the program constant are computed, and the new list to be associated with the program constant is written in

PROGRAM-CONSTANTS instead. All information on a program constant except its denoted object and its computation and rewrite rules (i.e. its type, degree of totality, arity and token type) is stable and hence can be kept as part of it. The *token type* can be either `const` (i.e. constant written as application) or one of: `postfix-op`, `prefix-op`, `binding-op`, `add-op`, `mul-op`, `rel-op`, `and-op`, `or-op`, `imp-op` and `pair-op`.

Constructor, accessors and tests for all kinds of constants:

```
(make-const obj-or-arity name kind uninst-type tsubst
  t-deg token-type . arrow-types-or-repro-formulas)
(const-to-object-or-arity const)
(const-to-name const)
(const-to-kind const)
(const-to-uninst-type const)
(const-to-tsubst const)
(const-to-t-deg const)
(const-to-token-type const)
(const-to-arrow-types-or-repro-formulas const)
(const? x)
(const=? x y)
```

The type substitution *tsubst* must be restricted to the type variables in `uninst-type`. `arrow-types-or-repro-formulas` are only present for the `Rec` constants. They are needed for the reproduction case.

From these we can define

```
(const-to-type const)
(const-to-tvars const)
```

A *constructor* is a special constant with no rules. We maintain an association list `CONSTRUCTORS` assigning to every name of a constructor an association list associating with every type substitution (restricted to the type parameters) the corresponding instance of the constructor. We provide

```
(constr-name? string)
(constr-name-to-constr name <tsubst>)
(constr-name-and-tsubst-to-constr name tsubst),
```

where in `(constr-name-to-constr name <tsubst>)`, *name* is a string or else of the form `(Ex-Intro formula)`. If the optional *tsubst* is not present, the empty substitution is used.

For given algebras one can display the associated constructors with their types by calling

```
(display-constructors alg-name1 ...).
```

We also need procedures recovering information from the string denoting a program constant (via `PROGRAM-CONSTANTS`):

```
(pconst-name-to-pconst name)
```

```
(pconst-name-to-comprules name)
(pconst-name-to-rewrules name)
(pconst-name-to-inst-objs name)
(pconst-name-and-tsubst-to-object name tsubst)
(pconst-name-to-object name).
```

One can display the program constants together with their current computation and rewrite rules by calling

```
(display-program-constants name1 ...).
```

To add and remove program constants we use

```
(add-program-constant name type <rest>)
(remove-program-constant symbol);
```

rest consists of an initial segment of the following list: **t-deg** (default 0), **token-type** (default **const**) and **arity** (default maximal number of argument types).

To add and remove computation and rewrite rules we have

```
(add-computation-rule lhs rhs)
(add-rewrite-rule lhs rhs)
(remove-computation-rules-for lhs)
(remove-rewrite-rules-for lhs).
```

To generate our constants with fixed rules we use

```
(finalg-to--const finalg)                equality
(finalg-to-e-const finalg)                existence
(arrow-types-to-rec-const . arrow-types)  recursion
(ex-formula-and-concl-to-ex-elim-const
  ex-formula concl)
```

Similarly to **arrow-types-to-rec-const** we also define the procedure **all-formulas-to-rec-const**. It will be used in to achieve normalization of proofs via translating them in terms.

[Noch einfügen: **arrow-types-to-cases-const** und zur Behandlung von Beweisen **all-formulas-to-cases-const**]

5. PREDICATE VARIABLES AND CONSTANTS

5.1. Predicate variables. A predicate variable of arity ρ_1, \dots, ρ_n is a placeholder for a formula A with distinguished (different) variables x_1, \dots, x_n of types ρ_1, \dots, ρ_n . Such an entity is called a *comprehension term*, written $\{x_1, \dots, x_n \mid A\}$.

Predicate variable names are provided in the form of an association list, which assigns to the names their arities. By default we have the predicate variable **bot** of arity (**arity**), called (logical) falsity. It is viewed as a predicate variable rather than a predicate constant, since (when translating a classical proof into a constructive one) we want to substitute for **bot**.

Often we will argue about *Harrop formulas* only, i.e. formulas without computational content. For convenience we use a special sort of predicate variables intended to range over comprehension terms with Harrop formulas only. For example, $P_0, P_1, P_2, \dots, Q_0, \dots$ range over comprehension terms with Harrop formulas, and P^0, P^1, P^2, \dots are general predicate variables. We say that *Harrop degree* for the former is 1, and for the latter 0.

We need constructors and accessors for arities

```
(make-arity type1 ...)
(arity-to-types arity)
```

To display an arity we have

```
(arity-to-string arity)
```

We can test whether a string is a name for a predicate variable, and if so compute its associated arity:

```
(pvar-name? string)
(pvar-name-to-arity pvar-name)
```

To add and remove names for predicate variables of a given arity (e.g. Q for predicate variables of arity `nat`), we use

```
(add-pvar-name name1 ... arity)
(remove-pvar-name name1 ...)
```

We need a constructor, accessors and tests for predicate variables.

```
(make-pvar arity index h-deg name)  constructor
(pvar-to-arity pvar)                accessor
(pvar-to-index pvar)                accessor
(pvar-to-h-deg pvar)                accessor
(pvar-to-name pvar)                 accessor
(pvar? x)
(equal-pvars? pvar1 pvar2)
```

For convenience we have the function

```
(mk-pvar arity <index> <h-deg> <name>)
```

The arity is a required argument; the remaining arguments are optional. The default for *index* is -1 , for *h-deg* is 1 (i.e. Harrop-formula) and for *name* it is given by `(default-pvar-name arity)`.

It is guaranteed that parsing a displayed predicate variable reproduces the predicate variable; the converse need not be the case (we may want to convert it into some canonical form).

5.2. Predicate constants. We also allow *predicate constants*. The general reason for having them is that we need predicates to be axiomatized, e.g. `Equal` and `Total` (which are *not* placeholders for formulas). Prime formulas built from predicate constants do not give rise to extracted terms, and cannot be substituted for.

Notice that a predicate constant does not change its name under a type substitution; this is in contrast to predicate (and other) variables. Notice also that the parser can infer from the arguments the types $\rho_1 \dots \rho_n$ to be substituted for the type variables in the uninstantiated arity of P .

To add and remove names for predicate constants of a given arity, we use

```
(add-predconst-name name1 ... arity)
(remove-predconst-name name1 ...)
```

We need a constructor, accessors and tests for predicate constants.

```
(make-predconst uninst-arity tsubst index name)  constructor
(predconst-to-uninst-arity predconst)              accessor
(predconst-to-tsubst predconst)                    accessor
(predconst-to-index predconst)                     accessor
(predconst-to-name predconst)                      accessor
(predconst? x)
```

Moreover we need

```
(predconst-name? name)
(predconst-name-to-arity predconst-name).
(predconst-to-string predconst).
```

SS:IDPredConsts

5.3. Inductively defined predicate constants. As we have seen, type variables allow for a general treatment of inductively generated types $\mu\vec{\alpha}\vec{\kappa}$. Similarly, we can use predicate variables to inductively generate predicates $\mu\vec{X}\vec{K}$.

More precisely, we allow the formation of inductively generated predicates $\mu\vec{X}\vec{K}$, where $\vec{X} = (X_j)_{j=1,\dots,N}$ is a list of distinct predicate variables, and $\vec{K} = (K_i)_{i=1,\dots,k}$ is a list of constructor formulas (or “clauses”) containing X_1, \dots, X_N in strictly positive positions only.

To introduce inductively defined predicates we use

```
add-ids.
```

An example is

```
(add-ids (list (list "Ev" (make-arity (py "nat"))) "algEv")
           (list "Od" (make-arity (py "nat"))) "algOd"))
'("Ev 0" "InitEv")
'("allnc n.Od n -> Ev(n+1)" "GenEv")
'("Od 1" "InitOd")
'("allnc n.Ev n -> Od(n+1)" "GenOd"))
```

This simultaneously introduces the two inductively defined predicate constants `Ev` and `Od`, by the clauses given. The presence of an algebra name after the arity (here `algEv` and `algOd`) indicates that this inductively defined predicate constant is to have computational content. Then all clauses with this constant in the conclusion must provide a constructor name (here `InitEv`, `GenEv`, `InitOd`, `GenOd`). If the constant is to have no computational content, then all its clauses must be invariant (under realizability, a.k.a. “negative”).

Here are some further examples of inductively defined predicates:

```
(add-ids
  (list (list "Even" (make-arity (py "nat")) "algEven"))
  '("Even 0" "InitEven")
  '("allnc n.Even n -> Even(n+2)" "GenEven"))

(add-ids
  (list (list "Acc" (make-arity (py "nat")) "algAcc"))
  '("allnc n.(all m.m<n -> Acc m) -> Acc n" "GenAccSup"))

(add-ids (list (list "OrID" (make-arity) "algOrID"))
  '("P^1 -> OrID" "InlOrID")
  '("P^2 -> OrID" "InrOrID"))

(add-ids
  (list (list "EqID" (make-arity (py "alpha") (py "alpha"))
            "algEqID"))
  '("allnc x^ EqID x^ x^" "GenEqID"))

(add-ids (list (list "ExID" (make-arity) "algExID"))
  '("allnc x^.Q^ x^ -> ExID" "GenExID"))

(add-ids
  (list (list "FalsityID" (make-arity) "algFalsityID")))
```

6. TERMS AND OBJECTS

Terms

Terms are built from (typed) variables and constants by abstraction, application, pairing, formation of left and right components (i.e. projections) and the *if*-construct.

The *if*-construct distinguishes cases according to the outer constructor form; the simplest example (for the type `boole`) is *if-then-else*. Here we do not want to evaluate all arguments right away, but rather evaluate the test argument first and depending on the result evaluate at most one of the other arguments. This phenomenon is well known in functional languages; e.g. in SCHEME the *if*-construct is called a *special form* as opposed to an operator. In accordance with this terminology we also call our *if*-construct a special form. It will be given a special treatment in `nbe-term-to-object`.

Usually it will be the case that every closed term of an sfa ground type reduces via the computation rules to a constructor term, i.e. a closed term built from constructors only. However, we do not require this.

We have constructors, accessors and tests for variables

<code>(make-term-in-var-form var)</code>	constructor
<code>(term-in-var-form-to-var term)</code>	accessor,
<code>(term-in-var-form? term)</code>	test,

for constants

<code>(make-term-in-const-form const)</code>	constructor
--	-------------

(term-in-const-form-to-const *term*) accessor
 (term-in-const-form? *term*) test,

for abstractions

(make-term-in-abst-form *var term*) constructor
 (term-in-abst-form-to-var *term*) accessor
 (term-in-abst-form-to-kernel *term*) accessor
 (term-in-abst-form? *term*) test,

for applications

(make-term-in-app-form *term1 term2*) constructor
 (term-in-app-form-to-op *term*) accessor
 (term-in-app-form-to-arg *term*) accessor
 (term-in-app-form? *term*) test,

for pairs

(make-term-in-pair-form *term1 term2*) constructor
 (term-in-pair-form-to-left *term*) accessor
 (term-in-pair-form-to-right *term*) accessor
 (term-in-pair-form? *term*) test,

for the left and right component of a pair

(make-term-in-lcomp-form *term*) constructor
 (make-term-in-rcomp-form *term*) constructor
 (term-in-lcomp-form-to-kernel *term*) accessor
 (term-in-rcomp-form-to-kernel *term*) accessor
 (term-in-lcomp-form? *term*) test
 (term-in-rcomp-form? *term*) test

and for if-constructs

(make-term-in-if-form *test alts . rest*) constructor
 (term-in-if-form-to-test *term*) accessor
 (term-in-if-form-to-alts *term*) accessor
 (term-in-if-form-to-rest *term*) accessor
 (term-in-if-form? *term*) test.

where in `make-term-in-if-form`, *rest* is either empty or an all-formula.

It is convenient to have more general application constructors and accessors available, where application takes arbitrary many arguments and works for ordinary application as well as for component formation.

(mk-term-in-app-form *term term1 ...*) constructor
 (term-in-app-form-to-final-op *term*) accessor
 (term-in-app-form-to-args *term*) accessor,

Also for abstraction it is convenient to have a more general constructor taking arbitrary many variables to be abstracted one after the other

```
(mk-term-in-abst-form var1 ... term).
```

We also allow vector notation for recursion (cf. Joachimski and Matthes [8]). Moreover we need

```
(term? x)
(term=? term1 term2)
(terms=? terms1 terms2)
(term-to-type term)
(term-to-free term)
(term-to-bound term)
(term-to-t-deg term)
(synt-total? term)
(term-to-string term).
```

6.1. Normalization. We need an operation which transforms a term into its normal form w.r.t. the given computation and rewrite rules. Here we base our treatment on *normalization by evaluation* introduced in [5], and extended to arbitrary computation and rewrite rules in [4].

For normalization by evaluation we need semantical *objects*. For an arbitrary ground type every term family of that type is an object. For an sfa ground type, in addition the constructors have semantical counterparts. The freeness of the constructors is expressed by requiring that their ranges are disjoint and that they are injective. Moreover, we view the free algebra as a domain and require that its bottom element is not in the range of the constructors. Hence the constructors are total and non-strict. Then by applying `nbe-reflect` followed by `nbe-reify` we can normalize every term, where normalization refers to the computation as well as the rewrite rules.

An object consists of a semantical value and a type.

```
(nbe-make-object type value)  constructor
(nbe-object-to-type object)   accessor
(nbe-object-to-value object)  accessor
(nbe-object? x)               test.
```

To work with objects, we need

```
(nbe-object-apply function-obj arg-obj)
```

Again it is convenient to have a more general application operation available, which takes arbitrary many arguments and works for ordinary application as well as for component formation. We also need an operation composing two unary function objects.

```
(nbe-object-app function-obj arg-obj1 ...)
(nbe-object-compose function-obj1 function-obj2)
```

For ground type values we need constructors, accessors and tests. To make constructors “self-evaluating”, a constructor value has the form

(constr-value *name objs delayed-constr*),

where *delayed-constr* is a procedure of zero arguments which evaluates to this very same constructor. This is necessary to avoid having a cycle (for nullary constructors, and only for those).

(nbe-make-constr-value <i>name objs</i>)	constructor
(nbe-constr-value-to-name <i>value</i>)	accessor
(nbe-constr-value-to-args <i>value</i>)	accessor
(nbe-constr-value-to-constr <i>value</i>)	accessor
(nbe-constr-value? <i>value</i>)	test
(nbe-fam-value? <i>value</i>)	test.

The essential function which “animates” the program constants according to the given computation and rewrite rules is

(nbe-pconst-and-tsubst-and-rules-to-object
 pconst tsubst comprules rewrules)

Using it we can define an *evaluation* function, which assigns to a term and an environment a semantical object:

(nbe-term-to-object *term bindings*) evaluation.

Here *bindings* is an association list assigning objects of the same type to variables. In case a variable is not assigned anything in *bindings*, by default we assign the constant term family of this variable, which always is an object of the correct type.

The interpretation of the program constants requires some auxiliary functions (cf. [4]):

(nbe-constructor-pattern? <i>term</i>)	test
(nbe-inst? <i>constr-pattern obj</i>)	test
(nbe-genargs <i>constr-pattern obj</i>)	generalized arguments
(nbe-extract <i>termfam</i>)	extracts a term from a family
(nbe-match <i>pattern term</i>)	

Then we can define

(nbe-reify <i>object</i>)	reification
(nbe-reflect <i>term</i>)	reflection

and by means of these

(nbe-normalize-term *term*) normalization,

abbreviated **nt**.

The **if**-form needs a special treatment. In particular, for a full normalization of terms (including permutative conversions), we define a preprocessing step that η expands the alternatives of all **if**-terms. The result contains **if**-terms with ground type alternatives only.

6.2. Substitution. Recall the generalities on substitutions in Section 2.1.

We define simultaneous substitution for type and object variables in a term, via `tsubst` and `subst`. It is assumed that `subst` only affects those vars whose type is not changed by `tsubst`.

In the abstraction case of the recursive definition, the abstracted variable may need to be renamed. However, its type can be affected by `tsubst`. Then the renaming cannot be made part of `subst`, because the condition above would be violated. Therefore we carry along a procedure renaming variables, which remembers the renaming of variables done so far.

```
(term-substitute term tosubst)
(term-subst term arg val)
(compose-o-substitutions subst1 subst2)
```

The `o` in `compose-o-substitutions` indicates that we substitute for *object* variables. However, since this is the most common form of substitution, we also write `compose-substitutions` instead.

Display functions for substitutions are

```
(display-substitution subst)
(substitution-to-string subst)
```

7. FORMULAS AND COMPREHENSION TERMS

S:Formulas

A prime formula can have the form

- `(atom r)` with a term `r` of type `boole`,
- `(predicate a r1 ... rn)` with a predicate variable or constant `a` and terms `r1 ... rn`.

Formulas are built from prime formulas by

- implication `(imp formula1 formula2)`
- conjunction `(and formula1 formula2)`
- tensor `(tensor formula1 formula2)`
- all quantification `(all x formula)`
- existential quantification `(ex x formula)`
- all quantification `(allnc x formula)` without computational content
- existential quantification `(exnc x formula)` without computational content

Moreover we have classical existential quantification in an arithmetical and a logical form:

```
(exca (x1 ...) formula)  arithmetical version
(excl (x1 ...) formula)  logical version.
```

Here we allow that the quantified variable is formed without \sim , i.e. ranges over total objects only.

Formulas can be *unfolded* in the sense that the all classical existential quantifiers are replaced according to their definition. Inversely a formula can be *folded* in the sense that classical existential quantifiers are introduced wherever possible.

Comprehension terms have the form `(cterm vars formula)`. Note that *formula* may contain further free variables.

Tests:

```
(atom-form? formula)
(predicate-form? formula)
(prime-form? formula)
(imp-form? formula)
(and-form? formula)
(tensor-form? formula)
(all-form? formula)
(ex-form? formula)
(allnc-form? formula)
(exnc-form? formula)
(exca-form? formula)
(excl-form? formula)
```

and also

```
(quant-prime-form? formula)
(quant-free? formula).
```

We need constructors and accessors for prime formulas

```
(make-atomic-formula boolean-term)
(make-predicate-formula predicate term1 ...)
atom-form-to-kernel
predicate-form-to-predicate
predicate-form-to-args.
```

We also have constructors for special atomic formulas

```
(make-eq term1 term2)  constructor for equalities
(make-= term1 term2)  constructor for equalities on finalgs
(make-total term)      constructor for totalities
(make-e term)          constructor for existence on finalgs
truth
falsity
falsity-log.
```

We need constructors and accessors for implications

```
(make-imp premise conclusion)  constructor
(imp-form-to-premise imp-formula)  accessor
(imp-form-to-conclusion imp-formula)  accessor,
```

conjunctions

```
(make-and formula1 formula2)  constructor
```

`(and-form-to-left and-formula)` accessor
`(and-form-to-right and-formula)` accessor,
tensors
`(make-tensor formula1 formula2)` constructor
`(tensor-form-to-left tensor-formula)` accessor
`(tensor-form-to-right tensor-formula)` accessor,
universally quantified formulas
`(make-all var formula)` constructor
`(all-form-to-var all-formula)` accessor
`(all-form-to-kernel all-formula)` accessor,
existentially quantified formulas
`(make-ex var formula)` constructor
`(ex-form-to-var ex-formula)` accessor
`(ex-form-to-kernel ex-formula)` accessor,
universally quantified formulas without computational content
`(make-allnc var formula)` constructor
`(allnc-form-to-var allnc-formula)` accessor
`(allnc-form-to-kernel allnc-formula)` accessor,
existentially quantified formulas without computational content
`(make-exnc var formula)` constructor
`(exnc-form-to-var exnc-formula)` accessor
`(exnc-form-to-kernel exnc-formula)` accessor,
existentially quantified formulas in the sense of classical arithmetic
`(make-exca var formula)` constructor
`(exca-form-to-var exca-formula)` accessor
`(exca-form-to-kernel exca-formula)` accessor,
existentially quantified formulas in the sense of classical logic
`(make-excl var formula)` constructor
`(excl-form-to-var excl-formula)` accessor
`(excl-form-to-kernel excl-formula)` accessor.

For convenience we also have as generalized constructors

`(mk-imp formula formula1 ...)` implication
`(mk-neg formula1 ...)` negation
`(mk-neg-log formula1 ...)` logical negation
`(mk-and formula formula1 ...)` conjunction
`(mk-tensor formula formula1 ...)` tensor
`(mk-all var1 ... formula)` all-formula

<code>(mk-ex var1 ... formula)</code>	ex-formula
<code>(mk-allnc var1 ... formula)</code>	allnc-formula
<code>(mk-exnc var1 ... formula)</code>	exnc-formula
<code>(mk-exca var1 ... formula)</code>	classical ex-formula (arithmetical)
<code>(mk-excl var1 ... formula)</code>	classical ex-formula (logical)

and as generalized accessors

```

      (imp-form-to-premises-and-final-conclusion formula)
      (tensor-form-to-parts formula)
      (all-form-to-vars-and-final-kernel formula)
      (ex-form-to-vars-and-final-kernel formula)

```

and similarly for `exca`-forms and `excl`-forms. Occasionally it is convenient to have

```

      (imp-form-to-premises formula <n>)           all (first n) premises
      (imp-form-to-final-conclusion formula <n>)

```

where the latter computes the final conclusion (conclusion after removing the first n premises) of the formula.

It is also useful to have some general procedures working for arbitrary quantified formulas. We provide

<code>(make-quant-formula quant var1 ... kernel)</code>	constructor
<code>(quant-form-to-quant quant-form)</code>	accessor
<code>(quant-form-to-vars quant-form)</code>	accessor
<code>(quant-form-to-kernel quant-form)</code>	accessor
<code>(quant-form? x)</code>	test.

and for convenience also

```

      (mk-quant quant var1 ... formula).

```

To fold and unfold formulas we have

```

      (fold-formula formula)
      (unfold-formula formula).

```

To test equality of formulas up to normalization and α -equality we use

```

      (classical-formula=? formula1 formula2)
      (formula=? formula1 formula2),

```

where in the first procedure we unfold before comparing.

Moreover we need

```

      (formula-to-free formula),
      (nbe-formula-to-type formula),
      (formula-to-prime-subformulas formula),

```

Constructors, accessors and a test for comprehension terms are

```

      (make-cterm var1 ... formula)  constructor

```

<code>(cterm-to-vars cterm)</code>	accessor
<code>(cterm-to-formula cterm)</code>	accessor
<code>(cterm? x)</code>	test.

Moreover we need

```
(cterm-to-free cterm)
(cterm=? x)
(classical-cterm=? x)
(fold-cterm cterm)
(unfold-cterm cterm).
```

Normalization of formulas is done with

```
(normalize-formula formula)  normalization,
```

abbreviated `nf`.

To check equality of formulas we use

```
(classical-formula=? formula1 formula2)
(formula=? formula1 formula2)
```

where the former unfolds the classical existential quantifiers and normalizes all subterms in its formulas.

Display functions for formulas and comprehension terms are

```
(formula-to-string formula)
(cterm-to-string cterm).
```

The former is abbreviated `nf`.

We can define simultaneous substitution for type, object and predicate variables in a formula, via `tsubst`, `subst` and `psubst`. It is assumed that `subst` only affects those variables whose type is not changed by `tsubst`, and that `psubst` only affects those predicate variables whose arity is not changed by `tsubst`.

In the quantifier case of the recursive definition, the abstracted variable may need to be renamed. However, its type can be affected by `tsubst`. Then the renaming cannot be made part of `subst`, because then the condition above would be violated. Therefore we carry along a procedure `rename` renaming variables, which remembers the renaming of variables done so far.

We will also need formula substitution to compute the formula of an assumption constant. However, there (type and) predicate variables are (implicitly) considered to be bound. Therefore, we also have to carry along a procedure `prename` renaming predicate variables, which remembers the renaming of predicate variables done so far.

```
(formula-substitute formula tosubst)
(formula-subst formula arg val)
(cterm-substitute cterm tosubst)
(cterm-subst cterm arg val)
```

Display functions for predicate substitutions are

```
(display-p-substitution psubst)
(p-substitution-to-string psubst)
```

8. ASSUMPTION VARIABLES AND CONSTANTS

S:AssumptionVarConst

8.1. Assumption variables. Assumption variables are for proofs what variables are for terms. The main difference, however, is that assumption variables have formulas as types, and that formulas may contain free variables. Therefore we must be careful when substituting terms for variables in assumption variables. Our solution (as in Matthes’ thesis [9]) is to consider an assumption variable as a pair of a (typefree) identifier and a formula, and to take equality of assumption variables to mean that both components are equal. Rather than using symbols as identifiers we prefer to use numbers (i.e. indices). However, sometimes it is useful to provide an optional string as name for display purposes.

We need a constructor, accessors and tests for assumption variables.

```
(make-avar formula index name)  constructor
(avar-to-formula avar)           accessor
(avar-to-index avar)             accessor
(avar-to-name avar)              accessor
(avar? x)                        test
(avar=? avar1 avar2)             test.
```

For convenience we have the function

```
(mk-avar formula <index> <name>)
```

The formula is a required argument; however, the remaining arguments are optional. The default for the name string is `u`. We also require that a function

```
(formula-to-new-avar formula)
```

is defined that returns an assumption variable of the requested formula different from all assumption variables that have ever been returned by any of the specified functions so far.

An assumption constant appears in a proof, as an axiom, a theorem or a global assumption. Its formula is given as an “uninstantiated formula”, where only type and predicate variables can occur freely; these are considered to be bound in the assumption constant. In the proof the bound type variables are implicitly instantiated by types, and the bound predicate variables by comprehension terms (the arity of a comprehension term is the type-instantiated arity of the corresponding predicate variable). Since we do not have type and predicate quantification in formulas, the assumption constant contains these parts left implicit in the proof: `tsubst` and `pinst` (which will become a `psubst`, once the arities of predicate variables are type-instantiated with `tsubst`).

So we have assumption constants of the following kinds:

- axioms,

- theorems, and
- global assumptions.

To normalize a proof we will first translate it into a term, then normalize the term and finally translate the normal term back into a proof. To make this work, in case of axioms we pass to the term appropriate formulas: all-formulas for induction, an existential formula for existence introduction, and an existential formula together with a conclusion for existence elimination. During normalization of the term these formulas are passed along. When the normal form is reached, we have to translate back into a proof. Then these formulas are used to reconstruct the axiom in question.

Internally, the formula of an assumption constant is split into an uninstantiated formula where only type and predicate variables can occur freely, and a substitution for at most these type and predicate variables. The formula assumed by the constant is the result of carrying out this substitution in the uninstantiated formula. Note that free variables may again occur in the assumed formula. For example, assumption constants axiomatizing the existential quantifier will internally have the form

$$\begin{aligned}
 &(\text{aconst Ex-Intro } \forall \hat{x}^\alpha. \hat{P}(\hat{x}) \rightarrow \exists \hat{x}^\alpha \hat{P}(\hat{x}) \ (\alpha \mapsto \tau, \hat{P}^{(\alpha)} \mapsto \{ \hat{z}^\tau \mid A \})) \\
 &(\text{aconst Ex-Elim } \exists \hat{x}^\alpha \hat{P}(\hat{x}) \rightarrow (\forall \hat{x}^\alpha. \hat{P}(\hat{x}) \rightarrow \hat{Q}) \rightarrow \hat{Q} \\
 &\quad (\alpha \mapsto \tau, \hat{P}^{(\alpha)} \mapsto \{ \hat{z}^\tau \mid A \}, \hat{Q} \mapsto \{ \mid B \}))
 \end{aligned}$$

Interface for general assumption constants. To avoid duplication of code it is useful to formulate some procedures generally for arbitrary assumption constants, i.e. for all of the forms listed above.

```

(make-aconst name kind uninstant-formula tpsubst
  repro-formula1 ...)                                constructor
(aconst-to-name aconst)                             accessor
(aconst-to-kind aconst)                             accessor
(aconst-to-uninst-formula aconst)                   accessor
(aconst-to-tpsubst aconst)                          accessor
(aconst-to-repro-formulas aconst)                   accessor
(aconst? x)                                          test.

```

To construct the formula associated with an aconst, it is useful to separate the instantiated formula from the variables to be generalized. The latter can be obtained as free variables in inst-formula. We therefore provide

```

(aconst-to-inst-formula aconst)
(aconst-to-formula aconst)

```

We also provide

```

(aconst? aconst)
(aconst=? aconst1 aconst2)
(aconst-without-rules? aconst)
(aconst-to-string aconst)

```

8.2. Axiom constants. We use the natural numbers as a prototypical finitary algebra; recall Figure 1. Assume that n, p are variables of type **nat**, **boole**. Let \approx denote the equality relation in the model. Recall the domain of type **boole**, consisting of **tt**, **ff** and the bottom element **bb**. The boolean valued functions equality $=_{\text{nat}}: \text{nat} \rightarrow \text{nat} \rightarrow \text{boole}$ and existence (definedness, totality) $e_{\text{nat}}: \text{nat} \rightarrow \text{boole}$ need to be continuous. So we have

$$\begin{aligned}
&=(0, 0) \approx \text{tt} \\
&=(0, S\hat{n}) \approx =(S\hat{n}, 0) \approx \text{ff} & e(0) \approx \text{tt} \\
&=(S\hat{n}_1, S\hat{n}_2) \approx =(\hat{n}_1, \hat{n}_2) & e(S\hat{n}) \approx e(\hat{n}) \\
&=(\text{bb}_{\text{nat}}, \hat{n}) \approx =(\hat{n}, \text{bb}_{\text{nat}}) \approx \text{bb} & e(\text{bb}_{\text{nat}}) \approx \text{bb} \\
&=(\infty_{\text{nat}}, \hat{n}) \approx =(\hat{n}, \infty_{\text{nat}}) \approx \text{bb} & e(\infty_{\text{nat}}) \approx \text{bb}
\end{aligned}$$

Write T, F for $\text{atom}(\text{tt}), \text{atom}(\text{ff})$, $r = s$ for $\text{atom}(=(r, s))$ and $E(r)$ for $\text{atom}(e(r))$. We stipulate as axioms

T	Truth-Axiom
$\hat{x} \approx \hat{x}$	Eq-Refl
$\hat{x}_1 \approx \hat{x}_2 \rightarrow \hat{x}_2 \approx \hat{x}_1$	Eq-Symm
$\hat{x}_1 \approx \hat{x}_2 \rightarrow \hat{x}_2 \approx \hat{x}_3 \rightarrow \hat{x}_1 \approx \hat{x}_3$	Eq-Trans
$\forall \hat{x} \hat{f}_1 \hat{x} \approx \hat{f}_2 \hat{x} \rightarrow \hat{f}_1 \approx \hat{f}_2$	Eq-Ext
$\hat{x}_1 \approx \hat{x}_2 \rightarrow \hat{P}(\hat{x}_1) \rightarrow \hat{P}(\hat{x}_2)$	Eq-Compat
$\text{Total}_{\rho \rightarrow \sigma}(\hat{f}) \leftrightarrow \forall \hat{x}. \text{Total}_{\rho}(\hat{x}) \rightarrow \text{Total}_{\sigma}(\hat{f} \hat{x})$	Total
$\text{Total}_{\rho}(c)$	Constr-Total
$\text{Total}(c\vec{\hat{x}}) \rightarrow \text{Total}(\hat{x}_i)$	Constr-Total-Args

and for every finitary algebra, e.g. **nat**

$\hat{n}_1 \approx \hat{n}_2 \rightarrow E(\hat{n}_1) \rightarrow \hat{n}_1 = \hat{n}_2$	Eq-to==1-nat
$\hat{n}_1 \approx \hat{n}_2 \rightarrow E(\hat{n}_2) \rightarrow \hat{n}_1 = \hat{n}_2$	Eq-to==2-nat
$\hat{n}_1 = \hat{n}_2 \rightarrow \hat{n}_1 \approx \hat{n}_2$	==to-Eq-nat
$\hat{n}_1 = \hat{n}_2 \rightarrow E(\hat{n}_1)$	==to-E1-nat
$\hat{n}_1 = \hat{n}_2 \rightarrow E(\hat{n}_2)$	==to-E2-nat
$\text{Total}(\hat{n}) \rightarrow E(\hat{n})$	Total-to-E-nat
$E(\hat{n}) \rightarrow \text{Total}(\hat{n})$	E-to-Total-nat

Here c is a constructor. Notice that in $\text{Total}(c\vec{\hat{x}}) \rightarrow \text{Total}(\hat{x}_i)$, the type of $(c\vec{\hat{x}})$ need not be a finitary algebra, and hence \hat{x}_i may have a function type.

Remark. $(E(\hat{n}_1) \rightarrow \hat{n}_1 = \hat{n}_2) \rightarrow (E(\hat{n}_2) \rightarrow \hat{n}_1 = \hat{n}_2) \rightarrow \hat{n}_1 \approx \hat{n}_2$ is *not* valid in our intended model (see Figure 1), since we have *two* distinct undefined objects bb_{nat} and ∞_{nat} .

We abbreviate

$$\forall \hat{x}. \text{Total}_{\rho}(\hat{x}) \rightarrow A \quad \text{by} \quad \forall xA,$$

$$\exists \hat{x}. \text{Total}_\rho(\hat{x}) \wedge A \quad \text{by} \quad \exists x A.$$

Formally, these abbreviations appear as axioms

$$\begin{aligned} \forall x \hat{P}(x) &\rightarrow \forall \hat{x}. \text{Total}(\hat{x}) \rightarrow \hat{P}(\hat{x}) && \text{All-AllPartial} \\ (\forall \hat{x}. \text{Total}(\hat{x}) \rightarrow \hat{P}(\hat{x})) &\rightarrow \forall x \hat{P}(x) && \text{AllPartial-All} \\ \exists x \hat{P}(x) &\rightarrow \exists \hat{x}. \text{Total}(\hat{x}) \wedge \hat{P}(\hat{x}) && \text{Ex-ExPartial} \\ (\exists \hat{x}. \text{Total}(\hat{x}) \wedge \hat{P}(\hat{x})) &\rightarrow \exists x \hat{P}(x) && \text{ExPartial-Ex} \end{aligned}$$

and for every finitary algebra, e.g. **nat**

$$\begin{aligned} \forall n \hat{P}(n) &\rightarrow \forall \hat{n}. E(\hat{n}) \rightarrow \hat{P}(\hat{n}) && \text{All-AllPartial-nat} \\ (\exists \hat{n}. E(\hat{n}) \wedge \hat{P}(\hat{n})) &\rightarrow \exists n \hat{P}(n) && \text{ExPartial-Ex-nat} \end{aligned}$$

Notice that **AllPartial-All-nat** i.e. $(\forall \hat{n}. E(\hat{n}) \rightarrow \hat{P}(\hat{n})) \rightarrow \forall n \hat{P}(n)$ is provable (since $E(n) \mapsto T$). Similarly, **Ex-ExPartial-nat**, i.e. $\exists n \hat{P}(n) \rightarrow \exists \hat{n}. E(\hat{n}) \wedge \hat{P}(\hat{n})$ is provable.

Finally we have axioms for the existential quantifier

$$\begin{aligned} \forall \hat{x}^\alpha. \hat{P}(\hat{x}) &\rightarrow \exists \hat{x}^\alpha \hat{P}(\hat{x}) && \text{Ex-Intro} \\ \exists \hat{x}^\alpha \hat{P}(\hat{x}) &\rightarrow (\forall \hat{x}^\alpha. \hat{P}(\hat{x}) \rightarrow \hat{Q}) \rightarrow \hat{Q} && \text{Ex-Elim} \end{aligned}$$

The assumption constants corresponding to these axioms are

<code>truth-aconst</code>	for Truth-Axiom
<code>eq-refl-aconst</code>	for Eq-Refl
<code>eq-symm-aconst</code>	for Eq-Symm
<code>eq-trans-aconst</code>	for Eq-Trans
<code>ext-aconst</code>	for Eq-Ext
<code>eq-compat-aconst</code>	for Eq-Compat
<code>total-aconst</code>	for Total
<code>(finalg-to-eq-to==1-aconst finalg)</code>	for Eq-to==1
<code>(finalg-to-eq-to==2-aconst finalg)</code>	for Eq-to==2
<code>(finalg-to==to-eq-aconst finalg)</code>	for ==to-Eq
<code>(finalg-to==to-e-1-aconst finalg)</code>	for ==to-E-1
<code>(finalg-to==to-e-2-aconst finalg)</code>	for ==to-E-2
<code>(finalg-to-total-to-e-aconst finalg)</code>	for Total-to-E
<code>(finalg-to-e-to-total-aconst finalg)</code>	for E-to-Total
<code>all-allpartial-aconst</code>	for All-AllPartial
<code>allpartial-all-aconst</code>	for AllPartial-All
<code>ex-expartial-aconst</code>	for Ex-ExPartial
<code>expartial-ex-aconst</code>	for ExPartial-Ex
<code>(finalg-to-all-allpartial-aconst finalg)</code>	for All-AllPartial
<code>(finalg-to-expartial-ex-aconst finalg)</code>	for ExPartial-Ex

We now spell out what precisely we mean by induction over simultaneous free algebras $\vec{\mu} = \mu \vec{\alpha} \vec{\kappa}$, with goal formulas $\forall x_j^{\mu_j} \hat{P}_j(x_j)$. For the constructor type

$$\kappa_i = \vec{\rho} \rightarrow (\vec{\sigma}_1 \rightarrow \alpha_{j_1}) \rightarrow \cdots \rightarrow (\vec{\sigma}_n \rightarrow \alpha_{j_n}) \rightarrow \alpha_j \in \mathbf{KT}(\vec{\alpha})$$

we have the *step formula*

$$\begin{aligned} D_i := \forall y_1^{\rho_1}, \dots, y_m^{\rho_m}, y_{m+1}^{\vec{\sigma}_1 \rightarrow \mu_{j_1}}, \dots, y_{m+n}^{\vec{\sigma}_n \rightarrow \mu_{j_n}}. \forall \vec{x}^{\vec{\sigma}_1} \hat{P}_{j_1}(y_{m+1} \vec{x}) \rightarrow \cdots \rightarrow \\ \forall \vec{x}^{\vec{\sigma}_n} \hat{P}_{j_n}(y_{m+n} \vec{x}) \rightarrow \\ \hat{P}_j(\text{constr}_i^{\vec{\mu}}(\vec{y})). \end{aligned}$$

Here $\vec{y} = y_1^{\rho_1}, \dots, y_m^{\rho_m}, y_{m+1}^{\vec{\sigma}_1 \rightarrow \mu_{j_1}}, \dots, y_{m+n}^{\vec{\sigma}_n \rightarrow \mu_{j_n}}$ are the *components* of the object $\text{constr}_i^{\vec{\mu}}(\vec{y})$ of type μ_j under consideration, and

$$\forall \vec{x}^{\vec{\sigma}_1} \hat{P}_{j_1}(y_{m+1} \vec{x}), \dots, \forall \vec{x}^{\vec{\sigma}_n} \hat{P}_{j_n}(y_{m+n} \vec{x})$$

are the hypotheses available when proving the induction step. The induction axiom Ind_{μ_j} then proves the formula

$$\text{Ind}_{\mu_j} : D_1 \rightarrow \cdots \rightarrow D_k \rightarrow \forall x_j^{\mu_j} \hat{P}_j(x_j).$$

We will often write Ind_j for Ind_{μ_j} .

An example is

$$E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4 \rightarrow \forall x_1^{\text{tree}} \hat{P}_1(x_1)$$

with

$$\begin{aligned} E_1 &:= \hat{P}_1(\text{Leaf}), \\ E_2 &:= \forall x^{\text{tlist}}. \hat{P}_2(x) \rightarrow \hat{P}_1(\text{Branch}(x)), \\ E_3 &:= \hat{P}_2(\text{Empty}), \\ E_4 &:= \forall x_1^{\text{tree}}, x_2^{\text{tlist}}. \hat{P}_1(x_1) \rightarrow \hat{P}_2(x_2) \rightarrow \hat{P}_2(\text{Tcons}(x_1, x_2)). \end{aligned}$$

Here the fact that we deal with a simultaneous induction (over **tree** and **tlist**), and that we prove a formula of the form $\forall x^{\text{tree}} \dots$, can all be inferred from what is given: the $\forall x^{\text{tree}} \dots$ is right there, and for **tlist** we can look up the simultaneously defined algebras. – The internal representation is

$$\begin{aligned} (\text{aconst Ind } E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4 \rightarrow \forall x_1^{\text{tree}} \hat{P}_1(x_1) \\ (\hat{P}_1 \mapsto \{x_1^{\text{tree}} \mid A_1\}, \hat{P}_2 \mapsto \{x_2^{\text{tlist}} \mid A_2\})) \end{aligned}$$

A simplified version (without the recursive calls) of the induction axiom is the following cases axiom.

$$(\text{aconst Cases } E_1 \rightarrow E_2 \rightarrow \forall x_1^{\text{tree}} \hat{P}_1(x_1) \ (\hat{P}_1 \mapsto \{x_1^{\text{tree}} \mid A_1\}))$$

with

$$\begin{aligned} E_1 &:= \hat{P}_1(\text{Leaf}), \\ E_2 &:= \forall x^{\text{tlist}} \hat{P}_1(\text{Branch}(x)). \end{aligned}$$

However, rather than using this as an assumption constant we will – parallel to the **if**-construct for terms – use a **cases**-construct for proofs. This does

not change our notion of proof; it is done to have the `if`-construct in the extracted programs.

The assumption constants corresponding to these axioms are generated by

```
(all-formulas-to-ind-aconst all-formula1 ...) for Ind
(all-formula-to-cases-aconst all-formula)      for Cases
```

For the introduction and elimination axioms `Ex-Intro` and `Ex-Elim` for the existential quantifier we provide

```
(ex-formula-to-ex-intro-aconst ex-formula)
(ex-formula-and-concl-to-ex-elim-aconst ex-formula concl)
```

and similarly for `exnc` instead of `ex`.

To deal with inductively defined predicate constants, we need additional axioms with names “Intro” and “Elim”, which can be generated by

```
(number-and-idpredconst-to-intro-aconst i idpc)
(imp-formulas-to-elim-aconst imp-formula1 ...);
```

here an `imp-formula` is expected to have the form $I(\vec{x}) \rightarrow A$.

SS:Theorems

8.3. Theorems. A theorem is a special assumption constant. Theorems are normally created after successfully completing an interactive proof. One may also create a theorem from an explicitly given (closed) proof. The command is

```
(add-theorem string . opt-proof) or save
```

From a theorem name we can access its `aconst`, its (original) proof and also its instantiated proof by

```
(theorem-name-to-aconst string)
(theorem-name-to-proof string)
(theorem-name-to-inst-proof string)
```

We also provide

```
(remove-theorem string1 ...)
(display-theorems string1 ...)
```

Initially we provide the following theorems

$\text{atom}(p) \rightarrow p = \mathbf{tt}$	<code>Atom-True</code>
$(\text{atom}(p) \rightarrow F) \rightarrow p = \mathbf{ff}$	<code>Atom-False</code>
$F \rightarrow \text{atom}(p)$	<code>Efq-Atom</code>
$((\text{atom}(p) \rightarrow F) \rightarrow F) \rightarrow \text{atom}(p)$	<code>Stab-Atom</code>

and for every finitary algebra, e.g. `nat`

$n = n$	<code>--Refl-nat</code>
$\hat{n}_1 = \hat{n}_2 \rightarrow \hat{n}_2 = \hat{n}_1$	<code>--Symm-nat</code>
$\hat{n}_1 = \hat{n}_2 \rightarrow \hat{n}_2 = \hat{n}_3 \rightarrow \hat{n}_1 = \hat{n}_3$	<code>--Trans-nat</code>

Proof of Atom-True. By Ind. In case \mathbf{tt} use Eq-Compat with $\mathbf{tt} \approx =(\mathbf{tt}, \mathbf{tt})$ to infer $\text{atom}(=(\mathbf{tt}, \mathbf{tt}))$ (i.e. $\mathbf{tt} = \mathbf{tt}$) from $\text{atom}(\mathbf{tt})$. In case \mathbf{ff} use Eq-Compat with $\mathbf{ff} \approx =(\mathbf{ff}, \mathbf{tt})$ to infer $\text{atom}(=(\mathbf{ff}, \mathbf{tt}))$ (i.e. $\mathbf{ff} = \mathbf{tt}$) from $\text{atom}(\mathbf{ff})$. \square

Proof of Atom-False. Use Ind, and Truth-Axiom in both cases. – Notice that the more general $(\text{atom}(\hat{p}) \rightarrow F) \rightarrow \hat{p} = \mathbf{ff}$ does *not* hold with \mathbf{bb} for \hat{p} , since $=(\mathbf{bb}, \mathbf{ff}) \approx \mathbf{bb}$. \square

Proof of Efq-Atom. Again by Ind. In case \mathbf{tt} use Truth-Axiom, and the case \mathbf{ff} is obvious. \square

Proof of Stab-Atom. By Ind. In case \mathbf{tt} use Truth-Axiom, and the case \mathbf{ff} is obvious. \square

Remark. Notice that from Efq-Atom one easily obtains $F \rightarrow A$ for every formula A all whose strictly positions occurrences of prime formulas are of the form $\text{atom}(r)$, by induction on A . For all other formulas we shall make use of the global assumption Efq: $F \rightarrow \hat{P}$ (cf. Section 8.4). Similarly, Notice that from Stab-Atom one again obtains $((A \rightarrow F) \rightarrow F) \rightarrow A$ for every formula A all whose strictly positions occurrences of prime formulas are of the form $\text{atom}(r)$, by induction on A . For all other formulas we shall make use of the global assumption Stab: $((\hat{P} \rightarrow F) \rightarrow F) \rightarrow \hat{P}$ (cf. Section 8.4).

Proof of ==Refl-nat. Use Ind, and Truth-Axiom in both cases. – Notice that $\hat{n} = \hat{n}$ does *not* hold, since $=(\mathbf{bb}, \mathbf{bb}) \approx \mathbf{bb}$. \square

Here are some other examples of theorems; we give the internal representation as assumption constants, which show how the assumed formula is split into an uninstantiated formula and a substitution, in this case a type substitution $\alpha \mapsto \rho$, an object substitution $f^{\alpha \rightarrow \text{nat}} \mapsto g^{\rho \rightarrow \text{nat}}$ and a predicate variable substitution $\hat{P}^{(\alpha)} \mapsto \{\hat{z}^\rho \mid A\}$.

(aconst Cvind-with-measure-11

$$\begin{aligned} & \forall f^{\alpha \rightarrow \text{nat}}. (\forall x^\alpha. \forall y (f(y) < f(x) \rightarrow \hat{P}(y)) \rightarrow \hat{P}(x)) \rightarrow \forall x \hat{P}(x) \\ & (\alpha \mapsto \rho, f^{\alpha \rightarrow \text{nat}} \mapsto g^{\rho \rightarrow \text{nat}}, \hat{P}^{(\alpha)} \mapsto \{\hat{z}^\rho \mid A\}). \end{aligned}$$

(aconst Minpr-with-measure-111

$$\begin{aligned} & \forall f^{\alpha \rightarrow \text{nat}}. \exists^{\text{cl}} x^\alpha \hat{P}(x) \rightarrow \exists^{\text{cl}} x. \hat{P}(x) ! \forall y. f(y) < f(x) \rightarrow \hat{P}(y) \rightarrow \perp \\ & (\alpha \mapsto \rho, f^{\alpha \rightarrow \text{nat}} \mapsto g^{\rho \rightarrow \text{nat}}, \hat{P}^{(\alpha)} \mapsto \{\hat{z}^\rho \mid A\}). \end{aligned}$$

Here \exists^{cl} is the classical existential quantifier defined by $\exists^{\text{cl}} x A := \forall x (A \rightarrow \perp) \rightarrow \perp$ with the logical form of falsity \perp (as opposed to the arithmetical form $(\text{atom } \mathbf{ff})$). 1 indicates “logic” (we have used the logical form of falsity), the first 1 that we have one predicate variable \hat{P} , and the second that we quantify over just one variable x . Both theorems can easily be generalized to more such parameters.

When dealing with classical logic it will be useful to have

$$(\hat{P} \rightarrow \hat{P}_1) \rightarrow ((\hat{P} \rightarrow \perp) \rightarrow \hat{P}_1) \rightarrow \hat{P}_1 \quad \text{Cases-Log}$$

The proof uses the global assumption Stab-Log (see below) for \hat{P}_1 ; hence we cannot extract a term from it.

The assumption constants corresponding to these theorems are generated by

`(theorem-name-to-aconst name)`

SS:GlobalAss

8.4. Global assumptions. A global assumption is a special assumption constant. It provides a proposition whose proof does not concern us presently. Global assumptions are added, removed and displayed by

`(add-global-assumption name formula)` (abbreviated `aga`)

`(remove-global-assumption string1 ...)`

`(display-global-assumptions string1 ...)`

We initially supply global assumptions for ex-falso-quodlibet and stability, both in logical and arithmetical form (for our two forms of falsity).

$\perp \rightarrow \hat{P}$	Efq-Log
$((\hat{P} \rightarrow \perp) \rightarrow \perp) \rightarrow \hat{P}$	Stab-Log
$F \rightarrow \hat{P}$	Efq
$((\hat{P} \rightarrow F) \rightarrow F) \rightarrow \hat{P}$	Stab

The assumption constants corresponding to these global assumptions are generated by

`(global-assumption-name-to-aconst name)`

9. PROOFS

Proof

Proofs are built from assumption variables and assumption constants (i.e. axioms, theorems and global assumption) by the usual rules of natural deduction, i.e. introduction and elimination rules for implication, conjunction and universal quantification. From a proof we can read off its *context*, which is an ordered list of object and assumption variables.

9.1. Constructors and accessors. We have constructors, accessors and tests for assumption variables

<code>(make-proof-in-avar-form avar)</code>	constructor
<code>(proof-in-avar-form-to-avar proof)</code>	accessor,
<code>(proof-in-avar-form? proof)</code>	test,

for assumption constants

<code>(make-proof-in-aconst-form aconst)</code>	constructor
<code>(proof-in-aconst-form-to-aconst proof)</code>	accessor
<code>(proof-in-aconst-form? proof)</code>	test,

for implication introduction

<code>(make-proof-in-imp-intro-form avar proof)</code>	constructor
<code>(proof-in-imp-intro-form-to-avar proof)</code>	accessor
<code>(proof-in-imp-intro-form-to-kernel proof)</code>	accessor
<code>(proof-in-imp-intro-form? proof)</code>	test,

for implication elimination

<code>(make-proof-in-imp-elim-form <i>proof1 proof2</i>)</code>	constructor
<code>(proof-in-imp-elim-form-to-op <i>proof</i>)</code>	accessor
<code>(proof-in-imp-elim-form-to-arg <i>proof</i>)</code>	accessor
<code>(proof-in-imp-elim-form? <i>proof</i>)</code>	test,

for and introduction

<code>(make-proof-in-and-intro-form <i>proof1 proof2</i>)</code>	constructor
<code>(proof-in-and-intro-form-to-left <i>proof</i>)</code>	accessor
<code>(proof-in-and-intro-form-to-right <i>proof</i>)</code>	accessor
<code>(proof-in-and-intro-form? <i>proof</i>)</code>	test,

for and elimination

<code>(make-proof-in-and-elim-left-form <i>proof</i>)</code>	constructor
<code>(make-proof-in-and-elim-right-form <i>proof</i>)</code>	constructor
<code>(proof-in-and-elim-left-form-to-kernel <i>proof</i>)</code>	accessor
<code>(proof-in-and-elim-right-form-to-kernel <i>proof</i>)</code>	accessor
<code>(proof-in-and-elim-left-form? <i>proof</i>)</code>	test
<code>(proof-in-and-elim-right-form? <i>proof</i>)</code>	test,

for all introduction

<code>(make-proof-in-all-intro-form <i>var proof</i>)</code>	constructor
<code>(proof-in-all-intro-form-to-var <i>proof</i>)</code>	accessor
<code>(proof-in-all-intro-form-to-kernel <i>proof</i>)</code>	accessor
<code>(proof-in-all-intro-form? <i>proof</i>)</code>	test,

for all elimination

<code>(make-proof-in-all-elim-form <i>proof term</i>)</code>	constructor
<code>(proof-in-all-elim-form-to-op <i>proof</i>)</code>	accessor
<code>(proof-in-all-elim-form-to-arg <i>proof</i>)</code>	accessor
<code>(proof-in-all-elim-form? <i>proof</i>)</code>	test

and for cases-constructs

<code>(make-proof-in-cases-form <i>test alt1 ...</i>)</code>	constructor
<code>(proof-in-cases-form-to-test <i>proof</i>)</code>	accessor
<code>(proof-in-cases-form-to-alts <i>proof</i>)</code>	accessor
<code>(proof-in-cases-form-to-rest <i>proof</i>)</code>	accessor
<code>(proof-in-cases-form? <i>proof</i>)</code>	test.

It is convenient to have more general introduction and elimination operators that take arbitrary many arguments. The former works for implication-introduction and all-introduction, and the latter for implication-elimination, and-elimination and all-elimination.

`(mk-proof-in-intro-form x1 ... proof)`

```
(mk-proof-in-elim-form proof arg1 ...)
(proof-in-intro-form-to-kernel-and-vars proof)
(proof-in-elim-form-to-final-op proof)
(proof-in-elim-form-to-args proof).
```

(mk-proof-in-intro-form *x1* ... *proof*) is formed from *proof* by first abstracting *x1*, then *x2* and so on. Here *x1*, *x2* ... can be assumption or object variables. We also provide

```
(mk-proof-in-and-intro-form proof proof1 ...)
```

In our setup there are axioms rather than rules for the existential quantifier. However, sometimes it is useful to construct proofs as if an existence introduction rule would be present; internally then an existence introduction axiom is used.

```
(make-proof-in-ex-intro-form term ex-formula proof-of-inst)
(mk-proof-in-ex-intro-form .
  terms-and-ex-formula-and-proof-of-inst)
```

Moreover we need

```
(proof? x)
(proof=? proof1 proof2)
(proofs=? proofs1 proofs2)
(proof-to-formula proof)
(proof-to-context proof)
(proof-to-free proof)
(proof-to-free-avars proof)
(proof-to-bound-avars proof)
(proof-to-free-and-bound-avars proof)
(proof-to-aconst-without-rules proof).
(proof-to-aconst proof).
```

To work with contexts we need

```
(context-to-vars context)
(context-to-avars context)
(context=? context1 context2).
```

9.2. Normalization. Normalization of proofs will be done by reduction to normalization of terms. (1) Construct a term from the proof. To do this properly, create for every free avar in the given proof a new variable whose type comes from the formula of the avar; store this information. Note that in this construction one also has to create new variables for the bound avars. Similary to avars we have to treat assumption constants which are not axioms, i.e. theorems or global assumptions. (2) Normalize the resulting term. (3) Reconstruct a normal proof from this term, the end formula and the stored information. – The critical variables are carried along for efficiency reasons.

To assign recursion constants to induction constants, we need to associate type variables with predicate variables, in such a way that we can later refer to this assignment. Therefore we carry along a procedure `pvar-to-tvar` which remembers the assignment done so far (cf. `make-rename`).

Due to our distinction between general variables x^0, x^1, x^2, \dots and variables x_0, x_1, x_2, \dots intended to range over existing (i.e. total) objects only, η -conversion of proofs cannot be done via reduction to η -conversion of terms. To see this, consider the proof

$$\frac{\frac{\frac{\forall \hat{x} P \hat{x}}{Px}}{\forall x Px}}{\forall \hat{x} P \hat{x} \rightarrow \forall x Px}$$

The proof term is $\lambda u \lambda x. ux$. If we η -normalize this to λuu , the proven formula would be all $\forall \hat{x} P \hat{x} \rightarrow \forall \hat{x} P \hat{x}$. Therefore we split `nbe-normalize-proof` into `nbe-normalize-proof-without-eta` and `proof-to-eta-nf`.

Moreover, for a full normalization of proofs (including permutative conversions) we need a preprocessing step that η -expands each `ex-elim` axiom such that the conclusion is atomic or existential.

We need the following functions.

```
(proof-and-genavar-var-alist-to-pterm pvar-to-tvar proof)
(nppterm-and-var-genavar-alist-and-formula-to-proof
 npterm var-genavar-alist crit formula)
(elim-npterm-and-var-genavar-alist-to-proof
 npterm var-genavar-alist crit).
```

Then we can define `nbe-normalize-proof`, abbreviated `np`.

9.3. Substitution. In a proof we can substitute

- types for type variables (by a type variable substitution `tsubst`),
- terms for variables (by a substitution `subst`),
- comprehension terms for predicate variables (by a predicate variable substitution `psubst`), and
- proofs for assumption variables (by an assumption variable substitution `asubst`).

It is assumed that `subst` only affects those vars whose type is not changed by `tsubst`, `psubst` only affects those predicate variables whose arity is not changed by `tsubst`, and that `asubst` only affects those assumption variables whose formula is not changed by `tsubst`, `subst` and `psubst`.

In the abstraction cases of the recursive definition, the abstracted variable (or assumption variable) may need to be renamed. However, its type (or formula) can be affected by `tsubst` (or `tsubst`, `subst` and `psubst`). Then the renaming cannot be made part of `subst` (or `asubst`), because the condition above would be violated. Therefore we carry along procedures `rename` renaming variables and `arename` for assumption variables, which remember the renaming done so far.

All these substitutions can be packed together, as an argument `topasubst` for `proof-substitute`.

`(proof-substitute proof topasubst)`

If we want to substitute for a single variable only (which can be a type-, an object-, a predicate - or an assumption-variable), then we can use

`(proof-subst proof arg val)`

The procedure `expand-theorems` expects a proof and a test whether a string denotes a theorem to be replaced by its proof. The result is the (normally quite long) proof obtained by replacing the theorems by their saved proofs.

`(expand-theorems proof name-test?)`

9.4. Display. There are many ways to display a proof. We normally use `display-proof` for a linear representation, showing the formulas and the rules used. When we in addition want to check the correctness of the proof, we can use `check-and-display-proof`.

However, we also provide a readable type-free lambda expression via `(proof-to-expr proof)`.

To display proofs we use the following functions. In case the optional proof argument is not present, the current proof of an interactive proof development is taken instead.

<code>(display-proof . opt-proof)</code>	abbreviated <code>dp</code>
<code>(check-and-display-proof . opt-proof)</code>	abbreviated <code>cdp</code>
<code>(display-pterm . opt-proof)</code>	abbreviated <code>dpt</code>
<code>(display-proof-expr . opt-proof)</code>	abbreviated <code>dpe</code>

We also provide versions which normalize the proof first:

<code>(display-normalized-proof . opt-proof)</code>	abbreviated <code>dnp</code>
<code>(display-normalized-pterm . opt-proof)</code>	abbreviated <code>dnpt</code>
<code>(display-normalized-proof-expr . opt-proof)</code>	abbreviated <code>dnpe</code>

9.5. Classical logic. `(proof-of-stab-at formula)` generates a proof of $((A \rightarrow F) \rightarrow F) \rightarrow A$. For F, T one takes the obvious proof, and for other atomic formulas the proof using cases on booleans. For all other prime or existential formulas one takes an instance of the global assumption **Stab**: $((\hat{P} \rightarrow F) \rightarrow F) \rightarrow \hat{P}$. Here the argument *formula* must be unfolded. For the logical form of falsity we take `(proof-of-stab-log-at formula)`, and similary for ex-falso-quodlibet we provide

`(proof-of-efq-at formula)`
`(proof-of-efq-log-at formula)`

Using these functions we can then define `(reduce-efq-and-stab proof)`, which reduces all instances of stability and ex-falso-quodlibet axioms in a proof to instances of these global assumptions with prime or existential formulas, or (if possible) replaces them by their proofs.

With `rm-exc` we can transform a proof involving classical existential quantifiers in another one without, i.e. in minimal logic. The `Exc-Intro` and `Exc-Elim` theorems are replaced by their proofs, using `expand-theorems`.

10. INTERACTIVE THEOREM PROVING WITH PARTIAL PROOFS

Pproof

10.1. Partial proofs. A partial proof is a proof with holes, i.e. special assumption variables (called goal variables) `v`, `v1`, `v2` ... whose formulas must be closed. We assume that every goal variable `v` has a single occurrence in the proof. We then select a (not necessarily maximal) subproof `vx1...xn` with distinct object or assumption variables `x1...xn`. Such a subproof is called a *goal*. When interactively developing a partial proof, a goal `vx1...xn` is replaced by another partial proof, whose context is a subset of `x1...xn` (i.e. the context of the goal with `v` removed).

To gain some flexibility when working on our goals, we do not at each step of an interactive proof development traverse the partial proof searching for the remaining goals, but rather keep a list of all open goals together with their numbers as we go along. We maintain a global variable `PPROOF-STATE` containing a list of three elements: (1) `num-goals`, an alist of entries (`number goal drop-info hypname-info`), (2) `proof` and (3) `maxgoal`, the maximal goal number used.

At each stage of an interactive proof development we have access to the current proof and the current goal by executing

```
(current-proof)
(current-goal)
```

10.2. Interactive theorem proving. For interactively building proofs we need

```
(goal-to-goalvar goal)
(goal-to-context goal)
(goal-to-formula goal)
(goal=? proof goal)
(goal-subst proof goal proof1)
(pproof-state-to-num-goals)
(pproof-state-to-proof)
(pproof-state-to-formula)
(display-current-goal)
(display-current-goal-with-normalized-formulas)
(display-current-pproof-state)
```

We list some commands for interactively building proofs.

10.2.1. *set-goal*. An interactive proof starts with `(set-goal formula)`, i.e. with setting a goal. Here *formula* should be closed; if it is not, universal quantifiers are inserted automatically.

10.2.2. *normalize-goal*. `(normalize-goal goal)` (abbreviated `ng`) replaces every formula in the goal by its normal form.

10.2.3. *assume*. With `(assume x1 ...)` we can move universally quantified variables and hypotheses into the context. The variables must be given names (known to the parser as valid variable names for the given type), and the hypotheses should be identified by numbers or strings.

10.2.4. *use*. In `(use x . elab-path-and-terms)`, x is

- a number or string identifying a hypothesis from the context,
- the string “Truth-Axiom”,
- the name of a theorem or global assumption. If it is a global assumption whose final conclusion is a nullary predicate variable distinct from \perp (e.g. `Efq-Log` or `Stab-Log`), this predicate variable is substituted by the goal formula.
- a closed proof,
- a formula with free variables from the context, generating a new goal.

The optional *elab-path-and-terms* is a list consisting of symbols `left` or `right`, giving directions in case the used formula contains conjunctions, and of terms. The universal quantifiers of the used formula are instantiated (via `pattern-unify`) with appropriate terms in case a conclusion has the form of the goal. The terms provided are substituted for those variables that cannot be instantiated by pattern unification (e.g. using $\forall x.Px \rightarrow \perp$ for the goal \perp). For the instantiated premises new goals are created.

10.2.5. *use-with*. This is a more verbose form of *use*, where the terms are not inferred via unification, but have to be given explicitly. Also, for the instantiated premises one can indicate how they are to come about. So in `(use-with x . x-list)`, x is as in *use*, and *x-list* is a list consisting of

- a number or string identifying a hypothesis from the context,
- the name of a theorem or global assumption,
- a closed proof,
- the string “?” (value of `DEFAULT-GOAL-NAME`), generating a new goal,
- a symbol `left` or `right`,
- a term, whose free variables are added to the context,
- a type, which is substituted for the first type variable,
- a comprehension term, which is substituted for the first predicate variable.

Notice that new free variables not in the ordered context can be introduced in *use-with*. They will be present in the newly generated goals. The reason is that proofs should be allowed to contain free variables. This is necessary to allow logic in ground types where no constant is available (e.g. to prove $\forall x.Px \rightarrow \forall x.\neg Px \rightarrow \perp$).

Notice also that there are situations where *use-with* can be applied but *use* can not. For an example, consider the goal $P(S(k+l))$ with the hypothesis $\forall l.P(k+l)$ in the context. Then *use* cannot find the term Sl by matching; however, applying *use-with* to the hyposthesis and the term Sl succeeds (since $k+Sl$ and $S(k+l)$ have the same normal form).

10.2.6. *inst-with*. **inst-with** does for forward chaining the same as **use-with** for backward chaining. It replaces the present goal by a new one, with one additional hypothesis obtained by instantiating a previous one. Notice that this effect could also be obtained by **cut**. In **(inst-with x . x-list)**, *x* is

- a number or string identifying a hypothesis from the context,
- the name of a theorem or global assumption,
- a closed proof,
- a formula with free variables from the context, generating a new goal.

and *x-list* is a list consisting of

- a number or string identifying a hypothesis from the context,
- the name of a theorem or global assumption,
- a closed proof,
- the string “?” (value of **DEFAULT-GOAL-NAME**), generating a new goal,
- a symbol **left** or **right**,
- a term, whose free variables are added to the context,
- a type, which is substituted for the first type variable,
- a comprehension term, which is substituted for the first predicate variable.

10.2.7. *inst-with-to*. **inst-with-to** expects a string as its last argument, which is used (via **name-hyp**) to name the newly introduced instantiated hypothesis.

10.2.8. *cut*. The command **(cut A)** replaces the goal *B* by the two new goals *A* and $A \rightarrow B$.

10.2.9. *strip*. To move (all or *n*) universally quantified variables and hypotheses of the current goal into the context, we use the command **(strip)** or **(strip n)**.

10.2.10. *drop*. In **(drop . x-list)**, *x-list* is a list of numbers or strings identifying hypotheses from the context. A new goal is created, which differs from the previous one only in display aspects: the listed hypotheses are hidden (but still present). If *x-list* is empty, all hypotheses are hidden.

10.2.11. *name-hyp*. The command **name-hyp** expects an index *i* and a string. Then a new goal is created, which differs from the previous one only in display aspects: the string is used to label the *i*th hypothesis.

10.2.12. *split*. The command **(split)** expects a conjunction $A \wedge B$ as goal and splits it into the two new goals *A* and *B*.

10.2.13. *get*. To be able to work on a goal different from that on top of the goal stack, we have to move it up using **(get n)**.

10.2.14. *undo*. With **(undo . n)**, the last *n* steps of an interactive proof can be made undone. **(undo)** has the same effect as **(undo 1)**.

10.2.15. *ind.* (**ind**) expects a goal $\forall x^\rho A$ with ρ an algebra. Let c_1, \dots, c_n be the constructors of the algebra ρ . Then n new goals $\forall \vec{x}_i. A[x:=x_{1i}] \rightarrow \dots \rightarrow A[x:=x_{ki}] \rightarrow A[x:=c_i \vec{x}_i]$ are generated.

(**ind t**) expects a goal $A[x:=t]$. It computes the algebra ρ as type of the term t . Then again n new goals $\forall \vec{x}_i. A[x:=x_{1i}] \rightarrow \dots \rightarrow A[x:=x_{ki}] \rightarrow A[x:=c_i \vec{x}_i]$ are generated.

10.2.16. *simind.* (**simind all-formula1 ...**) also expects a goal $\forall x^\rho A$ with ρ an algebra. Then we have to provide the other all formulas to be proved simultaneously with the given one.

10.2.17. *intro.* (**intro i . terms**) expects as goal an inductively defined predicate. The i -th introduction axiom for this predicate is applied, via **use** (hence **terms** may have to be provided).

10.2.18. *elim.* (**elim**) expects a goal $I(\vec{t}) \rightarrow A[\vec{x}:=\vec{t}]$. Then the (strengthened) clauses are generated as new goals, via **use-with**.

10.2.19. *ex-intro.* In (**ex-intro term**), the user provides a term to be used for the present (existential) goal. (**exnc-intro x**) works similarly for the **exnc**-quantifier.

10.2.20. *ex-elim.* In (**ex-elim x**), x is

- a number or string identifying an existential hypothesis from the context,
- the name of an existential global assumption or theorem,
- a closed proof on an existential formula,
- an existential formula with free variables from the context, generating a new goal.

Let $\exists y A$ be the existential formula identified by x . The user is then asked to provide a proof for the present goal, assuming that a y satisfying A is available. (**exnc-elim x**) works similarly for the **exnc**-quantifier.

10.2.21. *by-assume-with.* Suppose we are proving a goal G from an existential hypothesis $ExHyp: \exists y A$. Then the natural way to use this hypothesis is to say “by $ExHyp$ assume we have a y satisfying A ”. Correspondingly we provide (**by-assume-with x y u**). Here x – as in **ex-elim** – identifies an existential hypothesis, and we assume (i.e. add to the context) the variable y and – with label u – the kernel A . (**by-assume-with x y u**) is implemented by the sequence (**ex-elim x**), (**assume y u**), (**drop x**). **by-exnc-assume-with** works similarly for the **exnc**-quantifier.

10.2.22. *cases.* (**cases**) expects a goal $\forall x^\rho A$ with ρ an algebra. Assume that c_1, \dots, c_n are the constructors of the algebra ρ . Then n new (simplified) goals $\forall \vec{x}_i. A[x:=c_i \vec{x}_i]$ are generated.

(**cases t**) expects a goal $A[x:=t]$. It computes the algebra ρ as type of the term t . Then again n new goals $\forall \vec{x}_i. A[x:=c_i \vec{x}_i]$ are generated.

(**cases 'auto**) expects an atomic goal and checks whether its boolean kernel contains an if-term whose test is neither an if-term nor contains bound variables. With the first such test (**cases test**) is called.

10.2.23. *casedist*. (`casedist t`) replaces the goal A containing a boolean term t by two new goals $(\text{atom } t) \rightarrow A[t:=\text{tt}]$ and $((\text{atom } t) \rightarrow \text{ff}) \rightarrow A[t:=\text{ff}]$.

10.2.24. *simp*. In (`simp opt-dir x . elab-path-and-terms`), the optional argument *opt-dir* is either the string “<-” or missing. x is

- a number or string identifying a hypothesis from the context,
- the name of a theorem or global assumption,
- a closed proof,
- a formula with free variables from the context, generating a new goal.

The optional *elab-path-and-terms* is a list consisting of symbols `left` or `right`, giving directions in case the used formula contains conjunctions, and of terms. The universal quantifiers of the used formula are instantiated with appropriate terms to match a part of the goal. The terms provided are substituted for those variables that cannot be inferred. For the instantiated premises new goals are created. This generates a used formula, which is to be an atom, a negated atom or $t \approx s$. If it is a (negated) atom, it is checked whether the kernel or its normal form is present in the goal. If so, it is replaced by `T` (or `F`). If it is an equality $t = s$ or $t \approx s$ with t or its normal form present in the goal, t is replaced by s . In case “<-” exchange t and s .

10.2.25. *simp-with*. This is a more verbose form of `simp`, where the terms are not inferred via matching, but have to be given explicitly. Also, for the instantiated premises one can indicate how they are to come about. So in (`simp-with opt-dir x . x-list`), *opt-dir* and x are as in `simp`, and *x-list* is a list consisting of

- a number or string identifying a hypothesis from the context,
- the name of a theorem or global assumption,
- a closed proof,
- the string “?” (value of `DEFAULT-GOAL-NAME`), generating a new goal,
- a symbol `left` or `right`,
- a term, whose free variables are added to the context,
- a type, which is substituted for the first type variable,
- a comprehension term, which is substituted for the first predicate variable.

10.2.26. *min-pr*. In (`min-pr x measure`), x is

- a number or string identifying a classical existential hypothesis from the context,
- the name of a classical existential global assumption or theorem,
- a closed proof on a classical existential formula,
- a classical existential formula with free variables from the context, generating a new goal.

The result is a new implicational goal, whose premise provides the (classical) existence of instances with least measure.

10.2.27. *exc-intro*. In (`exc-intro terms`), the user provides terms to be used for the present (classical existential) goal.

10.2.28. *exc-elim*. In `(exc-elim x)`, x is

- a number or string identifying a classical existential hypothesis from the context,
- the name of a classical existential global assumption or theorem,
- a closed proof on a classical existential formula,
- a classical existential formula with free variables from the context, generating a new goal.

Let $\exists^{\text{ca}} \vec{y} \vec{A}$ or $\exists^{\text{cl}} \vec{y} \vec{A}$ be the classical existential formula identified by x . The user is then asked to provide a proof for the present goal, assuming that terms \vec{y} satisfying \vec{A} are available.

10.2.29. *pair-elim*. In `(pair-elim)`, a goal $\forall p P(p)$ is replaced by the new goal $\forall x_1, x_2 P(\langle x_1, x_2 \rangle)$.

11. SEARCH

Following MILLER [10] and BERGER, we have implemented a proof search algorithm for minimal logic. To enforce termination, every assumption can only be used a fixed number of times.

We begin with a short description of the theory involved.

Q always denotes a $\forall\exists\forall$ -prefix, say $\forall \vec{x} \exists \vec{y} \forall \vec{z}$, with distinct variables. We call \vec{x} the *signature variables*, \vec{y} the *flexible variables* and \vec{z} the *forbidden variables* of Q , and write Q_{\exists} for the existential part $\exists \vec{y}$ of Q .

Q -terms are inductively defined by the following clauses.

- If u is a universally quantified variable in Q or a constant, and \vec{r} are Q -terms, then $u\vec{r}$ is a Q -term.
- For any flexible variable y and distinct forbidden variables \vec{z} from Q , $y\vec{z}$ is a Q -term.
- If r is a $Q\forall z$ -term, then $\lambda z r$ is a Q -term.

Explicitly, r is a Q -term iff all its free variables are in Q , and for every subterm $y\vec{r}$ of r with y free in r and flexible in Q , the \vec{r} are distinct variables either λ -bound in r (such that $y\vec{r}$ is in the scope of this λ) or else forbidden in Q .

Q -goals and Q -clauses are simultaneously defined by

- If \vec{r} are Q -terms, then $P\vec{r}$ is a Q -goal as well as a Q -clause.
- If D is a Q -clause and G is a Q -goal, then $D \rightarrow G$ is a Q -goal.
- If G is a Q -goal and D is a Q -clause, then $G \rightarrow D$ is a Q -clause.
- If G is a $Q\forall x$ -goal, then $\forall x G$ is a Q -goal.
- If $D[y:=Y\vec{z}]$ is a $\forall \vec{x} \exists \vec{y}, Y\forall \vec{z}$ -clause, then $\forall y D$ is a $\forall \vec{x} \exists \vec{y} \forall \vec{z}$ -clause.

Explicitly, a formula A is a Q -goal iff all its free variables are in Q , and for every subterm $y\vec{r}$ of A with y either existentially bound in A (with $y\vec{r}$ in the scope) or else free in A and flexible in Q , the \vec{r} are distinct variables either λ - or universally bound in A (such that $y\vec{r}$ is in the scope) or else free in A and forbidden in Q .

A Q -sequent has the form $\mathcal{P} \Rightarrow G$, where \mathcal{P} is a list of Q -clauses and G is a Q -goal.

A Q -substitution is a substitution of Q -terms.

A *unification problem* \mathcal{U} consists of a $\forall\exists\forall$ -prefix Q and a conjunction C of equations between Q -terms of the same type, i.e. $\bigwedge_{i=1}^n r_i = s_i$. We may assume that each such equation is of the form $\lambda\vec{x}r = \lambda\vec{x}s$ with the same \vec{x} (which may be empty) and r, s of ground type.

A *solution* to such a unification problem \mathcal{U} is a Q -substitution φ such that for every i , $r_i\varphi = s_i\varphi$ holds (i.e. $r_i\varphi$ and $s_i\varphi$ have the same normal form). We sometimes write C as $\vec{r} = \vec{s}$, and (for obvious reasons) call it a list of unification pairs.

We work with lists of sequents instead of single sequents; they all are Q -sequents for the same prefix Q . One then searches for a Q -substitution φ and proofs of the φ -substituted sequents. **intro-search** takes the first sequent and extends Q by all universally quantified variables $x_1 \dots$. It then calls **select**, which selects (using **or**) a fitting clause. If one is found, a new prefix Q' (raising the new flexible variables) is formed, and the n (≥ 0) new goals with their clauses (and also all remaining sequents) are substituted with $\mathbf{star} \circ \rho$, where **star** is the “raising” substitution and ρ is the mgu. For this constellation **intro-search** is called again. In case of success, one obtains a Q' -substitution φ' and proofs of the $\mathbf{star} \circ \rho \circ \varphi'$ -substituted new sequents. Let $\varphi := (\rho \circ \varphi') \upharpoonright Q_\exists$, and take the first n proofs of these to build a proof of the φ -substituted (first) sequent originally considered by **intro-search**.

Compared with Miller [10], we make use of several simplifications, optimizations and extensions, in particular the following.

- Instead of arbitrarily mixed prefixes we only use those of the form $\forall\exists\forall$. Nipkow in [11] already had presented a version of Miller’s pattern unification algorithm for such prefixes, and Miller in [10, Section 9.2] notes that in such a situation any two unifiers can be transformed into each other by a variable renaming substitution. Here we restrict ourselves to $\forall\exists\forall$ -prefixes throughout, i.e. in the proof search algorithm as well.
- The order of events in the pattern unification algorithm is changed slightly, by postponing the raising step until it is really needed. This avoids unnecessary creation of new higher type variables. – Already Miller noted in [10, p.515] that such optimizations are possible.
- The extensions concern the (strong) existential quantifier, which has been left out in Miller’s treatment, and also conjunction. The latter can be avoided in principle, but of course is a useful thing to have.

(**search** m (**name1** $m1$) ...) expects for m a default value for multiplicity (i.e. how often assumptions are to be used), for **name1** ...

- numbers of hypotheses from the present context or
- names for theorems or global assumptions,

and for $m1 \dots$ multiplicities (positive integers for global assumptions or theorems). A search is started for a proof of the goal formula from the given hypotheses with the given multiplicities and in addition from the other hypotheses (but not any other global assumptions or theorems) with m or **mult-default**. To exclude a hypothesis from being tried, list it with multiplicity 0.

12. COMPUTATIONAL CONTENT OF CLASSICAL PROOFS

Classical

This section is based on [3]. We restrict to formulas in the language $\{\perp, \rightarrow, \forall\}$ in this section, and - as in the paper - make use of a special nullary predicate variable X .

A formula is *relevant* if it ends with (logical) falsity. *Definite* and *goal* formulas are defined by a simultaneous recursion, as in [3].

(atr-relevant? *formula*)

(atr-definite? *formula*)

(atr-goal? *formula*)

To implement [3, Lemma 3.1], we need to construct proofs from formulas:

$N_D: ((D \rightarrow \perp) \rightarrow X) \rightarrow D^X$ for D relevant

$M_D: D \rightarrow D^X$

$K_G: G \rightarrow G^X$ for G irrelevant

$H_G: G^X \rightarrow (G \rightarrow X) \rightarrow X$

This is done by

(atr-rel-definite-proof *formula*)

(atr-arb-definite-proof *formula*)

(atr-irrel-goal-proof *formula*)

(atr-arb-goal-proof *formula*)

Next we need to implement [3, Lemma 3.2], which says that for goal formulas $\vec{G} = G_1, \dots, G_n$ we can derive in minimal logic augmented with a special predicate variable X

$$(\vec{G} \rightarrow X) \rightarrow \vec{G}^X \rightarrow X.$$

In our implementation this function is called

(atr-goals-to-x-proof *goal1 ...*)

Finally we implement [3, Theorem 3.3], which says the following. Assume that for definite formulas \vec{D} and goal formulas \vec{G} we can derive in minimal logic

$$\vec{D} \rightarrow (\forall \vec{y}. \vec{G} \rightarrow \perp) \rightarrow \perp.$$

Then we can also derive in intuitionistic logic augmented with the special predicate variable X

$$\vec{D} \rightarrow (\forall \vec{y}. \vec{G} \rightarrow X) \rightarrow X.$$

In particular, substitution of the formula

$$\exists \vec{y}. \vec{G} := \exists \vec{y}. G_1 \wedge \dots \wedge G_n$$

for X yields a derivation in intuitionistic logic of

$$\vec{D} \rightarrow \exists \vec{y}. \vec{G}.$$

This is done by

(atr-min-excl-proof-to-x-proof *min-excl-proof*)

(atr-min-excl-proof-to-intuit-ex-proof *min-excl-proof*)

See section 13 for an interpretation of the symbols of the extracted terms in Minlog's output.

13. EXTRACTED TERMS

We assign to every formula A an object $\tau(A)$ (a type or the symbol `nulltype`). $\tau(A)$ is intended to be the type of the program to be extracted from a proof of A . This is done by

`(formula-to-et-type formula)`

In `formula-to-et-type` we assign type variables to the predicate variables. For to be able to later refer to this assignment, we use a global variable `PVAR-TO-TVAR-ALIST`, which memorizes the assignment done so far. Later reference is necessary, because such type variables will appear in extracted programs of theorems involving predicate variables, and in a given development there may be many auxiliary lemmata containing the same predicate variable. A fixed `pvar-to-tvar` refers to and updates `PVAR-TO-TVAR-ALIST`.

When we want to execute the program, we have to replace the constant `cL` corresponding to a lemma `L` by the extracted program of its proof, and the constant `cGA` corresponding to a global assumption `GA` by an assumed extracted term to be provided by the user. This can be achieved by adding computation rules for `cL` and `cGA`. We can be rather flexible here and enable/block rewriting by using `animate/deanimate` as desired. Notice that the type of the extracted term provided for a `cGA` must be the extracted type of the assumed formula. When predicate variables are present, one must use the type variables assigned to them in `PVAR-TO-TVAR-ALIST`.

`(animate thm-or-ga-name . opt-eterm)`
`(deanimate thm-or-ga-name)`

We can define, for a given derivation M of a formula A with $\tau(A) \neq \text{nulltype}$, its *extracted term* (or *extracted program*) $\text{et}(M)$ of type $\tau(A)$. We also need extracted terms for the axioms. For induction we take recursion, for the proof-by-cases axiom we take the cases-construct for terms; for the other axioms the extracted terms are rather clear. Term extraction is implemented by

`(proof-to-extracted-term proof)`

The following table gives the symbols of Minlog's output and the corresponding notation in the λ -calculus.

Explanation	Symbol	Minlog's Output
λ -abstraction:	$\lambda x.M$	<code>([x]M)</code>
pair:	$\langle M \mid N \rangle$	<code>(M@N)</code>
left element of a pair:	$(M \ 0)$	<code>left M</code>
right element of a pair:	$(M \ 1)$	<code>right M</code>
arrow for types:	\rightarrow	<code>=></code>
product for types:	\times	<code>@@</code>
recursion operator:	\mathcal{R}	<code>\mathcal{R}</code>

It is also possible to give an internal proof of soundness. This can be done by

(proof-to-soundness-proof *proof*)

14. READING FORMULAS IN EXTERNAL FORM

Reading

A formula can be produced from an external representation, for example a string, using the `pt` function. It has one argument, a string denoting a formula, that is converted to the internal representation of the formula. For the following syntactical entities parsing functions are provided:

(py *string*) for parsing types
 (pv *string*) for parsing variables
 (pt *string*) for parsing terms
 (pf *string*) for parsing formulas

The conversion occurs in two steps: lexical analysis and parsing.

14.1. Lexical analysis. In this stage the string is broken into short sequences, called *tokens*.

A token can be one of the following:

- An alphabetic symbol: A sequence of letters `a-z` and `A-Z`. Upper and lower case letters are considered different.
- A number: A sequence of digits `0-9`
- A punctuation mark: One of the characters: `() [] . , ;`
- A special symbol: A sequence of characters, that are neither letters, digits, punctuation marks nor white space.

For example: `abc`, `ABC` and `A` are alphabetic symbols, `123`, `0123` and `7` are numbers, `(` is a punctuation mark, and `<=`, `+`, and `#:-^` are special symbols.

Tokens are always character sequences of maximal length belonging to one of the above categories. Therefore `fx` is a single alphabetic symbol not two and likewise `<+` is a single special symbol. The sequence `alpha<=(-x+z)`, however, consists of the 8 tokens `alpha`, `<=`, `(`, `-`, `x`, `+`, `z`, and `)`. Note that the special symbols `<=` and `-` are separated by a punctuation mark, and the alphabetic symbols `x` and `z` are separated by the special symbol `+`.

If two alphabetic symbols, two special symbols, or two numbers follow each other they need to be separated by white space (spaces, newlines, tabs, formfeeds, etc.). Except for a few situations mentioned below, whitespace has no significance other than separating tokens. It can be inserted and removed between any two tokens without affecting the significance of the string.

Every token has a *token type*, and a value. The token type is one of the following: number, var-index, var-name, const, pvar-name, predconst, type-symbol, pscheme-symbol, postfix-op, prefix-op, binding-op, add-op, mul-op, rel-op, and-op, or-op, imp-op, pair-op, if-op, postfix-jct, prefix-jct, and-jct, or-jct, tensor-jct, imp-jct, quantor, dot, hat, underscore, comma, semicolon, arrow, lpar, rpar, lbracket, rbracket.

The possible values for a token depend on the token type and are explained below.

New tokens can be added using the function

`(add-token string token-type value).`

The inverse is the function

`(remove-token string).`

A list of all currently defined tokens sorted by token types can be obtained by the function

`(display-tokens).`

14.2. Parsing. The second stage, *parsing*, extracts structure from the sequence of tokens.

Types. Type-symbols are types; the value of a type-symbol must be a type. If σ and τ are types, then $\sigma;\tau$ is a type (pair type) and $\sigma\Rightarrow\tau$ is a type (function type). Parentheses can be used to indicate proper nesting. For example `boole` is a predefined type-symbol and hence, `(boole;boole) \Rightarrow boole` is again a type. The parentheses in this case are not strictly necessary, since `;` binds stronger than `\Rightarrow` . Both operators associate to the right.

Variables. Var-names are variables; the value of a var-name token must be a pair consisting of the type and the name of the variable (the same name string again! This is not nice and may be later, we find a way to give the parser access to the string that is already implicit in the token). For example to add a new boolean variable called “flag”, you have to invoke the function `(add-token "flag" 'var-name (cons 'boole "flag"))`. This will enable the parser to recognize “flag3”, “flag[^]”, or “flag[^]14” as well.

Further, types, as defined above, can be used to construct variables.

A variable given by a name or a type can be further modified. If it is followed by a [^], a partial variable is constructed. Instead of the [^] a _{_} can be used to specify a total variable.

Total variables are the default and therefore, the _{_} can be omitted.

As another modifier, a number can immediately follow, with no whitespace in between, the [^] or the _{_}, specifying a specific variable index.

In the case of indexed total variables given by a variable name or a type symbol, again the _{_} can be omitted. The number must then follow, with no whitespace in between, directly after the variable name or the type.

Note: This is the only place where whitespace is of any significance in the input. If the [^], _{_}, type name or variable name is separated from the following number by whitespace, this number is no longer considered to be an index for that variable but a numeric term in its own right.

For example, assuming that `p` is declared as a variable of type `boole`, we have:

- `p` a total variable of type `boole` with name `p` and no index.
- `p_` a total variable of type `boole` with name `p` and no index.
- `p^` a partial variable of type `boole` with name `p` and no index.
- `p2` a total variable of type `boole` with name `p` and index 2.
- `p2` a total variable of type `boole` with name `p` and index 2.
- `p^2` a partial variable of type `boole` with name `p` and index 2.
- `boole` a total anonymous variable of type `boole` with no index.
- `boole_` a total anonymous variable of type `boole` with no index.

- `boole^` a partial anonymous variable of type `boole` with no index.
- `boole_2` a total anonymous variable of type `boole` with index 2.
- `boole2` a total anonymous variable of type `boole` with index 2.
- `boole^2` a partial anonymous variable of type `boole` with index 2.
- `boole 2` a total anonymous variable of type `boole` applied to the numeric term 2.
- `(boole)2` a total anonymous variable of type `boole` applied to the numeric term 2.
- `(boole)_2` a total anonymous variable of type `boole` with index 2.
- `boole=>boole^2` a partial anonymous variable of type function of `boole` to `boole` with index 2.

Terms are built from atomic terms using application and operators.

An atomic term is one of the following: a constant, a variable, a number, a conditional, or any other term enclosed in parentheses.

Constants have `const` as token type, and the respective term in internal form as value. There are also composed constants, so-called *constant schemata*. A constant schema has the form of the name of the constant schema (token type `constscheme`) followed by a list of types, the whole thing enclosed in parentheses. There are a few built in constant schemata: `(Rec <typelist>)` is the recursion over the types given in the type list; `(EQat <type>)` is the equality for the given type; `(Eat <type>)` is the existence predicate for the given type. The constant schema `EQat` can also be written as the relational infix operator `=`; the constant schemata `Eat` can also be written as the prefix operator `E`.

For a number, the user defined function `make-numeric-term` is called with the number as argument. The return value of `make-numeric-term` should be the internal term representation of the number.

To form a conditional term, the if operator `if` followed by a list of atomic terms is enclosed in square brackets. Depending on the constructor of the first term, the selector, a conditional term can be reduced to one of the remaining terms.

From these atomic terms, compound terms are built not only by application but also using a variety of operators, that differ in binding strength and associativity.

Postfix operators (token type `postfix-op`) bind strongest, next in binding strength are prefix operators (token type `prefix-op`), next come binding operators (token type `binding-op`).

A binding operator is followed by a list of variables and finally a term. There are two more variations of binding operators, that bind much weaker and are discussed later.

Next, after the binding operators, is plain application. Juxtaposition of two terms means applying the first term to the second. Sequences of applications associate to the left. According to the *vector notation* convention the meaning of application depends on the type of the first term. Two forms of applications are defined by default: if the type of the first term is of `arrow-form?` then `make-term-in-app-form` is used; for the type of a free algebra we use the corresponding form of recursion. However, there is one exception: if the first term is of type `boole` application is read as a short-hand

for the “if...then...else” construct (which is a special form) rather than boolean recursion. The user may use the function `add-new-application` to add new forms of applications. This function takes two arguments, a predicate for the type of the first argument, and a function taking the two terms and returning another term intended to be the result of this form of application. Predicates are tested in the inverse order of their definition, so more general forms of applications should be added first.

By default these new forms of application are *not* used for output; but the user might declare that certain terms should be output as formal application. *When doing so it is the user’s responsibility to make sure that the syntax used for the output can still be parsed correctly by the parser!* To do so the function `(add-new-application-syntax pred toarg toop)` can be used, where the first argument has to be a predicate (i.e., a function mapping terms to `#t` and `#f`) telling whether this special form of application can be used. If so, the arguments `toarg` and `toop` have to be functions mapping the term to operator and argument of this “application” respectively.

After that, we have binary operators written in infix notation. In order of decreasing binding strength these are:

- multiplicative operators, leftassociative, token type `mul-op`;
- additive operators, leftassociative, token type `add-op`;
- relational operators, not associative, token type `rel-op`;
- boolean and operators, leftassociative, token type `and-op`;
- boolean or operators, leftassociative, token type `or-op`;
- boolean implication operators, rightassociative, token type `imp-op`;
- pairing operators, rightassociative, token type `pair-op`.

On the top level, we have two more forms of binding operators, one using the dot “.”, the other using square brackets “[]”. Recall that a binding operator is followed by a list of variables and a term. This notation can be augmented by a period “.” following after the variable list and before the term. In this case the scope of the binding extends as far to the right as possible. Bindings with the lambda operator can also be specified by including the list of variables in square brackets. In this case, again, the scope of the binding extends as far as possible.

Predefined operators are `E` and `=` as described above, the binding operator `lambda`, and the pairing operator `@` with two prefix operators `left` and `right` for projection.

The value of an operator token is a function that will obtain the internal representation of the component terms as arguments and returns the internal representation of the whole term.

If a term is formed by application, the function `make-gen-application` is called with two subterms and returns the compound term. The default here (for terms with an arrow type) is to make a term in application form. However other rules of composition might be introduced easily.

Formulas are built from atomic formulas using junctors and quantors.

The simplest atomic formulas are made from terms using the implicit predicate “atom”. The semantics of this predicate is well defined only for terms of type `boole`. Further, a predicate constant (token type `predconst`) or a predicate variable (token type `pvar`) followed by a list of atomic terms is

an atomic formula. Lastly, any formula enclosed in parentheses is considered an atomic formula.

The composition of formulas using junctors and quantors is very similar to the composition of terms using operators and binding. So, first postfix junctors, token type `postfix-jct`, are applied, next prefix junctors, token type `prefix-jct`, and quantors, token type `quantor`, in the usual form: quantor, list of variables, formula. Again, we have a notation using a period after the list of variables, making the scope of the quantor as large as possible. Predefined quantors are `ex`, `excl`, `exca`, and `all`.

The remaining junctors are binary junctors written in infix form. In order of decreasing binding strength we have:

- conjunction junctors, leftassociative, token type `and-jct`;
- disjunction junctors, leftassociative, token type `or-jct`;
- tensor junctors, rightassociative, token type `tensor-jct`;
- implication junctors, rightassociative, token type `imp-jct`.

Predefined junctors are `&` (and), `!` (tensor), and `->` (implication).

The value of junctors and quantors is a function that will be called with the appropriate subformulas, respectively variable lists, to produce the compound formula in internal form.

REFERENCES

- [1] Klaus Aehlig and Helmut Schwichtenberg, *A syntactical analysis of non-size-increasing polynomial time computation*, Proceedings 15'th Symposium on Logic in Computer Science (LICS 2000), 2000, pp. 84–91. 2.2
- [2] Ulrich Berger, *Program extraction from normalization proofs*, Typed Lambda Calculi and Applications (M. Bezem and J.F. Groote, eds.), LNCS, vol. 664, Springer Verlag, Berlin, Heidelberg, New York, 1993, pp. 91–106. 1.6
- [3] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg, *Refined program extraction from classical proofs*, Annals of Pure and Applied Logic **114** (2002), 3–25. 1, 12
- [4] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg, *Term rewriting for normalization by evaluation*, Information and Computation **183** (2003), 19–42. 1.1, 2.2, 4.1, 6.1
- [5] Ulrich Berger and Helmut Schwichtenberg, *An inverse of the evaluation functional for typed λ -calculus*, Proceedings 6'th Symposium on Logic in Computer Science (LICS'91) (R. Vemuri, ed.), IEEE Computer Society Press, Los Alamitos, 1991, pp. 203–211. 6.1
- [6] Stefan Berghofer, *Proofs, programs and executable specifications in higher order logic*, Ph.D. thesis, Institut für Informatik, TU München, 2003. 1.6
- [7] Martin Hofmann, *Linear types and non-size-increasing polynomial time computation*, Proceedings 14'th Symposium on Logic in Computer Science (LICS'99), 1999, pp. 464–473. 2.2
- [8] Felix Joachimski and Ralph Matthes, *Short proofs of normalisation for the simply-typed λ -calculus, permutative conversions and Gödel's T*, Archive for Mathematical Logic **42** (2003), 59–87. 6
- [9] Ralph Matthes, *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*, Ph.D. thesis, Mathematisches Institut der Universität München, 1998. 8.1
- [10] Dale Miller, *A logic programming language with lambda-abstraction, function variables and simple unification*, Journal of Logic and Computation **2** (1991), no. 4, 497–536. 1.6, 11

- [11] Tobias Nipkow, *Higher-order critical pairs*, Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (Los Alamitos) (R. Vemuri, ed.), IEEE Computer Society Press, 1991, pp. 342–349. 11
- [12] Viggo Stoltenberg-Hansen, Edward Griffor, and Ingrid Lindström, *Mathematical theory of domains*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1994. 1.1, 1.2, 3
- [13] Anne S. Troelstra and Helmut Schwichtenberg, *Basic proof theory*, 2nd ed., Cambridge University Press, 2000. 1.5

INDEX

==to-E-1, 32
 ==to-E-2, 32
 ==to-Eq, 32

 aconst-to-formula, 30
 aconst-to-inst-formula, 30
 aconst-to-kind, 30
 aconst-to-name, 30
 aconst-to-repro-formulas, 30
 aconst-to-string, 30
 aconst-to-tpsubst, 30
 aconst-to-uninst-formula, 30
 aconst-without-rules?, 30
 aconst=?, 30
 aconst?, 30
 add-alg, 8
 add-algebras-with-parameters, 8
 add-algs, 8
 add-computation-rule, 17
 add-global-assumption, 36
 add-ids, 19
 add-new-application, 53
 add-param-alg, 8
 add-param-algs, 8
 add-predconst-name, 19
 add-program-constant, 17
 add-pvar-name, 18
 add-rewrite-rule, 17
 add-theorem, 34
 add-tvar-name, 7
 add-var-name, 10
 aga, 36
 alg-form-to-name, 8
 alg-form-to-types, 8
 alg-form?, 8
 alg-name-to-arity, 8
 alg-name-to-simalg-names, 8
 alg-name-to-token-types, 8
 alg-name-to-tvars, 8
 alg-name-to-typed-constr-names, 8
 alg?, 8
 All-AllPartial, 32
 all-allpartial-aconst, 32
 All-AllPartial-nat, 32
 all-form-to-kernel, 26
 all-form-to-var, 26
 all-form-to-vars-and..., 27
 all-form?, 25
 all-formula-to-cases-aconst, 34
 all-formulas-to-ind-aconst, 34
 allnc-form-to-kernel, 26
 allnc-form-to-var, 26
 allnc-form?, 25
 AllPartial-All, 32
 allpartial-all-aconst, 32

 AllPartial-All-nat,, 32
 and-form-to-left, 26
 and-form-to-right, 26
 and-form?, 25
 animate, 49
 animation, 23
 arity
 of a predicate variable, 17
 of a program constant, 12
 arity-to-string, 18
 arity-to-types, 18
 arrow-form-to-arg-type, 9
 arrow-form-to-arg-types, 9
 arrow-form-to-final-val-type, 9
 arrow-form-to-val-type, 9
 arrow-form?, 9
 arrow-types-to-rec-const, 17
 assume, 42
 assumption constant, 30
 asubst, 6
 Atom-False, 34
 atom-form-to-kernel, 25
 atom-form?, 25
 Atom-True, 34
 atr-arb-definite-proof, 48
 atr-arb-goal-proof, 48
 atr-definite?, 48
 atr-goal?, 48
 atr-goals-to-x-proof, 48
 atr-irrel-goal-proof, 48
 atr-min-...-to-intuit-ex-proof, 48
 atr-min-excl-proof-to-x-proof, 48
 atr-rel-definite-proof, 48
 atr-relevant?, 48
 avar-proof-equal?, 6
 avar-to-formula, 29
 avar-to-index, 29
 avar-to-name, 29
 avar=?, 6
 avar=?, 29
 avar?, 29

 Berger, 46
 bottom, 17
 by-assume-with, 44
 by-exnc-assume-with, 44

 Cases, 33, 34
 cases, 44
 cases-construct, 33
 Cases-Log, 35
 check-and-display-proof, 40
 classical-cterm=?, 28
 classical-formula=?, 27, 28
 clause, 19
 Compatibility, 31

- Compose, 15
- compose-o-substitutions, 24
- compose-substitutions, 24
- compose-substitutions-wrt, 6
- compose-t-substitutions, 6
- composition, 6
- comprehension term, 17, 25
- computation rule, 12
- computation rules, 12
- consistent-substitutions-wrt?, 6
- const-to-arrow-types-or..., 16
- const-to-kind, 16
- const-to-name, 16
- const-to-object-or-arity, 16
- const-to-t-deg, 16
- const-to-token-type, 16
- const-to-tsubst, 16
- const-to-tvars, 16
- const-to-type, 16
- const-to-uninst-type, 16
- const=?, 16
- const?, 16
- constant scheme, 52
- constr-name-and-tsubst..., 16
- constr-name-to-constr, 16
- constr-name?, 16
- Constr-Total, 31
- Constr-Total-Args, 31
- constructor, 16
- constructor pattern, 12
- context, 36
- context-to-avars, 38
- context-to-vars, 38
- context=?, 38
- conversion relation, 15
- Coquand, 4
- cterm-subst, 28
- cterm-substitute, 28
- cterm-to-formula, 28
- cterm-to-free, 28
- cterm-to-string, 28
- cterm-to-vars, 28
- cterm=?, 28
- cterm?, 28
- current-goal, 41
- current-proof, 41
- cut, 43
- Cvind-with-measure-11, 35
- deanimate, 49
- default-var-name, 10
- degree of totality, 10
- display-constructors, 16
- display-current-goal, 41
- display-current-goal-with..., 41
- display-current-num-goals..., 41
- display-global-assumptions, 36
- display-normalized-proof, 40
- display-normalized-proof-expr, 40
- display-normalized-pterm, 40
- display-p-substitution, 29
- display-program-constants, 17
- display-proof, 40
- display-proof-expr, 40
- display-pterm, 40
- display-substitution, 24
- display-t-substitution, 6
- display-theorems, 34
- dnp, 40
- dnpe, 40
- dnpt, 40
- cdp, 40
- dp, 40
- dpe, 40
- dpt, 40
- drop, 43
- E, 15
- E-to-Total, 32
- E-to-Total-nat, 31
- Efq, 36
- Efq-Atom, 34
- Efq-Log, 36
- Elim, 34
- elim, 44
- empty-subst, 5
- Eq, 15
- Eq-Compat, 32
- eq-compat-aconst, 32
- Eq-Ext, 32
- Eq-Refl, 31, 32
- eq-refl-aconst, 32
- Eq-Symm, 31, 32
- eq-symm-aconst, 32
- Eq-to--1, 32
- Eq-to--1-nat, 31
- Eq-to--2, 32
- Eq-to--2-nat, 31
- Eq-Trans, 31, 32
- eq-trans-aconst, 32
- equal-pvars?, 18
- Refl-nat, 34
- Symm-nat, 34
- Trans-nat, 34
- equality, 3
- to-E-1-nat, 31
- to-E-2-nat, 31
- to-Eq-nat, 31
- evaluation, 23
- Ex-Elim, 15, 30, 32
- Ex-Elim, 34
- ex-elim, 44
- Ex-ExPartial, 32
- ex-expartial-aconst, 32

- Ex-ExPartial-nat, 32
- ex-form-to-kernel, 26
- ex-form-to-var, 26
- ex-form-to-vars-and..., 27
- ex-form?, 25
- ex-for...-to-ex-elim-const, 17, 34
- ex-formula-to-ex-intro-aconst, 34
- Ex-Intro, 30, 32
- Ex-Intro, 34
- ex-intro, 44
- exc-elim, 46
- exc-intro, 45
- exca, 24
- exca-form-to-kernel, 26
- exca-form-to-var, 26
- exca-form?, 25
- excl, 24
- excl-form-to-kernel, 26
- excl-form-to-var, 26
- excl-form?, 25
- exnc-elim, 44
- exnc-form-to-kernel, 26
- exnc-form-to-var, 26
- exnc-form?, 25
- exnc-intro, 44
- expand-theorems, 40
- ExPartial-Ex, 32
- expartial-ex-aconst, 32
- ExPartial-Ex-nat, 32
- ext-aconst, 32
- Extensionality, 31
- extracted program, 49
- extracted term, 49
- falsity, 25
- falsity-log, 25
- Filliatre, 4
- finalg-to==-const, 17
- finalg-to==-to-e-1-aconst, 32
- finalg-to==-to-e-2-aconst, 32
- finalg-to==-to-eq-aconst, 32
- finalg-to-all-allpartial-aconst, 32
- finalg-to-e-const, 17
- finalg-to-e-to-total-aconst, 32
- finalg-to-eq-to==-1-aconst, 32
- finalg-to-eq-to==-2-aconst, 32
- finalg-to-expartial-ex-aconst, 32
- finalg-to-total-to-e-aconst, 32
- finalg?, 8
- fold-cterm, 28
- fold-formula, 27
- formula
 - definite, 48
 - folded, 24
 - goal, 48
 - prime, 24
 - relevant, 48
 - unfolded, 24
- formula-subst, 28
- formula-substitute, 28
- formula-to-et-type, 49
- formula-to-free, 27
- formula-to-prime-subformulas, 27
- formula-to-string, 28
- formula=?, 27, 28
- get, 43
- global assumption, 36
- global-assdots-name-to-aconst, 36
- goal, 41
- goal-subst, 41
- goal-to-context, 41
- goal-to-formula, 41
- goal-to-goalvar, 41
- goal=?, 41
- ground-type?, 8
- Harrop degree, 18
- Harrop formula, 18
- Huet, 4
- if-construct, 20, 33
- imp-form-to-conclusion, 25
- imp-form-to-final-conclusion, 27
- imp-form-to-premise, 25
- imp-form-to-premises, 27
- imp-form?, 25
- imp-formulas-to-elim-aconst, 34
- Ind, 33, 34
- Ind, 33
- ind, 44
- induction, 33
- inst-with, 43
- inst-with-to, 43
- Intro, 34
- intro, 44
- intro-search, 47
- Letouzey, 4
- lexical analysis, 50
- make==, 25
- make-aconst, 30
- make-alg, 8
- make-all, 26
- make-allnc, 26
- make-and, 25
- make-arity, 18
- make-arrow, 9
- make-atomic-formula, 25
- make-avar, 29
- make-const, 16
- make-cterm, 27
- make-e, 25
- make-eq, 25
- make-ex, 26

- make-exca, 26
- make-excl, 26
- make-exnc, 26
- make-imp, 25
- make-inhabited, 8
- make-predconst, 19
- make-predicate-formula, 25
- make-proof-in-aconst-form, 36
- make-proof-in-all-elim-form, 37
- make-proof-in-all-intro-form, 37
- make-proof-in-and-elim-l..., 37
- make-proof-in-and-elim-r..., 37
- make-proof-in-and-intro-form, 37
- make-proof-in-avar-form, 36
- make-proof-in-cases-form, 37
- make-proof-in-ex-intro-form, 38
- make-proof-in-imp-elim-form, 37
- make-proof-in-imp-intro-form, 36
- make-pvar, 18
- make-quant-formula, 27
- make-star, 9
- make-subst, 5
- make-subst-wrt, 5
- make-substitution, 5
- make-substitution-wrt, 5
- make-tensor, 26
- make-term-in-abst-form, 21
- make-term-in-app-form, 21
- make-term-in-const-form, 20
- make-term-in-if-form, 21
- make-term-in-lcomp-form, 21
- make-term-in-pair-form, 21
- make-term-in-rcomp-form, 21
- make-term-in-var-form, 20
- make-total, 25
- Miller, 4, 46
- min-pr, 45
- Minpr-with-measure-l11, 35
- mk-all, 26
- mk-allnc, 27
- mk-and, 26
- mk-arrow, 9
- mk-ex, 27
- mk-exca, 27
- mk-excl, 27
- mk-exnc, 27
- mk-imp, 26
- mk-neg, 26
- mk-neg-log, 26
- mk-proof-in-and-intro-form, 38
- mk-proof-in-elim-form, 38
- mk-proof-in-ex-intro-form, 38
- mk-proof-in-intro-form, 37
- mk-quant, 27
- mk-tensor, 26
- mk-term-in-abst-form, 22
- mk-term-in-app-form, 21
- mk-var, 10
- name-hyp, 43
- nbe-constr-value-to-constr, 23
- nbe-constr-value-to-name, 23
- nbe-constr-value?, 23
- nbe-constructor-pattern?, 23
- nbe-extract, 23
- nbe-fam-value?, 23
- nbe-formula-to-type, 27
- nbe-genargs, 23
- nbe-inst?, 23
- nbe-make-constr-value, 23
- nbe-make-object, 22
- nbe-match, 23
- nbe-normalize-proof, 39
- nbe-normalize-term, 23
- nbe-object-app, 22
- nbe-object-apply, 22
- nbe-object-compose, 22
- nbe-object-to-type, 22
- nbe-object-to-value, 22
- nbe-object?, 22
- nbe-pconst-...-to-object, 23
- nbe-reflect, 23
- nbe-reify, 23
- nbe-term-to-object, 23
- new-tvar, 7
- nf, 28
- ng, 41
- Nipkow, 4
- normalize-formula, 28
- normalize-goal, 41
- np, 39
- nf, 28
- nt, 23
- number-and-idpredconst-to-intro-aconst, 34
- numerated-var-to-index, 10
- numerated-var, 10
- object-type?, 9
- osubst, 6
- p-substitution-to-string, 29
- pair-elim, 46
- parsing, 51
- pattern-unify, 42
- Paulin-Mohring, 4
- Paulson, 4
- pconst-name-to-comprules, 17
- pconst-name-to-inst-objs, 17
- pconst-name-to-object, 17
- pconst-name-to-pconst, 16
- pconst-name-to-rewrules, 17
- pf, 50
- pproof-state-to-formula, 41
- pproof-state-to-proof, 41

predconst-name-to-arity, 19
 predconst-name?, 19
 predconst-to-index, 19
 predconst-to-name, 19
 predconst-to-string, 19
 predconst-to-tsubst, 19
 predconst-to-uninst-arity, 19
 predconst?, 19
 predicate constant, 18
 predicate-form-to-args, 25
 predicate-form-to-predicate, 25
 predicate-form?, 25
 prename, 28
 Presburger, 4
 prime-form?, 25
 proof-in-aconst-form-to-aconst, 36
 proof-in-aconst-form?, 36
 proof-in-all-elim-form-to-arg, 37
 proof-in-all-elim-form-to-op, 37
 proof-in-all-elim-form?, 37
 pr...all-intro-form-to-kernel, 37
 pr...all-intro-form-to-var, 37
 proof-in-all-intro-form?, 37
 proof-in-and-elim..., 37
 proof-in-and-elim-left-form?, 37
 proof-in-and-elim..., 37
 proof-in-and-elim-right-form?, 37
 pr...and-intro-form-to-left, 37
 pr...and-intro-form-to-right, 37
 proof-in-and-intro-form?, 37
 proof-in-avar-form-to-avar, 36
 proof-in-avar-form?, 36
 proof-in-cases-form-to-alts, 37
 proof-in-cases-form-to-rest, 37
 proof-in-cases-form-to-test, 37
 proof-in-cases-form?, 37
 proof-in-elim-form-to-args, 38
 pr...elim-form-to-final-op, 38
 proof-in-imp-elim-form-to-arg, 37
 proof-in-imp-elim-form-to-op, 37
 proof-in-imp-elim-form?, 37
 proof-in-imp-intro-form-to-avar, 36
 pr...imp-intro-form-to-kernel, 36
 proof-in-imp-intro-form?, 36
 proof-in-intro-form-to..., 38
 proof-of-efq-at, 40
 proof-of-efq-log-at, 40
 proof-of-stab-at, 40
 proof-of-stab-log-at, 40
 proof-subst, 40
 proof-substitute, 40
 proof-to-aconst, 38
 proof-to-aconst-without-rules, 38
 proof-to-bound-avars, 38
 proof-to-context, 38
 proof-to-expr, 40
 proof-to-extracted-term, 49
 proof-to-formula, 38
 proof-to-free, 38
 proof-to-free-and-bound-avars, 38
 proof-to-free-avars, 38
 proof-to-soundness-proof, 50
 proof=?, 38
 proof?, 38
 proofs=?, 38
 psubst, 6
 pt, 50
 pv, 50
 pvar-cterm-equal?, 6
 pvar-name-to-arity, 18
 pvar-name?, 18
 pvar-to-arity, 18
 pvar-to-h-deg, 18
 pvar-to-index, 18
 pvar-to-name, 18
 pvar?, 18
 py, 50
 Q-clause, 46
 Q-goal, 46
 Q-sequent, 46
 Q-substitution, 46
 Q-term, 46
 quant-form-to-kernel, 27
 quant-form-to-quant, 27
 quant-form-to-vars, 27
 quant-form?, 27
 quant-free?, 25
 quant-prime-form?, 25
 Rec, 14
 Rec, 52
 recursion, 13
 recursion operator, 13
 reduce-efq-and-stab, 40
 remove-alg-name, 9
 remove-computation-rules-for, 17
 remove-global-assumption, 36
 remove-predconst-name, 19
 remove-program-constant, 17
 remove-pvar-name, 18
 remove-rewrite-rules-for, 17
 remove-theorem, 34
 remove-tvar-name, 7
 remove-var-name, 10
 rename, 28
 restrict-substitution-to-args, 5
 restrict-substitution-wrt, 5
 rewrite rule, 12
 rm-exc, 41
 save, 34
 search, 47
 select, 47
 semantical model, 12

set-goal, 41
 simind, 44
 simp, 45
 simp-with, 45
 special form, 20
 split, 43
 Stab, 36
 Stab-Atom, 34
 Stab-Log, 36
 star-form-to-left-type, 9
 star-form-to-right-type, 9
 star-form?, 9
 strip, 43
 strong elimination, 4
 subst-item-equal-wrt?, 6
 substitution-equal-wrt?, 6
 substitution-equal?, 6
 substitution-to-string, 24
 synt-total?, 22

 tensor-form-to-left, 26
 tensor-form-to-parts, 27
 tensor-form-to-right, 26
 tensor-form?, 25
 term-in-abst-form-to-kernel, 21
 term-in-abst-form-to-var, 21
 term-in-abst-form?, 21
 term-in-app-form-to-arg, 21
 term-in-app-form-to-args, 21
 term-in-app-form-to-final-op, 21
 term-in-app-form-to-op, 21
 term-in-app-form?, 21
 term-in-const-form-to-const, 21
 term-in-const-form?, 21
 term-in-if-form-to-alts, 21
 term-in-if-form-to-rest, 21
 term-in-if-form-to-test, 21
 term-in-if-form?, 21
 term-in-lcomp-form-to-kernel, 21
 term-in-lcomp-form?, 21
 term-in-pair-form-to-left, 21
 term-in-pair-form-to-right, 21
 term-in-pair-form?, 21
 term-in-rcomp-form-to-kernel, 21
 term-in-rcomp-form?, 21
 term-in-var-form-to-var, 20
 term-in-var-form?, 20
 term-subst, 24
 term-substitute, 24
 term-to-bound, 22
 term-to-free, 22
 term-to-string, 22
 term-to-t-deg, 22
 term-to-type, 22
 term=?, 22
 term?, 22
 terms=?, 22

 theorem-name-to-aconst, 34, 36
 theorem-name-to-inst-proof, 34
 theorem-name-to-proof, 34
 token, 50
 token type, 16, 50
 Total, 31, 32
 total-aconst, 32
 Total-to-E, 32
 Total-to-E-nat, 31
 truth, 25
 truth-aconst, 32
 Truth-Axiom, 31, 32
 tsubst, 6
 tvar-to-index, 7
 tvar-to-name, 7
 tvar?, 7
 type constant, 5
 type substitutions, 6
 type variable, 5
 type-subst, 6
 type-substitute, 6
 type-to-new-partial-var, 11
 type-to-new-var, 11
 type-to-string, 9
 type?, 9

 undo, 43
 unfold-cterm, 28
 unfold-formula, 27
 use, 42
 use-with, 42

 var-form?, 10
 var-term-equal?, 6
 var-to-index, 10
 var-to-name, 10
 var-to-new-var, 11
 var-to-t-deg, 10
 var-to-type, 10
 var?, 10
 vector notation, 52