MINLOG REFERENCE MANUAL

HELMUT SCHWICHTENBERG

Contents

Date: March 18, 2011.

Acknowledgement. The Minlog system has been under development since around 1990; its first appearance in print is in [21]. My sincere thanks go to the many contributors:

- Freiric Barral (reflection),
- Holger Benl (Dijkstra algorithm, inductive data types),
- Ulrich Berger (very many contributions),
- Michael Bopp (program development by proof transformation),
- Wilfried Buchholz (translation of classical proofs into intuitionistic ones),
- Luca Chiarabini (program development by proof transformation),
- Laura Crosilla (tutorial),
- Matthias Eberl (normalization by evaluation),
- Simon Huber (many contributions, in particular guarded recursion, general induction),
- 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),
- Florian Ranzi (matching),
- Diana Ratiu (decoration),
- Martin Ruckert (many contributions, in particular grammar and the MPC tool),
- Stefan Schimanski (pretty printing),
- Robert Stärk (alpha equivalence),
- Monika Seisenberger (many contributions, including inductive definitions and translation of classical proofs into intuitionistic ones),
- Trifon Trifonov (functional interpretation),
- Klaus Weich (proof search, the Fibonacci numbers example),
- Wolfgang Zuber (documentation).

1. Introduction

Proofs in mathematics generally deal with abstract, "higher type" objects. Therefore an analysis of computational aspects of such proofs must be based on a theory of computation in higher types. A mathematically satisfactory such theory has been provided by Scott [24] and Ershov [9]. The basic concept is that of a *partial continuous functional*. Since each such can be seen as a limit of its finite approximations, we get for free the notion of a computable functional: it is given by a recursive enumeration of finite approximations. The price to pay for this simplicity is that functionals are now *partial*, in stark contrast to the view of Gödel [11]. However, the total functionals can be defined as a subset of partial ones. In fact, as observed by Kreisel, they form a dense subset w.r.t. the Scott topology. The next step is to build a theory, with the partial continuous functionals as the intended range of its (typed) variables. The constants of this "theory of computable functionals" TCF denote computable functionals. It suffices to restrict the prime formulas to those built with inductively defined predicates. For instance, falsity can be defined by $\mathbf{F} := \mathrm{Eq}(\mathtt{ff}, \mathtt{tt})$, where Eq is the inductively defined Leibniz equality. The only logical connectives are implication and universal quantification: existence, conjunction and disjunction can be seen as inductively defined (with parameters). TCF is well suited to reflect on the computational content of proofs, along the lines of the Brouwer-Heyting-Kolmogorov interpretation, or more technically a realizability interpretation in the sense of Kleene and Kreisel. Moreover the computational content of classical (or "weak") existence proofs can be analyzed in TCF, in the sense of Gödel's [11] Dialectica interpretation and the so-called A-translation of Friedman [10] and Dragalin [8]. The difference of TCF to well-established theories like

Martin-Löf's [16] intuitionistic type theory or the theory of constructions underlying the Coq proof assistant is that TCF treats partial continuous functionals as first class citizens. Since they are the mathematically correct domain of computable functionals, it seems that this is a reasonable step to take.

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

(i) 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.
(ii) 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.
(iii) 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]).
(iv) 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.
(v) Variables carry (simple) types, with free algebras as base types. The latter need not be finitary (we allow, e.g., countably branching trees), and can be simultaneously generated. Type and predicate parameters are allowed; they are thought of as being implicitly universally quantified ("ML polymorphism").
(vi) 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 explicitely 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

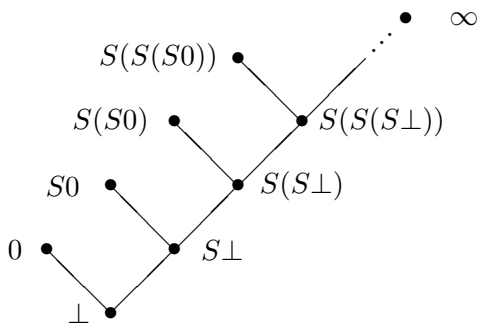

FIGURE 1. The domain of natural numbers

the constructors. Hence the constructors are total and non-strict. For the notion of totality cf. [25, 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. [25]. 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 inductively defined predicates T_ρ with the total functionals of type ρ as their intended meaning. To make formal arguments with quantifiers relativized to total objects more managable, we use a special sort of variables intended to range over such objects only. For example, $\mathtt{n0}, \mathtt{n1}, \mathtt{n2}, \ldots, \mathtt{m0}, \ldots$ range over total natural numbers, and $\mathtt{n\hat{}0}, \mathtt{n\hat{}1}, \mathtt{n\hat{}2}, \ldots$ are general variables. This amounts to an abbreviation of

$$\begin{array}{lll} \forall_{\hat{x}}(T_\rho(\hat{x}) \to A) & \text{by} & \forall_x A, \\ \exists_{\hat{x}}(T_\rho(\hat{x}) \wedge A) & \text{by} & \exists_x A. \end{array}$$

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 Y, which is axiomatized by $YF = F(YF)$. 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 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 T_ρ for infinitary algebras as well as composed types. The axioms are listed in 8.2.

1.5. **Minimal logic, proof transformation.** For generalities about minimal logic cf. [26]. 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`.

1.6. **Comparison with Coq and Isabelle.** Coq [7] 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.

(i) 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 $f(\mathtt{tt}) := A$ and $f(\mathtt{ff}) := B$ 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.

(ii) 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 $\to$ and $\forall$) can be achieved by means of inductively defined predicates. This feature is available in all three systems.

(iii) 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 [2]. At present this feature is available in the Minlog system only.

(iv) 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 explicitely via assumptions.

(v) Minlog's automated proof search `search` tool is based on [18]; 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 explicitely. This is avoided in Minlog by relativizing every proof to a set of rewrite rules, and identifyling terms and formulas with the same normal form w.r.t. these rules.

(vi) 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

Generally we consider typed theories only. Types are built from type variables and type constants by algebra type formation `(alg` $\rho_1 \dots \rho_n$`)` and arrow type formation $\rho \to \sigma$. Product types $\rho \times \sigma$ and sum types $\rho + \sigma$ can be seen as algebras with parameters. However, Minlog also has a native product type formation denoted by `star`.

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` $\to$ `atomic`, and $\forall_{n,m}\exists_k(n + m = k)$ receives the type `nat` $\to$ `nat` $\to$ `existential`. The type `prop` is used for predicate variables, e.g., R of arity `nat,nat -> 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`.

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 $\vartheta\eta$ of two substitutions

$$\vartheta = ((x_1\ s_1)\dots(x_m\ s_m)),$$
$$\eta = ((y_1\ t_1)\dots(y_n\ t_n))$$

is defined as follows. In the list $((x_1\ s_1\eta)\dots(x_m\ s_m\eta)\ (y_1\ t_1)\dots(y_n\ t_n))$ remove all bindings $(x_i\ s_i\eta)$ with $s_i\eta = x_i$, 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)
```

As display function for type substitutions one can use the general `pp-subst` or the special

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

We add here some notions and observations on substitutions ϑ for type, object, predicate and assumption variables (or topa-substitutions). Our treatment is based on (unpublished) work of Buchholz, who introduced the concept we call "admissibility" for substitutions.

Let

$$\overline{r^\rho} := \rho, \quad \overline{P^{(\vec\sigma)}} := \overline{\{\,\vec x^{\vec\sigma} \mid A\,\}} := (\vec\sigma), \quad \overline{M^A} := A.$$

Consider a substitution ϑ whose domain consists of type variables α, object variables x and predicate variables P. Let

$$\alpha\vartheta := \begin{cases} \vartheta(\alpha) & \text{if } \alpha \in \mathrm{dom}(\vartheta), \\ \alpha & \text{otherwise,} \end{cases} \qquad x\vartheta := \begin{cases} \vartheta(x) & \text{if } x \in \mathrm{dom}(\vartheta), \\ x & \text{otherwise,} \end{cases}$$

$$P\vartheta := \begin{cases} \vartheta(P) & \text{if } P \in \mathrm{dom}(\vartheta), \\ \{\,\vec x \mid P\vec x\,\} & \text{otherwise.} \end{cases}$$

Call ϑ *admissible* for x if $\overline{x\vartheta} = \overline{x}\vartheta$, and for P if $\overline{P\vartheta} = \overline{P}\vartheta$. We define the result $r\vartheta$ of carrying out a substitution ϑ in a term r, provided ϑ is admissible for all $x \in \mathrm{FV}(r)$ (in short: ϑ is admissible for r). The definition is by induction on r. $x\vartheta$ has been defined above, and

$$\begin{aligned} c\vartheta &:= c, \\ (\lambda_x r)\vartheta &:= \lambda_y(r\vartheta^y_x) \quad \text{with } y \text{ new, } \overline{y} = \overline{x}\vartheta, \\ (rs)\vartheta &:= (r\vartheta)(s\vartheta). \end{aligned}$$

To see that this definition makes sense we have to prove

Lemma. *If ϑ is admissible for $\lambda_x r$, then ϑ^y_x is admissible for r.*

Proof. Let $z \in \mathrm{FV}(r)$. We show $\overline{z\vartheta^y_x} = \overline{z}\vartheta^y_x$. *Case* $z \neq x$.

$$\overline{z\vartheta^y_x} = \overline{z\vartheta} = \overline{z}\vartheta = \overline{z}\vartheta^y_x \quad \text{since } \vartheta \text{ is admissible for } r.$$

Case $z = x$.

$$\overline{x\vartheta^y_x} = \overline{y} = \overline{x}\vartheta = \overline{x}\vartheta^y_x \quad \text{by assumption on } y. \qquad \square$$

Lemma. *Let ϑ be admissible for the term r. Then $\overline{r\vartheta} = \overline{r}\vartheta$.*

Proof. Case x. $\overline{x\vartheta} = \overline{\vartheta(x)}$ holds since ϑ is assumed to be admissible for x.

Case $\lambda_x r$.

$$\overline{(\lambda_x r)\vartheta} = \overline{\lambda_y(r\vartheta^y_x)} = \overline{y} \to \overline{r\vartheta^y_x} = \overline{x}\vartheta \to \overline{r}\vartheta^y_x = \overline{x}\vartheta \to \overline{r}\vartheta = \overline{(\lambda_x r)}\vartheta. \qquad \square$$

Lemma. *Assume that* ϑ *is admissible for* r *and* η *is admissible for* $r\vartheta$. *Then*
(a) $\eta \circ \vartheta$ *is admissible for* r, *and*
(b) $r\vartheta\eta = r(\eta \circ \vartheta)$.

Proof. (a). Let $x \in \mathrm{FV}(r)$. We show $\overline{x(\eta \circ \vartheta)} = \overline{x}(\eta \circ \vartheta)$, i.e., $\overline{x\vartheta\eta} = \overline{x\vartheta}\eta$. Consider $x\vartheta$. Since η is admissible for $r\vartheta$, it is also admissible for the subterm $x\vartheta$. Hence by the previous lemma $\overline{x\vartheta\eta} = \overline{x\vartheta}\eta$.

(b). We only consider the abstraction case. By definition

$(\lambda_x r)\vartheta = \lambda_y(r\vartheta^y_x)$ with y new, $\overline{y} = \overline{x}\vartheta$.

$(\lambda_x r)\vartheta\eta = \lambda_y(r\vartheta^y_x)\eta = \lambda_z(r\vartheta^y_x\eta^z_y)$ with z new, $\overline{z} = \overline{y}\eta$.

$(\lambda_x r)(\eta \circ \vartheta) = \lambda_u(r(\eta \circ \vartheta)^u_x)$ with u new, $\overline{u} = \overline{x}(\eta \circ \vartheta) = \overline{x}\vartheta\eta = \overline{y}\eta = \overline{z}$.

Hence we may assume $u = z$. But $\lambda_u(r(\eta \circ \vartheta)^u_x) = \lambda_z(r(\eta^z_y \circ \vartheta^y_x))$, since $y \notin \mathrm{FV}(r)$ and

$$(\eta \circ \vartheta)^u_x v = v = (\eta^z_y \circ \vartheta^y_x)v \quad \text{for } v \neq x, y,$$
$$(\eta \circ \vartheta)^u_x x = u = z = (\eta^z_y \circ \vartheta^y_x)x.$$

By induction hypothesis $\lambda_z(r(\eta^z_y \circ \vartheta^y_x)) = \lambda_z(r\vartheta^y_x\eta^z_y)$. Hence the claim. $\square$

The result $A\vartheta$ and $\{\,\vec{x} \mid A\,\}\vartheta$ of carrying out a substitution ϑ in a formula A or a comprehension term $\{\,\vec{x} \mid A\,\}$ is defined similarly, provided ϑ is admissible for the respective expression, and similar lemmata can be proven.

Now consider a type-object-predicate-assumption substitution ϑ with type variables α, object variables x, predicate variables P and assumption variables u in its domain. Again we allow that the type σ of x and the arity $(\vec{\sigma})$ of P depend on type variables $\alpha \in \mathrm{dom}(\vartheta)$, but we require $\overline{\vartheta(x)} = \overline{x}\vartheta$ and $\overline{\vartheta(P)} = \overline{P}\vartheta$. Moreover we allow that the formula A of u depends on $\alpha, x, P \in \mathrm{dom}(\vartheta)$, but we require $\overline{\vartheta(u)} = \overline{u}\vartheta$. Let

$$u\vartheta := \begin{cases} \vartheta(u) & \text{if } u \in \mathrm{dom}(\vartheta), \\ u & \text{otherwise.} \end{cases}$$

Call a type-object-predicate-assumption substitution *admissible* for a derivation M if for all $x, P, u \in \mathrm{FV}(M)$ we have $\overline{x\vartheta} = \overline{x}\vartheta$, $\overline{P\vartheta} = \overline{P}\vartheta$ and $\overline{u\vartheta} = \overline{u}\vartheta$. The result $M\vartheta$ of carrying out a substitution ϑ in a derivation M is defined as follows, provided ϑ is admissible for M. We define $M\vartheta$ by induction on M.

$$c\vartheta := c,$$

$$
\begin{aligned}
(\lambda_x M)\vartheta &:= \lambda_y(M\vartheta^y_x) \quad \text{with } y \text{ new, } \overline{y} = \overline{x}\vartheta,\\
(Mr)\vartheta &:= (M\vartheta)(r\vartheta),\\
(\lambda_u M)\vartheta &:= \lambda_v(M\vartheta^v_u) \quad \text{with } v \text{ new, } \overline{v} = \overline{u}\vartheta,\\
(MN)\vartheta &:= (M\vartheta)(N\vartheta).
\end{aligned}
$$

Again lemmata similar to those above can be proven.

As test for the admissibility of a substitution we provide

`(admissible-substitution? topasubst expr)`

2.2. **Type unification and matching.** We need type unification for object types only, that is, types built from type variables and algebra types by arrow and star. However, the type constants `atomic`, `existential`, `prop` and `nulltype` do not do any harm and can be included.

`type-unify` checks whether two terms can be unified. It returns `#f`, if this is impossible, and a most general unifier otherwise. `type-unify-list` does the same for lists of terms. We provide

`(type-unify type1 type2)`

`(type-unify-list types1 types2)`

Notice that the algorithm we use (via disagreement pairs) does not yield idempotent unifiers (as opposed to the Martelli-Montanari algorithm [14] in `modules/type-inf.scm`):

```
(pp-subst (type-unify (py "alpha1=>alpha2=>boole")
                      (py "alpha2=>alpha1=>alpha1")))
;   alpha2 -> boole
;   alpha1 -> alpha2
```

`type-match` checks whether a given pattern can be transformed by a substitution into a given instance. It returns `#f`, if this is impossible, and the substitution otherwise. `type-match-list` does the same for lists of terms. We provide

`(type-match pattern instance)`

`(type-match-list patterns instances)`

2.3. **Algebras and types.** We now consider concrete information systems, our basis for continuous functionals.

Types will be built from base types by the formation of function types, $\rho \to \sigma$. As domains for the base types we choose non-flat and possibly infinitary free algebras, given by their constructors. The main reason for taking non-flat base domains is that we want the constructors to be injective and with disjoint ranges. This generally is not the case for flat domains.

Definition (Algebras and types)**.** Let $\xi, \vec{\alpha}$ be distinct type variables; the α_l are called *type parameters*. We inductively define *type forms* $\rho, \sigma, \tau \in \mathrm{Ty}(\vec{\alpha})$, *constructor type forms* $\kappa \in \mathrm{KT}_\xi(\vec{\alpha})$ and *algebra forms* $\iota \in \mathrm{Alg}(\vec{\alpha})$; all these are called *strictly positive* in $\vec{\alpha}$. In case $\vec{\alpha}$ is empty we abbreviate $\mathrm{Ty}(\vec{\alpha})$ by Ty and call its elements *types* rather than type forms; similarly for the other notions.

$$\alpha_l \in \mathrm{Ty}(\vec{\alpha}), \qquad \frac{\iota \in \mathrm{Alg}(\vec{\alpha})}{\iota \in \mathrm{Ty}(\vec{\alpha})}, \qquad \frac{\rho \in \mathrm{Ty} \quad \sigma \in \mathrm{Ty}(\vec{\alpha})}{\rho \to \sigma \in \mathrm{Ty}(\vec{\alpha})},$$
$$\frac{\kappa_0, \dots, \kappa_{k-1} \in \mathrm{KT}_\xi(\vec{\alpha})}{\mu_\xi(\kappa_0, \dots, \kappa_{k-1}) \in \mathrm{Alg}(\vec{\alpha})} \quad (k \geq 1),$$
$$\frac{\vec{\rho} \in \mathrm{Ty}(\vec{\alpha}) \quad \vec{\sigma}_0, \dots, \vec{\sigma}_{n-1} \in \mathrm{Ty}}{\vec{\rho} \to (\vec{\sigma}_\nu \to \xi)_{\nu<n} \to \xi \in \mathrm{KT}_\xi(\vec{\alpha})} \quad (n \geq 0).$$

We use ι for algebra forms and ρ, σ, τ for type forms. $\vec{\rho} \to \sigma$ means $\rho_0 \to \dots \to \rho_{n-1} \to \sigma$, associated to the right. For $\vec{\rho} \to (\vec{\sigma}_\nu \to \xi)_{\nu<n} \to \xi \in \mathrm{KT}_\xi(\vec{\alpha})$ call $\vec{\rho}$ the *parameter* argument types and the $\vec{\sigma}_\nu \to \xi$ *recursive* argument types. To avoid empty types, we require that there is a *nullary* constructor type, i.e., one without recursive argument types.

Here are some examples of algebras.

$$\begin{aligned}
\mathbf{U} &:= \mu_\xi \xi && \text{(unit)},\\
\mathbf{B} &:= \mu_\xi(\xi, \xi) && \text{(booleans)},\\
\mathbf{N} &:= \mu_\xi(\xi, \xi \to \xi) && \text{(natural numbers, unary)},\\
\mathbf{P} &:= \mu_\xi(\xi, \xi \to \xi, \xi \to \xi) && \text{(positive numbers, binary)},\\
\mathbf{D} &:= \mu_\xi(\xi, \xi \to \xi \to \xi) && \text{(binary trees, or derivations)},\\
\mathbf{O} &:= \mu_\xi(\xi, \xi \to \xi, (\mathbf{N} \to \xi) \to \xi) && \text{(ordinals)},\\
\mathbf{T}_0 &:= \mathbf{N}, \quad \mathbf{T}_{n+1} := \mu_\xi(\xi, (\mathbf{T}_n \to \xi) \to \xi) && \text{(trees)}.
\end{aligned}$$

Important examples of algebra forms are

$$\begin{aligned}
\mathbf{L}(\alpha) &:= \mu_\xi(\xi, \alpha \to \xi \to \xi) && \text{(lists)},\\
\alpha \times \beta &:= \mu_\xi(\alpha \to \beta \to \xi) && \text{(product)},\\
\alpha + \beta &:= \mu_\xi(\alpha \to \xi, \beta \to \xi) && \text{(sum)}.
\end{aligned}$$

Remark (Substitution for type parameters). Let $\rho \in \mathrm{Ty}(\vec{\alpha})$; we write $\rho(\vec{\alpha})$ for ρ to indicate its dependence on the type parametes $\vec{\alpha}$. We can substitute types $\vec{\sigma}$ for $\vec{\alpha}$, to obtain $\rho(\vec{\sigma})$. Examples are $\mathbf{L}(\mathbf{B})$, the type of lists of booleans, and $\mathbf{N} \times \mathbf{N}$, the type of pairs of natural numbers.

Note that often there are many equivalent ways to define a particular type. For instance, we could take $\mathbf{U} + \mathbf{U}$ to be the type of booleans, $\mathbf{L}(\mathbf{U})$

to be the type of natural numbers, and $\mathbf{L}(\mathbf{B})$ to be the type of positive binary numbers.

For every constructor type $\kappa_i(\xi)$ of an algebra $\iota = \mu_\xi(\vec{\kappa})$ we provide a (typed) *constructor symbol* C_i of type $\kappa_i(\iota)$. In some cases they have standard names, for instance

$\mathtt{tt}^{\mathbf{B}}, \mathtt{ff}^{\mathbf{B}}$ for the two constructors of the type $\mathbf{B}$ of booleans,

$0^{\mathbf{N}}, \mathrm{S}^{\mathbf{N}\to\mathbf{N}}$ for the type $\mathbf{N}$ of (unary) natural numbers,

$1^{\mathbf{P}}, S_0^{\mathbf{P}\to\mathbf{P}}, S_1^{\mathbf{P}\to\mathbf{P}}$ for the type $\mathbf{P}$ of (binary) positive numbers,

$\mathrm{nil}^{\mathbf{L}(\rho)}, \mathrm{cons}^{\rho\to\mathbf{L}(\rho)\to\mathbf{L}(\rho)}$ for the type $\mathbf{L}(\rho)$ of lists,

$(\mathrm{inl}_{\rho\sigma})^{\rho\to\rho+\sigma}, (\mathrm{inr}_{\rho\sigma})^{\sigma\to\rho+\sigma}$ for the sum type $\rho+\sigma$.

We denote the constructors of the type $\mathbf{D}$ of derivations by $0^{\mathbf{D}}$ (axiom) and $\mathrm{C}^{\mathbf{D}\to\mathbf{D}\to\mathbf{D}}$ (rule).

One can extend the definition of algebras and types to *simultaneously defined algebras*: just replace ξ by a list $\vec{\xi} = \xi_0, \ldots, \xi_{N-1}$ of type variables and change the algebra introduction rule to

$$\frac{\kappa_0, \ldots, \kappa_{k-1} \in \mathrm{KT}_{\vec{\xi}}(\vec{\alpha})}{(\mu_{\vec{\xi}}(\kappa_0, \ldots, \kappa_{k-1}))_j \in \mathrm{Alg}(\vec{\alpha})} \quad (k \geq 1,\, j < N).$$

with each κ_i of the form

$$\vec{\rho} \to (\vec{\sigma}_\nu \to \xi_{j_\nu})_{\nu<n} \to \xi_j.$$

The definition of a "nullary" constructor type is a little more delicate here. We require that for every ξ_j $(j < N)$ there is a κ_{i_j} with final value type ξ_j, each of whose recursive argument types has a final value type ξ_{j_ν} with $j_\nu < j$. — Examples of simultaneously defined algebras are

$(\mathbf{Ev}, \mathbf{Od}) := \mu_{\xi,\zeta}(\xi, \zeta \to \xi, \xi \to \zeta)$ (even and odd numbers),

$(\mathbf{Ts}(\rho), \mathbf{T}(\rho)) := \mu_{\xi,\zeta}(\xi, \zeta \to \xi \to \xi, \rho \to \zeta, \xi \to \zeta)$ (tree lists and trees).

$\mathbf{T}(\rho)$ defines finitely branching trees, and $\mathbf{Ts}(\rho)$ finite lists of such trees; the trees carry objects of a type ρ at their leaves. The constructor symbols and their types are

$$\mathrm{Empty}^{\mathbf{Ts}(\rho)}, \quad \mathrm{Tcons}^{\mathbf{T}(\rho)\to\mathbf{Ts}(\rho)\to\mathbf{Ts}(\rho)},$$
$$\mathrm{Leaf}^{\rho\to\mathbf{T}(\rho)}, \quad \mathrm{Branch}^{\mathbf{Ts}(\rho)\to\mathbf{T}(\rho)}.$$

However, for simplicity we often consider non-simultaneous algebras only.

An algebra is *finitary* if all its constructor types (i) only have finitary algebras as parameter argument types, and (ii) have recursive argument types of the form ξ only (so the $\vec{\sigma}_\nu$ in the general definition are all empty).

Structure-finitary algebras are defined similarly, but without conditions on parameter argument types. In the examples above $\mathbf{U}$, $\mathbf{B}$, $\mathbf{N}$, $\mathbf{P}$ and $\mathbf{D}$ are all finitary, but $\mathbf{O}$ and $\mathbf{T}_{n+1}$ are not. $\mathbf{L}(\rho)$, $\rho \times \sigma$ and $\rho + \sigma$ are structure-finitary, and finitary if their parameter types are. An argument position in a type is called *finitary* if it is occupied by a finitary algebra.

An algebra is *explicit* if all its constructor types have parameter argument types only (i.e., no recursive argument types). In the examples above $\mathbf{U}$, $\mathbf{B}$, $\rho \times \sigma$ and $\rho + \sigma$ are explicit, but $\mathbf{N}$, $\mathbf{P}$, $\mathbf{L}(\rho)$, $\mathbf{D}$, $\mathbf{O}$ and $\mathbf{T}_{n+1}$ are not.

We will also need the notion of the *level* of a type, which is defined by

$$\mathrm{lev}(\iota) := 0, \qquad \mathrm{lev}(\rho \to \sigma) := \max\{\mathrm{lev}(\sigma), 1 + \mathrm{lev}(\rho)\}.$$

Base types are types of level 0, and a *higher* type has level at least 1.

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
`(sfinalg? 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.

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 algebra of binary trees. The domain $\mathcal{I}_{\mathtt{nat}}$ 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 `boole` 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 `boole`.

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)
(pp type).
```

2.4. **Coercion.** To develop analysis we use a subtype relation generated from `pos` $<$ `nat` $<$ `int` $<$ `rat` $<$ `real` $<$ `cpx`. We view `pos`, `nat`, `int`, `rat`, `real`, `cpx` as algebras with the following constructors and destructors.

`pos`: `One`, `SZero`, `SOne` (positive numbers written in binary)

`nat`: `Zero`, `Succ`

`int`: `IntPos`, `IntZero`, `IntNeg`

`rat`: `RatConstr` (written # infix) and destructors `RatN`, `RatD`

`real`: `RealConstr` and destructors `RealSeq`, `RealMod`

`cpx`: `CpxConstr` (written ## infix) and destructors `RealPart`, `ImagPart`

We provide

```
(alg-le?  alg1 alg2)
```

```
(type-le? type1 type2)
(algebras-to-embedding type1 type2)
(types-to-embedding type1 type2)
(types-lub type . types)
```

`type-match-modulo-coercion` checks whether a given pattern can be transformed modulo coercion by a substitution into a given instance. It returns `#f`, if this is impossible, and the substitution otherwise. We provide

```
(type-match-modulo-coercion pattern instance)
```

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 [25, Chapter 8.3]; particularly relevant here is exercise 8.5.7. To make formal arguments with quantifiers relativized to total objects more managable, we use a special sort of variables intended to range over such objects only. For example, `n0, n1, n2, ..., m0, ...` range over total natural numbers, and `n^0, n^1, n^2, ...` are general variables. We say that the *degree of totality* for the former is 1, and for the latter 0.

To add and remove names for variables of a given type (e.g., `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.

```
(make-var type index t-deg name)  constructor
```

`(var-to-type` *var*`)`	accessor
`(var-to-index` *var*`)`	accessor
`(var-to-t-deg` *var*`)`	accessor
`(var-to-name` *var*`)`	accessor
`(var-form?` *x*`)`	incomplete test
`(var?` *x*`).`	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 of the product of Ty, $\mathbb{N}$, and the degrees of totality to the set of numerated variables, 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, for instance for bound renaming. Therefore we supply

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

(i) constructors, kind `constr`,
(ii) constants with user defined rules (also called program(mable) constant, or pconst), kind `pconst`,
(iii) 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 $\alpha \to \alpha \to \mathbf{B}$ 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.

4.1. **Rewrite and computation rules for program constants.** For every program constant (or defined constant) D^ρ we assume that some rewrite rules of the form $D\vec{K} \mapsto N$ are given, where $\mathrm{FV}(N) \subseteq \mathrm{FV}(\vec{K})$ and $D\vec{K}$, N have the same type (not necessarily a ground type). Moreover, for any two rules $D\vec{K} \mapsto N$ and $D\vec{K}' \mapsto N'$ we require that $\vec{K}$ and $\vec{K}'$ are of the same length, called the *arity* of D. 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" $\vec{\rho} \to \sigma$ 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 D is of type $\tau \to (\tau \to \tau \to \tau) \times (\tau \to \tau)$, then the rules $Dy0xx \mapsto a$ and $Dy1 \mapsto b$ are admitted, and D comes with the type informations $(\tau, 0, \tau, \tau \to \tau) \to \tau$ and $(\tau, 1) \to (\tau \to \tau)$. – 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.

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

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

(i) If $D\vec{P} \mapsto Q \in \mathrm{Comp}$, then $P_1, \dots, P_n$ are constructor patterns or projection markers.

(ii) The rules are left-linear, i.e., if $D\vec{P} \mapsto Q \in \mathrm{Comp}$, then every variable in $D\vec{P}$ occurs only once in $D\vec{P}$.

(iii) The rules are essentially non-overlapping, i.e., for different rules $D\vec{K} \mapsto M$ and $D\vec{L} \mapsto N$ in Comp the left hand sides $D\vec{K}$ and $D\vec{L}$ are either non-unifiable, or else for the most general unifier ξ of $\vec{K}$ and $\vec{L}$ we have $M\xi = N\xi$.

We write $D\vec{M} \mapsto_{\mathrm{comp}} Q$ to indicate that the rule is in Comp. All other rules will be called (proper) rewrite rules.

In our reduction strategy computation rules will always be applied first, and since they are essentially 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 "select"-functions were used for this purpose).

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 $\vec{\iota} = \mu_{\vec{\xi}}\vec{\kappa}$ corresponds to two sorts of constants. With the *constructors* $\mathrm{C}_i^{\vec{\iota}}$: $\kappa_i[\vec{\iota}]$ we can construct elements of a type ι_j, and with the *recursion operators* $\mathcal{R}_{\iota_j}^{\vec{\iota},\vec{\tau}}$ we can construct mappings from ι_j to τ_j by recursion on the structure of $\vec{\iota}$. So in `(Rec arrow-types)`, `arrow-types` is a list $\iota_1 \to \tau_1, \ldots, \iota_k \to \tau_k$. Here $\iota_1, \ldots, \iota_k$ are the algebras defined simultaneously and $\tau_1, \ldots, \tau_k$ 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 $\forall_{x_1^{\iota_1}} A_1, \ldots, \forall_{x_k^{\iota_k}} A_k$ instead of `arrow-types`: they are treated as $\iota_1 \to \tau(A_1)$, $\ldots$, $\iota_k \to \tau(A_k)$ 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. The (structural) higher type *recursion operators* $\mathcal{R}_\iota^\tau$ (introduced by Gödel [11]) are used to construct maps from the algebra ι to τ, by recursion on the structure of ι. For instance, $\mathcal{R}_{\mathbf{N}}^\tau$ has type $\mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau$. The first argument is the recursion argument, the second one gives the base value, and the third one gives the step function, mapping the recursion argument and the previous value to the next value. For example, $\mathcal{R}_{\mathbf{N}}^{\mathbf{N}} nm\lambda_{n,p}(\mathrm{S}p)$ defines addition $m + n$ by recursion an n.

Generally, in order to define the type of the recursion operators w.r.t. $\iota = \mu_\xi\,(\kappa_0, \ldots, \kappa_{k-1})$ and result type τ, we first define for each constructor type

$$\kappa = \vec{\rho} \to (\vec{\sigma}_\nu \to \xi)_{\nu<n} \to \xi \in \mathrm{KT}_\xi$$

the *step type*

$$\delta := \vec{\rho} \to (\vec{\sigma}_\nu \to \iota)_{\nu<n} \to (\vec{\sigma}_\nu \to \tau)_{\nu<n} \to \tau.$$

The recursion operator $\mathcal{R}^\tau_\iota$ then has type

$$\iota \to \delta_0 \to \ldots \to \delta_{k-1} \to \tau$$

where k is the number of constructors. The recursion argument is of type ι. In the step type δ above, the $\vec{\rho}$ are parameter types, $(\vec{\sigma}_\nu \to \iota)_{\nu<n}$ are the types of the predecessor components in the recursion argument, and $(\vec{\sigma}_\nu \to \tau)_{\nu<n}$ are the types of the previously defined values.

For some common algebras listed in 2.3 we spell out the type of their recursion operators:

$\mathcal{R}^\tau_{\mathbf{B}} \colon \mathbf{B} \to \tau \to \tau \to \tau$,
$\mathcal{R}^\tau_{\mathbf{N}} \colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau$,
$\mathcal{R}^\tau_{\mathbf{P}} \colon \mathbf{P} \to \tau \to (\mathbf{P} \to \tau \to \tau) \to (\mathbf{P} \to \tau \to \tau) \to \tau$,
$\mathcal{R}^\tau_{\mathbf{O}} \colon \mathbf{O} \to \tau \to (\mathbf{O} \to \tau \to \tau) \to ((\mathbf{N} \to \mathbf{O}) \to (\mathbf{N} \to \tau) \to \tau) \to \tau$,
$\mathcal{R}^\tau_{\mathbf{L}(\rho)} \colon \mathbf{L}(\rho) \to \tau \to (\rho \to \mathbf{L}(\rho) \to \tau \to \tau) \to \tau$,
$\mathcal{R}^\tau_{\rho+\sigma} \colon \rho + \sigma \to (\rho \to \tau) \to (\sigma \to \tau) \to \tau$,
$\mathcal{R}^\tau_{\rho\times\sigma} \colon \rho \times \sigma \to (\rho \to \sigma \to \tau) \to \tau$.

One can extend the definition of the (structural) recursion operators to simultaneously defined algebras $\vec{\iota} = \mu_{\vec{\xi}}(\kappa_0, \ldots, \kappa_{k-1})$ and result types $\vec{\tau}$. Then for each constructor type

$$\kappa = \vec{\rho} \to (\vec{\sigma}_\nu \to \xi_{j_\nu})_{\nu<n} \to \xi_j \in \mathrm{KT}_{\vec{\xi}}$$

we have the *step type*

$$\delta := \vec{\rho} \to (\vec{\sigma}_\nu \to \iota_{j_\nu})_{\nu<n} \to (\vec{\sigma}_\nu \to \tau_{j_\nu})_{\nu<n} \to \tau_j.$$

The jth simultaneous recursion operator $\mathcal{R}^{\vec{\iota},\vec{\tau}}_j$ has type

$$\iota_j \to \delta_0 \to \ldots \to \delta_{k-1} \to \tau_j$$

where k is the *total* number of constructors. The recursion argument is of type ι_j. In the step type δ, the $\vec{\rho}$ are parameter types, $(\vec{\sigma}_\nu \to \iota_{j_\nu})_{\nu<n}$ are the types of the predecessor components in the recursion argument, and $(\vec{\sigma}_\nu \to \tau_{j_\nu})_{\nu<n}$ are the types of the previously defined values. We will often omit the upper indices $\vec{\iota}, \vec{\tau}$ when they are clear from the context. Notice that in case of a non-simultaneous free algebra we write $\mathcal{R}^\tau_\iota$ for $\mathcal{R}^{\iota,\tau}_1$. — An example of a simultaneous recursion on tree lists and trees will be given below.

Definition. *Terms of Gödel's* T are inductively defined from typed variables x^ρ and constants for constructors $\mathrm{C}_i^{\vec\iota}$ and recursion operators $\mathcal{R}_j^{\vec\iota,\vec\tau}$ by abstraction $\lambda_{x^\rho} M^\sigma$ and application $M^{\rho\to\sigma} N^\rho$.

4.3. **Conversion.** To define the conversion relation for the structural recursion operators, it will be helpful to use the following notation. Let $\vec\iota = \mu_{\vec\xi}\vec\kappa$,

$$\kappa_i = \rho_0 \to \ldots \to \rho_{m-1} \to (\vec\sigma_0 \to \xi_{j_0}) \to \ldots \to (\vec\sigma_{n-1} \to \xi_{j_{n-1}}) \to \xi_j \in \mathrm{KT}_{\vec\xi},$$

and consider $\mathrm{C}_i^{\vec\iota}\vec N$ of type ι_j. We write $\vec N^P = N_0^P, \ldots, N_{m-1}^P$ for the *parameter arguments* $N_0^{\rho_0}, \ldots, N_{m-1}^{\rho_{m-1}}$ and $\vec N^R = N_0^R, \ldots, N_{n-1}^R$ for the *recursive arguments* $N_m^{\vec\sigma_0\to\iota_{j_0}}, \ldots, N_{m+n-1}^{\vec\sigma_{n-1}\to\iota_{j_{n-1}}}$, and n^R for the number n of recursive arguments.

We define a *conversion relation* $\mapsto_\rho$ between terms of type ρ by

(1) $(\lambda_x M(x))N \mapsto M(N)$,

(2) $\lambda_x(Mx) \mapsto M$ if $x \notin \mathrm{FV}(M)$ (M not an abstraction),

(3) $\mathcal{R}_j(\mathrm{C}_i^{\vec\iota}\vec N)\vec M \mapsto M_i\vec N\big((\mathcal{R}_{j_0}\cdot\vec M)\circ N_0^R\big)\ldots\big((\mathcal{R}_{j_{n-1}}\cdot\vec M)\circ N_{n-1}^R\big)$.

Here we have written $\mathcal{R}_j\cdot\vec M$ for $\lambda_{x^{\iota_j}}(\mathcal{R}_j^{\vec\iota,\vec\tau}x^{\iota_j}\vec M)$; $\circ$ denotes ordinary composition. The rule (1) is called β-conversion, and (2) η-conversion; their left hand sides are called *β-redexes* or *η-redexes*, respectively. The left hand side of (3) is called *$\mathcal{R}$-redex*; it is a special case of a redex associated with a constant D defined by "computation rules" (cf. 4.1), and hence also called a *D-redex*.

Let us look at some examples of what can be defined in Gödel's T. We define the *canonical inhabitant* ε^ρ of a type $\rho \in \mathrm{Ty}$:

$$\varepsilon^{\iota_j} := \mathrm{C}_{ij}^{\vec\iota}\varepsilon^{\vec\rho}(\lambda_{\vec x_1}\varepsilon^{\iota_{j_1}})\ldots(\lambda_{\vec x_n}\varepsilon^{\iota_{j_n}}), \quad \varepsilon^{\rho\to\sigma} := \lambda_x\varepsilon^\sigma.$$

The *projections* of a pair to its components can be defined easily:

$$M0 := \mathcal{R}^\rho_{\rho\times\sigma}M^{\rho\times\sigma}(\lambda_{x^\rho,y^\sigma}x^\rho), \quad M1 := \mathcal{R}^\sigma_{\rho\times\sigma}M^{\rho\times\sigma}(\lambda_{x^\rho,y^\sigma}y^\sigma).$$

The *append*-function $*$ for lists is defined recursively as follows. We write $x :: l$ as shorthand for $\mathrm{cons}(x, l)$.

$$\mathrm{nil} * l_2 := l_2, \qquad (x :: l_1) * l_2 := x :: (l_1 * l_2).$$

It can be defined as the term

$$l_1 * l_2 := \mathcal{R}^{\mathbf{L}(\alpha)\to\mathbf{L}(\alpha)}_{\mathbf{L}(\alpha)} l_1(\lambda_{l_2}l_2)\lambda_{x,\_,p,l_2}(x :: (pl_2))l_2.$$

Here "_" is a name for a bound variable which is not used.

Using the append function $*$ we can define *list reversal* Rev by

$$\mathrm{Rev}(\mathrm{nil}) := \mathrm{nil}, \qquad \mathrm{Rev}(x :: l) := \mathrm{Rev}(l) * (x :: \mathrm{nil}).$$

The corresponding term is

$$\mathrm{Rev}(l) := \mathcal{R}^{\mathbf{L}(\alpha)}_{\mathbf{L}(\alpha)} l\ \mathrm{nil} \lambda_{x,\_,p}(p * (x :: \mathrm{nil})).$$

Assume we want to define by simultaneous recursion two functions on $\mathbf{N}$, say $\mathrm{even}, \mathrm{odd}\colon \mathbf{N} \to \mathbf{B}$. We want

$$\begin{aligned} \mathrm{even}(0) &:= \mathtt{tt}, & \mathrm{odd}(0) &:= \mathtt{ff}, \\ \mathrm{even}(\mathrm{S}n) &:= \mathrm{odd}(n), & \mathrm{odd}(\mathrm{S}n) &:= \mathrm{even}(n). \end{aligned}$$

This can be achieved by using pair types: we recursively define the single function $\mathrm{evenodd}\colon \mathbf{N} \to \mathbf{B} \times \mathbf{B}$. The step types are

$$\delta_0 = \mathbf{B} \times \mathbf{B}, \quad \delta_1 = \mathbf{N} \to \mathbf{B} \times \mathbf{B} \to \mathbf{B} \times \mathbf{B},$$

and we can define $\mathrm{evenodd}\, m := \mathcal{R}^{\mathbf{B}\times\mathbf{B}}_{\mathbf{N}} m \langle \mathtt{tt}, \mathtt{ff}\rangle \lambda_{n,p}\langle p1, p0\rangle$.

Another example concerns the algebras $(\mathbf{Ts}(\mathbf{N}), \mathbf{T}(\mathbf{N}))$ simultaneously defined in 2.3 (we write them without the argument $\mathbf{N}$ here), whose constructors $\mathrm{C}^{(\mathbf{Ts},\mathbf{T})}_i$ for $i \in \{0,\dots,3\}$ are

$$\mathrm{Empty}^{\mathbf{Ts}}, \quad \mathrm{Tcons}^{\mathbf{T}\to\mathbf{Ts}\to\mathbf{Ts}}, \quad \mathrm{Leaf}^{\mathbf{N}\to\mathbf{T}}, \quad \mathrm{Branch}^{\mathbf{Ts}\to\mathbf{T}}.$$

Recall that the elements of the algebra $\mathbf{T}$ (i.e., $\mathbf{T}(\mathbf{N})$) are just the finitely branching trees, which carry natural numbers on their leaves.

Let us compute the types of the recursion operators w.r.t. the result types τ_0, τ_1, i.e., of $\mathcal{R}^{(\mathbf{Ts},\mathbf{T}),(\tau_0,\tau_1)}_{\mathbf{Ts}}$ and $\mathcal{R}^{(\mathbf{Ts},\mathbf{T}),(\tau_0,\tau_1)}_{\mathbf{T}}$, or shortly $\mathcal{R}_{\mathbf{Ts}}$ and $\mathcal{R}_{\mathbf{T}}$. The step types are

$$\begin{aligned} \delta_0 &:= \tau_0, & \delta_2 &:= \mathbf{N} \to \tau_1, \\ \delta_1 &:= \mathbf{Ts} \to \mathbf{T} \to \tau_0 \to \tau_1 \to \tau_0, & \delta_3 &:= \mathbf{Ts} \to \tau_0 \to \tau_1. \end{aligned}$$

Hence the types of the recursion operators are

$$\begin{aligned} \mathcal{R}_{\mathbf{Ts}}&\colon \mathbf{Ts} \to \delta_0 \to \delta_1 \to \delta_2 \to \delta_3 \to \tau_0, \\ \mathcal{R}_{\mathbf{T}}&\colon \mathbf{T} \to \delta_0 \to \delta_1 \to \delta_2 \to \delta_3 \to \tau_1. \end{aligned}$$

The internal representation of $\mathcal{R}_{\mathbf{T}}$ is

$$\begin{aligned} &\texttt{(const Rec } \delta'_0 \to \delta'_1 \to \delta'_2 \to \delta'_3 \to \mathbf{T} \to \alpha_0 \\ &\qquad (\alpha_0 \mapsto \tau_0, \alpha_1 \mapsto \tau_1)\texttt{)} \end{aligned}$$

with

$$\begin{aligned} \delta'_0 &:= \alpha_0, & \delta'_2 &:= \mathbf{N} \to \alpha_1, \\ \delta'_1 &:= \mathbf{Ts} \to \mathbf{T} \to \alpha_0 \to \alpha_1 \to \alpha_0, & \delta'_3 &:= \mathbf{Ts} \to \alpha_0 \to \alpha_1. \end{aligned}$$

Here the fact that we deal with a simultaneous recursion (over `tree` and `tlist`), and that we define a constant of type $\mathbf{T} \to \dots$, can all be inferred from what is given: the type $\mathbf{T} \to \dots$ 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

$$\texttt{(Rec } \mathbf{T} \to \tau_0 \ \ \mathbf{Ts} \to \tau_1\texttt{)}.$$

4.4. **Internal representation of constants.** Every object constant has the internal representation

(`const` *object-or-arity name uninst-type tsubst*
t-deg token-type repro-data`)`

The type of the constant is the result of carrying out the type substitution *tsubst* in *uninst-type*; free type variables may again occur in this type. The type substitution *tsubst* must be restricted to the type variables in `uninst-type`. Examples for object constants are

`(const Compose` $(\alpha{\to}\beta){\to}(\beta{\to}\gamma){\to}\alpha{\to}\gamma \ \ (\alpha \mapsto \rho, \beta \mapsto \sigma, \gamma \mapsto \tau)$ `...)`

`(const Eq` $\alpha \to \alpha \to \mathbf{B} \ \ (\alpha \mapsto$ *finalg*$)$ `...)`

`(const E` $\alpha \to \mathbf{B} \ \ (\alpha \mapsto$ *finalg*$\ldots)$`)`

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`.

Repro-data are (only) necessary in `proof.scm`, for normalization of proofs: a (general) induction, efq, introduction or elimination axiom is translated into an appropriate constant, then normalized, and finally from the constant and its repro data the axiom is reproduced. The repro-data are of the following forms.

(1) For a recursion constant.

(a) A list of all-formulas. This form only occurs when translating an axiom for (simultaneous) induction into a recursion constant, in order to achieve normalization of proofs via term normalization. We have to consider the free variables in the scheme formulas, and let the type of the recursion constant depend on them. This is needed to have the allnc-conversion be represented in term normalization. The relevant operation is

`all-formulas-to-rec-const`.

(b) A list of implication formulas $I\vec{x}^\hat{} \to A(\vec{x}^\hat{})$, where all idpcs are simultaneously inductively defined. This form only occurs when translating an elimination axiom into a recursion constant, in order to achieve normalization of proofs via term normalization. We again have to consider the free variables in the scheme formulas, and let the type of the recursion constant depend on them. This is needed to have the allnc-conversion be represented in term normalization. The relevant operation is

`imp-formulas-to-rec-const`.

(2) For a cases constant. Here a single arrow-type or all-formula suffices. One uses

`all-formula-to-cases-const`.

(3) For a guarded general recursion constant: an all-formula. This form only occurs when translating a general induction axiom into a guarded general recursion constant, in order to achieve normalization of proofs via term normalization. We have to consider the free variables in the scheme formulas, and let the type of the guarded general recursion constant depend on them. This is needed to have the allnc-conversion be represented in term normalization. One uses

`all-formula-and-number-to-grecguard-const`.

(4) For an efq-constant (of kind `'fixed-rules`): a formula. This form only occurs when translating an efq-aconst into an efq-constant, in order to achieve normalization of proofs via term normalization. One uses

`formula-to-efq-const`.

(5) For a constructor associated with an "Intro" axiom.
   (a) A number i of a clause for an inductively defined predicate constant, and the constant idpc. One uses

`number-and-idpredconst-to-intro-const`.

(b) An ex-formula for an "Ex-Intro" axiom. One uses

`ex-formula-to-ex-intro-const`.

(c) An exnc-formula for an "Exnc-Intro" axiom. One uses

`exnc-formula-to-exnc-intro-const`.

(6) For an Ex-Elim constant (of kind `'fixed-rules`): an ex-formula and a conclusion. One uses

`ex-formula-and-concl-to-ex-elim-const`.

(7) For an Exnc-Elim constant (of kind `'fixed-rules`): an exnc-formula and a conclusion. One uses

`exnc-formula-and-concl-to-exnc-elim-const`.

Constructor, accessors and tests for all kinds of constants:

```
(make-const obj-or-arity name kind uninst-type tsubst
  t-deg token-type . repro-data)
(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-repro-data const)
(const? x)
(const=? x y)
```

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.

Similarly we have `arrow-types-to-cases-const` and on the proof level `all-formulas-to-cases-const`.

5. Predicate variables and constants

5.1. **Predicate variables.** A predicate variable of arity $\rho_1, \dots, \rho_n$ is a placeholder for a formula A with distinguished (different) variables $x_1, \dots, x_n$ of types $\rho_1, \dots, \rho_n$. 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, ...` range over comprehension terms with Harrop formulas, and `P0, P1, P2, ..., Q0, ...` are general predicate variables. We say that *Harrop degree* for the former is 1, and for the latter 0.

In the context of Gödel's Dialectica intepretation [11] we also need to deal with "negative" computational content. Therefore we also need a "degree of negativity" and denote it by `n-deg`, and we call the Harrop degree the "degree of positivity" denoted `h-deg`. We use `P'0, P'1, P'2, ..., Q'0, ...` for predicate variables of `h-deg` 0 and `n-deg` 1, and `P^'0, P^'1, P^'2, ...` for predicate variables whose `h-deg` and `n-deg` are both 1.

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 n-deg name)`	constructor
`(pvar-to-arity pvar)`	accessor
`(pvar-to-index pvar)`	accessor
`(pvar-to-h-deg pvar)`	accessor
`(pvar-to-n-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> <n-deg> <name>)
```

The arity is a required argument; the remaining arguments are optional. The default for *index* is -1, for *h-deg* and *n-deg* is 0 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 sometimes we wants predicates to be axiomatized, 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*`).`

5.3. **Inductively defined predicate constants.** When we want to make propositions about computable functionals and their domains of partial continuous functionals, it is perfectly natural to take, as initial propositions, ones formed inductively. For example, in the algebra **N** we can inductively define *totality* by the clauses

$$T0, \qquad \forall_n(Tn \to T(\mathrm{S}n)).$$

Its least-fixed-point scheme will be taken in the form

$$\forall_n(Tn \to A(0) \to \forall_n(Tn \to A(n) \to A(\mathrm{S}n)) \to A(n)).$$

The reason for writing it in this way is that it fits better with the logical elimination rules. It expresses that every "competitor" $\{\, n \mid A(n) \,\}$ satisfying the same clauses contains T. This is the usual induction schema for natural numbers, which clearly only holds for "total" numbers. Notice that we have used a "strengthened" form of the "step formula", namely $\forall_n(Tn \to A(n) \to A(\mathrm{S}n))$ rather than $\forall_n(A(n) \to A(\mathrm{S}n))$. In applications of the least-fixed-point axiom this simplifies the proof of the "induction step", since we have the additional hypothesis $T(n)$ available. Totality for an arbitrary algebra can be defined similarly.

Generally, an inductively defined predicate I is given by k clauses, which are of the form

$$\forall_{\vec{x}}(\vec{A}_i \to (\forall_{\vec{y}_{i\nu}}(\vec{B}_{i\nu} \to I\vec{s}_{i\nu}))_{\nu<n_i} \to I\vec{t}_i) \quad (i < k).$$

Our formulas will be defined by the operations of implication $A \to B$ and universal quantification $\forall_{x^\rho} A$ from inductively defined predicates $\mu_X \vec{K}$, where X is a predicate variable, and the K_i are clauses. Formulas will be treated more extensively later in section 7. However, in principle predicates and formulas are introduced simultaneously.

Definition (Predicates and formulas)**.** Let $X, \vec{Y}$ be distinct predicate variables; the Y_l are called *predicate parameters*. We inductively define *formula*

forms $A, B, C, D \in \mathrm{F}(\vec{Y})$, *predicate forms* $P, Q, I, J \in \mathrm{Preds}(\vec{Y})$ and *clause forms* $K \in \mathrm{Cl}_X(\vec{Y})$; all these are called *strictly positive* in $\vec{Y}$. In case $\vec{Y}$ is empty we abbreviate $\mathrm{F}(\vec{Y})$ by F and call its elements *formulas*; similarly for the other notions. (However, for brevity we often say "formula" etc. when it is clear from the context that parameters may occur.)

$$Y_l\vec{r} \in \mathrm{F}(\vec{Y}), \qquad \frac{A \in \mathrm{F} \quad B \in \mathrm{F}(\vec{Y})}{A \to B \in \mathrm{F}(\vec{Y})}, \qquad \frac{A \in \mathrm{F}(\vec{Y})}{\forall_x A \in \mathrm{F}(\vec{Y})},$$

$$\frac{C \in \mathrm{F}(\vec{Y})}{\{\,\vec{x} \mid C\,\} \in \mathrm{Preds}(\vec{Y})}, \qquad \frac{P \in \mathrm{Preds}(\vec{Y})}{P\vec{r} \in \mathrm{F}(\vec{Y})},$$

$$\frac{K_0, \dots, K_{k-1} \in \mathrm{Cl}_X(\vec{Y})}{\mu_X(K_0, \dots, K_{k-1}) \in \mathrm{Preds}(\vec{Y})} \quad (k \geq 1),$$

$$\frac{\vec{A} \in \mathrm{F}(\vec{Y}) \qquad \vec{B}_0, \dots, \vec{B}_{n-1} \in \mathrm{F}}{\forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_\nu}(\vec{B}_\nu \to X\vec{s}_\nu))_{\nu<n} \to X\vec{t}\,) \in \mathrm{Cl}_X(\vec{Y})} \quad (n \geq 0).$$

Here $\vec{A} \to B$ means $A_0 \to \cdots \to A_{n-1} \to B$, associated to the right. For a clause $\forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_\nu}(\vec{B}_\nu \to X\vec{s}_\nu))_{\nu<n} \to X\vec{t}\,) \in \mathrm{Cl}_X(\vec{Y})$ we call $\vec{A}$ *parameter* premises and $\forall_{\vec{y}_\nu}(\vec{B}_\nu \to X\vec{s}_\nu)$ *recursive* premises. We require that in $\mu_X(K_0, \dots, K_{k-1})$ the clause K_0 is "nullary", without recursive premises. The terms $\vec{r}$ are those introduced in section 6, i.e., typed terms built from constants by abstraction and application, and (importantly) those with a common reduct are identified.

A predicate of the form $\{\,\vec{x} \mid C\,\}$ is called a *comprehension term*. We identify $\{\,\vec{x} \mid C(\vec{x}\,)\,\}\vec{r}$ with $C(\vec{r}\,)$. The letter I will be used for predicates of the form $\mu_X(K_0, \dots, K_{k-1})$; they are called *inductively defined predicates*.

Remark (Substitution for predicate parameters). Let $A \in \mathrm{F}(\vec{Y})$; we write $A(\vec{Y})$ for A to indicate its dependence on the predicate parametes $\vec{Y}$. Similarly we write $I(\vec{Y})$ for I if $I \in \mathrm{Preds}(\vec{Y})$. We can substitute predicates $\vec{P}$ for $\vec{Y}$, to obtain $A(\vec{P})$ and $I(\vec{P})$, respectively.

An inductively defined predicate is *finitary* if its clauses have recursive premises of the form $X\vec{s}$ only (so the $\vec{y}_\nu$ and $\vec{B}_\nu$ in the general definition are all empty).

To introduce inductively defined predicates we use

`add-ids`.

An example is

```
(add-ids (list (list "Even" (make-arity (py "nat")) "nat"))
         '("Even 0" "InitEven")
```

```
'("allnc n^(Even n^ -> Even(n^ +2))" "GenEven"))
```

This simultaneously introduces the inductively defined predicate constant `Even`, by the clauses given. The presence of an algebra name after the arity (here `nat`) 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 `InitEven`, `GenEven`). We will also allow special *computationally irrelevant* (c.i.) inductively defined predicates.

An inductively defined predicate constant can only be understood from its clauses and its elimination or least-fixed-point axiom.

Definition (Theory of Computable Functionals TCF)**.** TCF is the system in minimal logic for $\to$ and $\forall$, whose formulas are those in F above, and whose axioms are the following. For each inductively defined predicate, there are "closure" or introduction axioms, together with a "least-fixed-point" or elimination axiom. In more detail, consider an inductively defined predicate $I := \mu_X(K_0, \dots K_{k-1})$. For each of the k clauses we have an introduction axiom, as follows. Let the i-th clause for I be

$$K_i(X) := \forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_\nu}(\vec{B}_\nu \to X\vec{s}_\nu))_{\nu<n} \to X\vec{t}\,).$$

Then the corresponding *introduction axiom* is $K_i(I)$, that is,

$$\forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_\nu}(\vec{B}_\nu \to I\vec{s}_\nu))_{\nu<n} \to I\vec{t}\,). \tag{4}$$

The *elimination axiom* is

$$\forall_{\vec{x}}(I\vec{x} \to (K_i(I,P))_{i<k} \to P\vec{x}\,), \tag{5}$$

where

$$K_i(I,P) := \forall_{\vec{x}}(\vec{A} \to (\forall_{\vec{y}_\nu}(\vec{B}_\nu \to I\vec{s}_\nu))_{\nu<n} \to (\forall_{\vec{y}_\nu}(\vec{B}_\nu \to P\vec{s}_\nu))_{\nu<n} \to P\vec{t}\,).$$

We label each introduction axiom $K_i(I)$ by I_i^+ and the elimination axiom by I^-.

As an important example we now give the inductive definition of Leibniz equality. However, a word of warning is in order here: we need to distinguish four separate, but closely related equalities.

(i) Firstly, defined function constants D are introduced by computation rules, written $l = r$, but intended as left-to-right rewrites.
(ii) Secondly, we have Leibniz equality Eq inductively defined below.
(iii) Thirdly, pointwise equality between partial continuous functionals will be defined inductively as well.

(iv) Fourthly, if l and r have a finitary algebra as their type, $l = r$ can be read as a boolean term, where $=$ is the decidable equality defined in 6 as a boolean-valued binary function.

Leibniz equality. We define Leibniz equality by

$$\mathrm{Eq}(\rho) := \mu_X(\forall_x X(x^\rho, x^\rho)).$$

The introduction axiom is

$$\forall_x \mathrm{Eq}(x^\rho, x^\rho)$$

and the elimination axiom

$$\forall_{x,y}(\mathrm{Eq}(x, y) \to \forall_x Pxx \to Pxy),$$

where $\mathrm{Eq}(x, y)$ abbreviates $\mathrm{Eq}(\rho)(x^\rho, y^\rho)$.

Lemma (Compatibility of Eq)**.** $\forall_{x,y}(\mathrm{Eq}(x, y) \to A(x) \to A(y))$.

Proof. Use the elimination axiom with $Pxy := (A(x) \to A(y))$. □

Using compatibility of Eq one easily proves symmetry and transitivity. Define *falsity* by $\mathbf{F} := \mathrm{Eq}(\mathtt{ff}, \mathtt{tt})$. Then we have

Theorem (Ex-Falso-Quodlibet)**.** *For every formula A without predicate parameters we can derive $\mathbf{F} \to A$.*

Proof. We first show that $\mathbf{F} \to \mathrm{Eq}(x^\rho, y^\rho)$. To see this, one first obtains $\mathrm{Eq}([\textbf{if } \mathtt{ff} \textbf{ then } x \textbf{ else } y], [\textbf{if } \mathtt{ff} \textbf{ then } x \textbf{ else } y])$ from the introduction axiom, since $[\textbf{if } \mathtt{ff} \textbf{ then } x \textbf{ else } y]$ is an allowed term, and then from $\mathrm{Eq}(\mathtt{ff}, \mathtt{tt})$ one gets $\mathrm{Eq}([\textbf{if } \mathtt{tt} \textbf{ then } x \textbf{ else } y], [\textbf{if } \mathtt{ff} \textbf{ then } x \textbf{ else } y])$ by compatibility. Hence $\mathrm{Eq}(x^\rho, y^\rho)$.

The claim can now be proved by induction on $A \in \mathrm{F}$. *Case* $I\vec{s}$. Let K_i be the nullary clause, with final conclusion $I\vec{t}$. By induction hypothesis from $\mathbf{F}$ we can derive all parameter premises. Hence $I\vec{t}$. From $\mathbf{F}$ we also obtain $\mathrm{Eq}(s_i, t_i)$, by the remark above. Hence $I\vec{s}$ by compatibility. The cases $A \to B$ and $\forall_x A$ are obvious. □

A crucial use of the equality predicate Eq is that it allows to lift a boolean term $r^{\mathbf{B}}$ to a formula, using $\mathrm{atom}(r^{\mathbf{B}}) := \mathrm{Eq}(r^{\mathbf{B}}, \mathtt{tt})$. This opens up a convenient way to deal with equality on finitary algebras. The computation rules ensure that for instance the boolean term $\mathrm{S}r =_{\mathbf{N}} \mathrm{S}s$ or more precisely, $=_{\mathbf{N}}(\mathrm{S}r, \mathrm{S}s)$, is identified with $r =_{\mathbf{N}} s$. We can now turn this boolean term into the formula $\mathrm{Eq}(\mathrm{S}r =_{\mathbf{N}} \mathrm{S}s, \mathtt{tt})$, which again is abbreviated by $\mathrm{S}r =_{\mathbf{N}} \mathrm{S}s$, but this time with the understanding that it is a formula. Then (importantly) the two formulas $\mathrm{S}r =_{\mathbf{N}} \mathrm{S}s$ and $r =_{\mathbf{N}} s$ are identified because the latter is a reduct of the first. Consequently there is no need to prove the implication $\mathrm{S}r =_{\mathbf{N}} \mathrm{S}s \to r =_{\mathbf{N}} s$ explicitly.

Pointwise equality $=_\rho$. For every constructor C_i of an algebra ι we have an introduction axiom

$$\forall_{\vec{y},\vec{z}}(\vec{y}^P =_{\vec{\rho}} \vec{z}^P \to (\forall_{\vec{x}_\nu}(y^R_{m+\nu}\vec{x}_\nu =_\iota z^R_{m+\nu}\vec{x}_\nu))_{\nu<n} \to \mathrm{C}_i\vec{y} =_\iota \mathrm{C}_i\vec{z}).$$

For an arrow type $\rho \to \sigma$ the introduction axiom is explicit, in the sense that it has no recursive premise:

$$\forall_{x_1,x_2}(\forall_y(x_1y =_\sigma x_2y) \to x_1 =_{\rho\to\sigma} x_2).$$

For example, $=_{\mathbf{N}}$ is inductively defined by

$$0 =_{\mathbf{N}} 0,$$
$$\forall_{n_1,n_2}(n_1 =_{\mathbf{N}} n_2 \to \mathrm{S}n_1 =_{\mathbf{N}} \mathrm{S}n_2),$$

and the elimination axiom is

$$\forall_{n_1,n_2}(n_1 =_{\mathbf{N}} n_2 \to P00 \to \forall_{n_1,n_2}(n_1 =_{\mathbf{N}} n_2 \to Pn_1n_2 \to P(\mathrm{S}n_1, \mathrm{S}n_2)) \to Pn_1n_2).$$

The main purpose of pointwise equality is that it allows to formulate the extensionality axiom: we express the extensionality of our intended model by stipulating that pointwise equality is equivalent to Leibniz equality.

Axiom (Extensionality)**.** $\forall_{x_1,x_2}(x_1 =_\rho x_2 \leftrightarrow \mathrm{Eq}(x_1, x_2))$.

We write E-TCF when the extensionality axioms are present. – One of the main points of TCF is that it allows the logical connectives existence, conjunction and disjunction to be inductively defined as predicates. This was first discovered by Martin-Löf [15].

Existential quantifier.

$$\mathrm{Ex}(Y) := \mu_X(\forall_x(Yx^\rho \to X)).$$

The introduction axiom is

$$\forall_x(A \to \exists_x A),$$

where $\exists_x A$ abbreviates $\mathrm{Ex}(\{\, x^\rho \mid A \,\})$, and the elimination axiom is

$$\exists_x A \to \forall_x(A \to P) \to P.$$

Conjunction. We define

$$\mathrm{And}(Y, Z) := \mu_X(Y \to Z \to X).$$

The introduction axiom is

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

where $A \wedge B$ abbreviates $\mathrm{And}(\{ \mid A \}, \{ \mid B \})$, and the elimination axiom is

$$A \wedge B \to (A \to B \to P) \to P.$$

Disjunction. We define

$$\mathrm{Or}(Y, Z) := \mu_X(Y \to X, Z \to X).$$

The introduction axioms are

$$A \to A \vee B, \qquad B \to A \vee B,$$

where $A \vee B$ abbreviates $\mathrm{Or}(\{ \mid A \}, \{ \mid B \})$, and the elimination axiom is

$$A \vee B \to (A \to P) \to (B \to P) \to P.$$

Remark. Alternatively, disjunction $A \vee B$ could be defined by the formula $\exists_p((p \to A) \wedge (\neg p \to B))$ with p a boolean variable. However, for an analysis of the computational content of coinductively defined predicates it is better to define it inductively.

We give some more familiar examples of inductively defined predicates.

The even numbers. The introduction axioms are

$$\mathrm{Even}(0), \qquad \forall_n(\mathrm{Even}(n) \to \mathrm{Even}(\mathrm{S}(\mathrm{S}n)))$$

and the elimination axiom is

$$\forall_n(\mathrm{Even}(n) \to P0 \to \forall_n(\mathrm{Even}(n) \to Pn \to P(\mathrm{S}(\mathrm{S}n))) \to Pn).$$

Transitive closure. Let $\prec$ be a binary relation. The *transitive closure* of $\prec$ is inductively defined as follows. The introduction axioms are

$$\forall_{x,y}(x \prec y \to \mathrm{TC}(x, y)),$$
$$\forall_{x,y,z}(x \prec y \to \mathrm{TC}(y, z) \to \mathrm{TC}(x, z))$$

and the elimination axiom is

$$\begin{aligned}\forall_{x,y}(\mathrm{TC}(x, y) \to \forall_{x,y}(x \prec y \to Pxy) \to \\ \forall_{x,y,z}(x \prec y \to \mathrm{TC}(y, z) \to Pyz \to Pxz) \to \\ Pxy).\end{aligned}$$

It is defined by

```
(add-ids
 (list (list "TrCl" (make-arity (py "alpha") (py "alpha"))
       "algTrCl"))
 '("allnc x^,y^(R x^ y^ -> TrCl x^ y^)" "InitTrCl")
 '("allnc x^,y^,z^(R x^ y^ -> TrCl y^ z^ -> TrCl x^ z^)"
   "GenTrCl"))
```

Accessible part. Let $\prec$ again be a binary relation. The *accessible part* of $\prec$ is inductively defined as follows. The introduction axioms are

$$\forall_x(\mathbf{F} \to \mathrm{Acc}(x)),$$
$$\forall_x(\forall_{y\prec x}\mathrm{Acc}(y) \to \mathrm{Acc}(x)),$$

and the elimination axiom is

$$\forall_x(\mathrm{Acc}(x) \to \forall_x(\mathbf{F} \to Px) \to \forall_x(\forall_{y\prec x}\mathrm{Acc}(y) \to \forall_{y\prec x}Py \to Px) \to Px).$$

Its definition in Minlog is

```
(add-ids
 (list (list "Acc" (make-arity (py "alpha=>alpha=>boole")
                               (py "alpha"))
             "algAcc"))
 '("allnc r^,x^(F -> Acc r^ x^)" "EfqAcc")
 '("allnc r^,x^(all y^(r^ y^ x^ -> Acc r^ y^) -> Acc r^ x^)"
   "GenAccSup"))
```

We now come to the inductively defined totality predicates. The least-fixed-point axiom for T_ι will provide us with the induction axiom. Let us first look at some examples. We already have stated the clauses defining totality for the algebra $\mathbf{N}$:

$$T_{\mathbf{N}}0, \qquad \forall_n(T_{\mathbf{N}}n \to T_{\mathbf{N}}(\mathrm{S}n)).$$

The least-fixed-point axiom is

$$\forall_n(T_{\mathbf{N}}n \to P0 \to \forall_n(T_{\mathbf{N}}n \to Pn \to P(\mathrm{S}n)) \to Pn).$$

As an example of a finitary algebra with parameters consider $\mathbf{L}(\rho)$. The clauses for the predicate $T_{\mathbf{L}(\rho)}$ expressing structure-totality are

$$T_{\mathbf{L}(\rho)}(\mathrm{nil}), \qquad \forall_{x,l}(T_{\mathbf{L}(\rho)}l \to T_{\mathbf{L}(\rho)}(x :: l)),$$

with no assumptions on x. The least-fixed-point axiom is

$$\forall_l(T_{\mathbf{L}(\rho)}l \to P(\mathrm{nil}) \to \forall_{x,l}(T_{\mathbf{L}(\rho)}l \to Pl \to P(x :: l)) \to Pl^{\mathbf{L}(\rho)}).$$

Generally, for arbitrary types ρ we inductively define predicates G_ρ of totality and T_ρ of structure-totality, by induction on ρ. This definition is relative to an assignment of predicate variables G_α, T_α of arity (α) to type variables α.

Definition. In case $\iota \in \mathrm{Alg}(\vec{\alpha}\,)$ we have $\iota = \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$, with $\kappa_i = \vec{\rho} \to (\vec{\sigma}_\nu \to \xi)_{\nu<n} \to \xi$. Then $G_\iota := \mu_X(K_0, \ldots, K_{k-1})$, with

$$K_i := \forall_{\vec{x}}(G_{\vec{\rho}}\vec{x}^P \to (\forall_{\vec{y}_\nu}(G_{\vec{\sigma}_\nu}\vec{y}_\nu \to X(x_\nu^R\vec{y}_\nu)))_{\nu<n} \to X(\mathrm{C}_i\vec{x}\,)).$$

Similarly, $T_\iota := \mu_X(K'_0,\dots,K'_{k-1})$, with

$$K'_i := \forall_{\vec{x}}((\forall_{\vec{y}_\nu}(T_{\vec{\sigma}_\nu}\vec{y}_\nu \to X(x_\nu^R\vec{y}_\nu)))_{\nu<n^R} \to X(\mathrm{C}_i\vec{x}\,)).$$

For arrow types the definition is *explicit*, that is, the clauses have no recursive premises but parameter premises only.

$$G_{\rho\to\sigma} := \mu_X\forall_f(\forall_x(G_\rho x \to G_\sigma(fx)) \to Xf),$$
$$T_{\rho\to\sigma} := \mu_X\forall_f(\forall_x(T_\rho x \to T_\sigma(fx)) \to Xf).$$

This concludes the definition.

In the case of an algebra ι the introduction axioms for T_ι are

$$(T_\iota)_i^+ : \forall_{\vec{x}}((\forall_{\vec{y}_\nu}(T_{\vec{\sigma}_\nu}\vec{y}_\nu \to T_\iota(x_\nu^R\vec{y}_\nu)))_{\nu<n} \to T_\iota(\mathrm{C}_i\vec{x}\,))$$

and the elimination axiom is

$$T_\iota^- : \forall_x(T_\iota x \to K_0(T_\iota,P) \to \cdots \to K_{k-1}(T_\iota,P) \to Px),$$

where

$$K_i(T_\iota,P) := \forall_{\vec{x}}((\forall_{\vec{y}_\nu}(T_{\vec{\sigma}_\nu}\vec{y}_\nu \to T_\iota(x_\nu^R\vec{y}_\nu)))_{\nu<n} \to (\forall_{\vec{y}_\nu}(T_{\vec{\sigma}_\nu}\vec{y}_\nu \to P(x_\nu^R\vec{y}_\nu)))_{\nu<n} \to P(\mathrm{C}_i\vec{x}\,)).$$

In the arrow type case, the introduction and elimination axioms are

$$\forall_x(T_\rho x \to T_\sigma(fx)) \to T_{\rho\to\sigma}f,$$
$$T_{\rho\to\sigma}f \to \forall_x(T_\rho x \to T_\sigma(fx)).$$

(The "official" axiom $T_{\rho\to\sigma}f \to (\forall_x(T_\rho x \to T_\sigma(fx)) \to P) \to P$ is clearly equivalent to one stated). Abbreviating $\forall_x(Tx \to A)$ by $\forall_{x\in T}A$ allows a shorter formulation of these axioms:

$$(\forall_{\vec{y}_\nu\in T_{\vec{\sigma}_\nu}} T_\iota(x_\nu^R\vec{y}_\nu))_{\nu<n} \to T_\iota(\mathrm{C}_i\vec{x}\,),$$
$$\forall_{x\in T_\iota}(K_0(T_\iota,P) \to \cdots \to K_{k-1}(T_\iota,P) \to Px),$$
$$\forall_{x\in T_\rho}T_\sigma(fx) \to T_{\rho\to\sigma}f,$$
$$\forall_{f\in T_{\rho\to\sigma},x\in T_\rho}T_\sigma(fx))$$

where

$$K_i(T_\iota,P) := \forall_{\vec{x}^P}\forall_{\vec{x}^R\in T_{\vec{\rho}}}((\forall_{\vec{y}_\nu\in T_{\vec{\sigma}_\nu}}P(x_\nu^R\vec{y}_\nu))_{\nu<n} \to P(\mathrm{C}_i\vec{x}\,)).$$

Hence the elimination axiom T_ι^- is the *induction* axiom, and the $K_i(T_\iota,P)$ are its *step formulas*. We write $\mathrm{Ind}_\iota^{x,P}$ or $\mathrm{Ind}_{x,P}$ for T_ι^-, and omit the indices x,P when they are clear from the context. Examples are

$$\mathrm{Ind}_{p,P} : \forall_{p\in T}(P\mathtt{tt} \to P\mathtt{ff} \to Pp^{\mathbf{B}}),$$
$$\mathrm{Ind}_{n,P} : \forall_{n\in T}(P0 \to \forall_{n\in T}(Pn \to P(\mathrm{S}n)) \to Pn^{\mathbf{N}}),$$
$$\mathrm{Ind}_{l,P} : \forall_{l\in T}(P(\mathrm{nil}) \to \forall_x\forall_{l\in T}(Pl \to P(x :: l)) \to Pl^{\mathbf{L}(\rho)}),$$

$$\mathrm{Ind}_{z,P}\colon \forall_{z\in T}(\forall_{x,y} P\langle x^\rho, y^\sigma\rangle \to Pz^{\rho\times\sigma}),$$

where $x :: l$ is shorthand for $\mathrm{cons}(x, l)$ and $\langle x, y\rangle$ for $\times^+ xy$.

All this can be done similarly for the G_ρ. A difference only occurs for algebras with parameters: for example, list induction then is

$$\forall_{l\in G}(P(\mathrm{nil}) \to \forall_{x,l\in G}(Pl \to P(x :: l)) \to Pl^{\mathbf{L}(\rho)}).$$

Parallel to general recursion, one can also consider *general induction*, which allows recurrence to *all* points "strictly below" the present one. For applications it is best to make the necessary comparisons w.r.t. a "measure function" μ. Then it suffices to use an initial segment of the ordinals instead of a well-founded set. For simplicity we here restrict ourselves to the segment given by ω, so the ordering we refer to is just the standard $<$-relation on the natural numbers. The principle of general induction then is

$$\forall_{\mu,x\in T}(\mathrm{Prog}^\mu_x Px \to Px) \tag{6}$$

where $\mathrm{Prog}^\mu_x Px$ expresses "progressiveness" w.r.t. the measure function μ and the ordering $<$:

$$\mathrm{Prog}^\mu_x Px := \forall_{x\in T}(\forall_{y\in T;\mu y<\mu x} Py \to Px).$$

It is easy to see that in our special case of the $<$-relation we can *prove* (6) from structural induction. However, it will be convenient to use general induction as a primitive axiom.

6. Terms and objects

6.1. **Constructors and accessors.** 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

`(make-term-in-var-form var)`	constructor
`(term-in-var-form-to-var` *`term`*`)`	accessor,

`(term-in-var-form? term)` test,

for constants

`(make-term-in-const-form const)`	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 [13]).

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).
```

6.2. **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` *name objs*`)`	constructor
`(nbe-constr-value-to-name` *value*`)`	accessor
`(nbe-constr-value-to-args` *value*`)`	accessor
`(nbe-constr-value-to-constr` *value*`)`	accessor
`(nbe-constr-value?` *value*`)`	test
`(nbe-fam-value?` *value*`)`	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 the 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? term)`	test
`(nbe-inst? constr-pattern obj)`	test
`(nbe-genargs constr-pattern obj)`	generalized arguments
`(nbe-extract termfam)`	extracts a term from a family
`(nbe-match pattern term)`	

Then we can define

`(nbe-reify object)`	reification
`(nbe-reflect term)`	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.

6.3. **Substitution.** Recall the generalities on substitutions in section 2.1. Under the conditions stated there on admissibility we define

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

Display functions for substitutions are

```
(pp-subst topsubst)
(display-substitutions topsubst)
(substitution-to-string subst)
```

7. Formulas and comprehension terms

A *prime formula* has the form `(predicate P r1 ... rn)` with a predicate variable or constant `P` and terms `r1` ...`rn`. *Formulas* are built from prime formulas by

(i) `(imp formula1 formula2)` implication
(ii) `(all x formula)` all quantification

(iii) `(impnc formula1 formula2)` implication without computational content
(iv) `(allnc x formula)` all quantification without computational content
(v) `(exca (x1 ...xn) formula)` classical existential quantification (with the arithmetical form of falsity **F**)
(vi) `(excl (x1 ...xn) formula)` classical existential quantification (with the logical form of falsity $\bot$).
(vii) `(tensor formula1 formula2)` tensor, for proper unfolding of formulas containing `exca` or `excl`.

We allow that quantified variables are formed without `^`, i.e., range 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. Notice that, since $\tilde{\exists}_x\tilde{\exists}_y A$ unfolds into a rather awkward formula, we have extended the $\tilde{\exists}$-terminology to lists of variables:

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

In this context the tensor connective (written $\tilde{\wedge}$) allows to abbreviate

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

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

Leibniz equality, the existential quantifier, conjunction and disjunction are provided by means of inductively defined predicates. Temporarily we still have built-in versions:

(i) `(and formula1 formula2)` conjunction
(ii) `(ex x formula)` existential quantification
(iii) `(exnc x formula)` existential quantification without computational content.

Temporarily we also allow prime formulas of the form `(atom r)` with a term `r` of type boole. They can be replaced by Leibniz equality of `r` with the boolean constant `True`, written `True eqd r`.

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
`(mk-ex` *var1 ... formula*`)`	ex-formula
`(mk-allnc` *var1 ... formula*`)`	allnc-formula
`(mk-exnc` *var1 ... formula*`)`	exnc-formula
`(mk-exca` *var1 ... formula*`)`	classical ex-formula (arithmetical)
`(mk-excl` *var1 ... formula*`)`	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

`(make-quant-formula` *quant var1 ... kernel*`)`	constructor
`(quant-form-to-quant` *quant-form*`)`	accessor
`(quant-form-to-vars` *quant-form*`)`	accessor
`(quant-form-to-kernel` *quant-form*`)`	accessor
`(quant-form?` *x*`)`	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.

Morever we need

```
(formula-to-free formula),
(formula-to-bound 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
`(cterm-to-vars cterm)`	accessor
`(cterm-to-formula cterm)`	accessor
`(cterm? x)`	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

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

We can simultaneously substitute for type, object and predicate variables in a formula or a comprehension term:

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

8. Assumption variables and constants

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 [17]) 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 implicitely 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, as `tpsubst`.

So we have assumption constants of the following kinds:

(i) axioms,
(ii) theorems, and
(iii) 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

`(aconst Ex-Intro` $\forall_{\hat{x}^\alpha}(P(\hat{x}) \to \exists_{\hat{x}^\alpha} P(\hat{x}))\ (\alpha \mapsto \tau, P^{(\alpha)} \mapsto \{\, \hat{z}^\tau \mid A \,\})$`)`

`(aconst Ex-Elim` $\exists_{\hat{x}^\alpha} P(\hat{x}) \to \forall_{\hat{x}^\alpha}(P(\hat{x}) \to \hat{Q}) \to \hat{Q}$
$(\alpha \mapsto \tau, P^{(\alpha)} \mapsto \{\, \hat{z}^\tau \mid A \,\}, \hat{Q} \mapsto \{\mid B \,\})$`)`

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 uninst-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 $\mathbf{N}$, $\mathbf{B}$. Let $\approx$ denote the equality relation in the model. Recall the domain of type $\mathbf{B}$, consisting of $\mathtt{tt}$, $\mathtt{ff}$ and the bottom element $\bot$. The boolean valued functions equality $=_{nat}\colon \mathbf{N} \to \mathbf{N} \to \mathbf{B}$ and existence (definedness, totality) $e_{nat}\colon \mathbf{N} \to \mathbf{B}$ need to be continuous. So we have

$$
\begin{aligned}
=(0,0) &\approx \mathtt{tt} \\
=(0,S\hat{n}) \approx =(S\hat{n},0) &\approx \mathtt{ff} & e(0) &\approx \mathtt{tt} \\
=(S\hat{n}_1,S\hat{n}_2) &\approx =(\hat{n}_1,\hat{n}_2) & e(S\hat{n}) &\approx e(\hat{n}) \\
=(\bot_{nat},\hat{n}) \approx =(\hat{n},\bot_{nat}) &\approx \bot & e(\bot_{\mathbf{N}}) &\approx \bot \\
=(\infty_{nat},\hat{n}) \approx =(\hat{n},\infty_{nat}) &\approx \bot & e(\infty_{\mathbf{N}}) &\approx \bot
\end{aligned}
$$

Write T, $\mathbf{F}$ for $\mathrm{atom}(\mathtt{tt})$, $\mathrm{atom}(\mathtt{ff})$, $r = s$ for $\mathrm{atom}(=(r,s))$ and $E(r)$ for $\mathrm{atom}(e(r))$. We require the following axioms. Notice that all these axioms become provable if we replace $\approx$ by the Leibniz equality (for `Eq-Ext` we need E-TCF).

T	`Truth-Axiom`
$\hat{x} \approx \hat{x}$	`Eq-Refl`
$\hat{x}_1 \approx \hat{x}_2 \to \hat{x}_2 \approx \hat{x}_1$	`Eq-Symm`
$\hat{x}_1 \approx \hat{x}_2 \to \hat{x}_2 \approx \hat{x}_3 \to \hat{x}_1 \approx \hat{x}_3$	`Eq-Trans`
$\forall_{\hat{x}}(\hat{f}_1\hat{x} \approx \hat{f}_2\hat{x}) \to \hat{f}_1 \approx \hat{f}_2$	`Eq-Ext`
$\hat{x}_1 \approx \hat{x}_2 \to P(\hat{x}_1) \to P(\hat{x}_2)$	`Eq-Compat`

$$T_{\rho\to\sigma}(\hat{f}) \leftrightarrow \forall_{\hat{x}}(T_\rho(\hat{x}) \to T_\sigma(\hat{f}\hat{x})) \quad \texttt{Total}$$
$$T_\rho(c) \quad \texttt{Constr-Total}$$
$$T(c\vec{\hat{x}}) \to T(\hat{x}_i) \quad \texttt{Constr-Total-Args}$$

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

$$\hat{n}_1 \approx \hat{n}_2 \to E(\hat{n}_1) \to \hat{n}_1 = \hat{n}_2 \quad \texttt{Eq-to-=-1-nat}$$
$$\hat{n}_1 \approx \hat{n}_2 \to E(\hat{n}_2) \to \hat{n}_1 = \hat{n}_2 \quad \texttt{Eq-to-=-2-nat}$$
$$\hat{n}_1 = \hat{n}_2 \to \hat{n}_1 \approx \hat{n}_2 \quad \texttt{=-to-Eq-nat}$$
$$\hat{n}_1 = \hat{n}_2 \to E(\hat{n}_1) \quad \texttt{=-to-E-1-nat}$$
$$\hat{n}_1 = \hat{n}_2 \to E(\hat{n}_2) \quad \texttt{=-to-E-2-nat}$$
$$T(\hat{n}) \to E(\hat{n}) \quad \texttt{Total-to-E-nat}$$
$$E(\hat{n}) \to T(\hat{n}) \quad \texttt{E-to-Total-nat}$$

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

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

We abbreviate

$$\forall_{\hat{x}}(T_\rho(\hat{x}) \to A) \quad \text{by} \quad \forall_x A,$$
$$\exists_{\hat{x}}(T_\rho(\hat{x}) \wedge A) \quad \text{by} \quad \exists_x A.$$

Formally, these abbreviations appear as axioms

$$\forall_x P(x) \to \forall_{\hat{x}}(T(\hat{x}) \to P(\hat{x})) \quad \texttt{All-AllPartial}$$
$$\forall_{\hat{x}}(T(\hat{x}) \to P(\hat{x})) \to \forall_x P(x) \quad \texttt{AllPartial-All}$$
$$\exists_x P(x) \to \exists_{\hat{x}}(T(\hat{x}) \wedge P(\hat{x})) \quad \texttt{Ex-ExPartial}$$
$$\exists_{\hat{x}}(T(\hat{x}) \wedge P(\hat{x})) \to \exists_x P(x) \quad \texttt{ExPartial-Ex}$$

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

$$\forall_n P(n) \to \forall_{\hat{n}}(E(\hat{n}) \to P(\hat{n})) \quad \texttt{All-AllPartial-nat}$$
$$\exists_{\hat{n}}(E(\hat{n}) \wedge P(\hat{n})) \to \exists_n P(n) \quad \texttt{ExPartial-Ex-nat}$$

Notice that `AllPartial-All-nat` i.e., $\forall_{\hat{n}}(E(\hat{n}) \to P(\hat{n})) \to \forall_n P(n)$ is provable (since $E(n) \mapsto T$). Similarly, `Ex-ExPartial-nat`, i.e., $\exists_n P(n) \to \exists_{\hat{n}}(E(\hat{n}) \wedge P(\hat{n}))$ is provable.

Finally we have axioms for the existential quantifier

$$\forall_{\hat{x}^\alpha}(P(\hat{x}) \to \exists_{\hat{x}^\alpha} P(\hat{x})) \quad \texttt{Ex-Intro}$$

$$\exists_{\hat{x}^\alpha} P(\hat{x}) \to \forall_{\hat{x}^\alpha}(P(\hat{x}) \to \hat{Q}) \to \hat{Q} \quad \texttt{Ex-Elim}$$

The assumption constants corresponding to these axioms are

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

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

$$\kappa_i = \vec{\rho} \to (\vec{\sigma}_1 \to \xi_{j_1}) \to \cdots \to (\vec{\sigma}_n \to \xi_{j_n}) \to \xi_j \in \mathrm{KT}_{\vec{\xi}}$$

we have the *step formula*

$$\begin{aligned} D_i := \forall_{y_1^{\rho_1},\ldots,y_m^{\rho_m},y_{m+1}^{\vec{\sigma}_1\to\mu_{j_1}},\ldots,y_{m+n}^{\vec{\sigma}_n\to\mu_{j_n}}} (&\forall_{\vec{x}^{\vec{\sigma}_1}} P_{j_1}(y_{m+1}\vec{x}) \to \cdots \to \\ &\forall_{\vec{x}^{\vec{\sigma}_n}} P_{j_n}(y_{m+n}\vec{x}) \to \\ &P_j(\mathrm{C}_i^{\vec{\mu}}(\vec{y}))). \end{aligned}$$

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

$$\forall_{\vec{x}^{\vec{\sigma}_1}} P_{j_1}(y_{m+1}\vec{x}), \dots, \forall_{\vec{x}^{\vec{\sigma}_n}} P_{j_n}(y_{m+n}\vec{x})$$

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

$$\mathrm{Ind}_{\mu_j} \colon D_1 \to \cdots \to D_k \to \forall_{x_j^{\mu_j}} P_j(x_j).$$

We will often write Ind_j for Ind_{μ_j}.

An example is

$$E_1 \to E_2 \to E_3 \to E_4 \to \forall_{x_1^{\mathbf{T}}} P_1(x_1)$$

with

$$\begin{aligned}
&E_1 := P_1(\mathrm{Leaf}),\\
&E_2 := \forall_{x^{\mathbf{Ts}}}(P_2(x) \to P_1(\mathrm{Branch}(x))),\\
&E_3 := P_2(\mathrm{Empty}),\\
&E_4 := \forall_{x_1^{\mathbf{T}}, x_2^{\mathbf{Ts}}}(P_1(x_1) \to P_2(x_2) \to P_2(\mathrm{Tcons}(x_1, x_2))).
\end{aligned}$$

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

$$\begin{aligned}
&\texttt{(aconst Ind } E_1 \to E_2 \to E_3 \to E_4 \to \forall_{x_1^{\mathbf{T}}} P_1(x_1)\\
&\qquad\qquad (P_1 \mapsto \{\, x_1^{\mathbf{T}} \mid A_1 \,\}, P_2 \mapsto \{\, x_2^{\mathbf{Ts}} \mid A_2 \,\})\texttt{)}
\end{aligned}$$

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

$$\texttt{(aconst Cases } E_1 \to E_2 \to \forall_{x_1^{\mathbf{T}}} P_1(x_1) \ (P_1 \mapsto \{\, x_1^{\mathbf{T}} \mid A_1 \,\})\texttt{)}$$

with

$$\begin{aligned}
&E_1 := P_1(\mathrm{Leaf}),\\
&E_2 := \forall_x^{\mathbf{Ts}} P_1(\mathrm{Branch}(x)).
\end{aligned}$$

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 $I(\vec{x}) \to A$.

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 explicitely 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 ...)
(pp theorem-name)
```

Initially we provide the following theorems

$\mathrm{atom}(p) \to p = \mathtt{tt}$	`Atom-True`
$(\mathrm{atom}(p) \to \mathbf{F}) \to p = \mathtt{ff}$	`Atom-False`
$\mathbf{F} \to \mathrm{atom}(p)$	`Efq-Atom`
$((\mathrm{atom}(p) \to \mathbf{F}) \to \mathbf{F}) \to \mathrm{atom}(p)$	`Stab-Atom`

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

$$n = n \qquad \texttt{=-Refl-nat}$$
$$\hat{n}_1 = \hat{n}_2 \to \hat{n}_2 = \hat{n}_1 \qquad \texttt{=-Symm-nat}$$
$$\hat{n}_1 = \hat{n}_2 \to \hat{n}_2 = \hat{n}_3 \to \hat{n}_1 = \hat{n}_3 \qquad \texttt{=-Trans-nat}$$

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

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

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

Proof of `Stab-Atom`. By `Ind`. In case $\mathtt{tt}$ use `Truth-Axiom`, and the case $\mathtt{ff}$ is obvious. □

Remark. Notice that from `Efq-Atom` one easily obtains $\mathbf{F} \to A$ for every formula A all whose strictly positions occurrences of prime formulas are of the form $\mathrm{atom}(r)$, by induction on A. For all other formulas we shall make use of the global assumption `Efq`: $\mathbf{F} \to P$ (cf. section 8.4). Similarly, notice that from `Stab-Atom` one again obtains $((A \to \mathbf{F}) \to \mathbf{F}) \to A$ for every formula A all whose strictly positions occurrences of prime formulas are of the form $\mathrm{atom}(r)$, by induction on A. For all other formulas we shall make use of the global assumption `Stab`: $((P \to \mathbf{F}) \to \mathbf{F}) \to P$ (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 ${=}(\bot, \bot) \approx \bot$. □

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 $f^{\alpha \to \mathbf{N}} \mapsto g^{\rho \to \mathbf{N}}$ and a predicate variable substitution $P^{(\alpha)} \mapsto \{\, \hat{z}^{\rho} \mid A \,\}$.

`(aconst Cvind-with-measure-11`

$$\forall_f^{\alpha \to \mathbf{N}}(\forall_{x^\alpha}(\forall_y(f(y) {<} f(x) \to P(y)) \to P(x)) \to \forall_x P(x))$$
$$(\alpha \mapsto \rho, f^{\alpha \to \mathbf{N}} \mapsto g^{\rho \to \mathbf{N}}, P^{(\alpha)} \mapsto \{\, \hat{z}^{\rho} \mid A \,\}))\,.$$

`(aconst Minpr-with-measure-l11`

$$\forall_{f^{\alpha \to \mathbf{N}}}(\tilde{\exists}_{x^\alpha} P(x) \to \tilde{\exists}_x(P(x) \mathbin{\tilde{\wedge}} \forall_y(f(y) {<} f(x) \to P(y) \to \bot)))$$
$$(\alpha \mapsto \rho, f^{\alpha \to \mathbf{N}} \mapsto g^{\rho \to \mathbf{N}}, P^{(\alpha)} \mapsto \{\, \hat{z}^{\rho} \mid A \,\}))\,.$$

Here $\tilde{\exists}$ is the classical existential quantifier defined by $\tilde{\exists}_x A := \forall_x(A \to \bot) \to \bot$ with the logical form of falsity $\bot$ (as opposed to the arithmetical form $\mathbf{F}$). `1` indicates "logic" (we have used the logical form of falsity), the first `1` that we have one predicate variable 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

$$(P \to P_1) \to ((P \to \bot) \to P_1) \to P_1 \quad \texttt{Cases-Log}$$

The proof uses the global assumption `Stab-Log` (see below) for 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)
```

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).

$\bot \to P$	`Efq-Log`
$((P \to \bot) \to \bot) \to P$	`Stab-Log`
$\mathbf{F} \to P$	`Efq`
$((P \to \mathbf{F}) \to \mathbf{F}) \to P$	`Stab`

The assumption constants corresponding to these global assumptions are generated by

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

9. Proofs

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

`(make-proof-in-avar-form` *avar*`)`	constructor
`(proof-in-avar-form-to-avar` *proof*`)`	accessor,
`(proof-in-avar-form?` *proof*`)`	test,

for assumption constants

`(make-proof-in-aconst-form` *aconst*`)`	constructor
`(proof-in-aconst-form-to-aconst` *proof*`)`	accessor
`(proof-in-aconst-form?` *proof*`)`	test,

for implication introduction

`(make-proof-in-imp-intro-form` *avar proof*`)`	constructor
`(proof-in-imp-intro-form-to-avar` *proof*`)`	accessor
`(proof-in-imp-intro-form-to-kernel` *proof*`)`	accessor
`(proof-in-imp-intro-form?` *proof*`)`	test,

for implication elimination

`(make-proof-in-imp-elim-form` *proof1 proof2*`)`	constructor
`(proof-in-imp-elim-form-to-op` *proof*`)`	accessor
`(proof-in-imp-elim-form-to-arg` *proof*`)`	accessor
`(proof-in-imp-elim-form?` *proof*`)`	test,

for and introduction

`(make-proof-in-and-intro-form` *proof1 proof2*`)`	constructor
`(proof-in-and-intro-form-to-left` *proof*`)`	accessor
`(proof-in-and-intro-form-to-right` *proof*`)`	accessor
`(proof-in-and-intro-form?` *proof*`)`	test,

for and elimination

`(make-proof-in-and-elim-left-form` *proof*`)`	constructor
`(make-proof-in-and-elim-right-form` *proof*`)`	constructor
`(proof-in-and-elim-left-form-to-kernel` *proof*`)`	accessor
`(proof-in-and-elim-right-form-to-kernel` *proof*`)`	accessor
`(proof-in-and-elim-left-form?` *proof*`)`	test
`(proof-in-and-elim-right-form?` *proof*`)`	test,

for all introduction

`(make-proof-in-all-intro-form` *var proof*`)`	constructor
`(proof-in-all-intro-form-to-var` *proof*`)`	accessor
`(proof-in-all-intro-form-to-kernel` *proof*`)`	accessor
`(proof-in-all-intro-form?` *proof*`)`	test,

for all elimination

`(make-proof-in-all-elim-form` *proof term*`)`	constructor
`(proof-in-all-elim-form-to-op` *proof*`)`	accessor
`(proof-in-all-elim-form-to-arg` *proof*`)`	accessor
`(proof-in-all-elim-form?` *proof*`)`	test

and for `cases`-constructs

`(make-proof-in-cases-form` *test alt1 ...*`)`	constructor
`(proof-in-cases-form-to-test` *proof*`)`	accessor
`(proof-in-cases-form-to-alts` *proof*`)`	accessor
`(proof-in-cases-form-to-rest` *proof*`)`	accessor
`(proof-in-cases-form?` *proof*`)`	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-aconsts-without-rules proof).
(proof-to-aconsts proof).
```

To work with contexts we need

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

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

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

We show that such an optimal decoration exists, and give an algorithm to construct it.

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

The *proof pattern* $\mathrm{P}(M)$ of a proof M is the result of marking in c.r. formulas of M (i.e., those not above a c.i. formula) all occurrences of implications and universal quantifiers as non-computational, except the "uninstantiated" formulas of axioms and theorems. For instance, the induction axiom for $\mathbf{N}$ consists of the uninstantiated formula $\forall^{\mathrm{c}}_n(P0 \to^{\mathrm{c}} \forall^{\mathrm{c}}_n(Pn \to^{\mathrm{c}} P(\mathrm{S}n)) \to^{\mathrm{c}} Pn^{\mathbf{N}})$ with a unary predicate variable P and a predicate substitution $P \mapsto \{\, x \mid A(x) \,\}$. Notice that a proof pattern in most cases is not a correct proof, because at axioms formulas may not fit.

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

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

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

We assume that every axiom has the property that for every extension of its formula we can find a further extension which is an instance of an axiom, and which is the least one under all further extensions that are instances of axioms. This property clearly holds for axioms whose uninstantiated formula only has the decorated $\to^{\mathrm{c}}$ and $\forall^{\mathrm{c}}$, for instance induction. However, in $\forall^{\mathrm{c}}_n(A(0) \to^{\mathrm{c}} \forall^{\mathrm{c}}_n(A(n) \to^{\mathrm{c}} A(\mathrm{S}n)) \to^{\mathrm{c}} A(n^{\mathbf{N}}))$ the given extension of the four A's might be different. One needs to pick their "least upper bound" as further extension. To make this assumption true for the other (introduction and elimination) axioms we simply add all their extensions as axioms, if necessary.

One can define a *decoration algorithm* [20], assigning to every proof pattern U and every extension of its sequent an "optimal" decoration M_∞ of U, which further extends the given extension of its sequent.

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

(a) $\mathrm{Seq}(M_\infty)$ *extends the given extension of* $\mathrm{Seq}(U)$*, and*

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

The main function for decorating is

```
decorate proof . opt-decfla-and-name-and-altname
```

The default case for `opt-decfla` is the formula of the proof. If `opt-decfla` is present, it must be a decoration variant of the formula of the proof. If the optional argument `name-and-altname` is present, then in every recursive call it is checked whether (1) the present proof is an application of the assumption constant `op` with `name` to some `args`, (2) `op` applied to `args` proves an extension of `decfla`, and (3) `altop` applied to `args` and some of `decavars` is between `op` applied to `args` and `decfla` w.r.t. extension. If so, a proof based on `altop` is returned, else one carries on.

An important auxiliary function is `proof-to-ppat`. used to transform a proof into its proof pattern. It turns every $\to$, $\forall$ formula in the given proof into an $\to^{\mathrm{nc}}$, $\forall^{\mathrm{nc}}$ formula, including the parts of an assumption constant which come from its uninstatiated formula. It does not touch the c.i. parts of the proof, i.e., those which are above a c.i. formula. Recall that the proof pattern `ppat` is in general not a proof.

9.3. **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 $\mathtt{x\hat{}0}, \mathtt{x\hat{}1}, \mathtt{x\hat{}2}, \ldots$ and variables $\mathtt{x0}, \mathtt{x1}, \mathtt{x2}, \ldots$ intended to range over total objects only, η-conversion of proofs cannot be done via reduction to η-conversion of terms. To see this, consider the proof

$$\dfrac{\dfrac{\dfrac{\forall_{\hat{x}} P\hat{x} \qquad x}{Px}}{\forall_x Px}}{\forall_{\hat{x}} P\hat{x} \to \forall_x Px}$$

The proof term is $\lambda_u\lambda_x(ux)$. If we η-normalize this to $\lambda_u u$, the proven formula would be all $\forall_{\hat{x}}P\hat{x} \to \forall_{\hat{x}}P\hat{x}$. 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)
(npterm-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`.

9.4. **Substitution.** In a proof we can substitute

(i) types for type variables (by a type variable substitution `tsubst`),
(ii) terms for variables (by a substitution `subst`),
(iii) comprehension terms for predicate variables (by a predicate variable substitution `psubst`), and
(iv) proofs for assumption variables (by a assumption variable substitution `asubst`).

All these substitutions can be packed together, as an argument `topasubst` for `proof-substitute`. It is assumed that `topasubst` is admissible, in the sense of section 2.1.

```
(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?)
```

9.5. **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`, abbreviated `cdp`. We also

provide a readable type-free lambda expression via `proof-to-expr`, and can add useful information with one of `proof-to-expr-with-formulas`, `proof-to-expr-with-aconsts`.

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.

`(display-proof . opt-proof)`	abbreviated `dp`
`(check-and-display-proof . opt-proof)`	abbreviated `cdp`
`(proof-to-expr . opt-proof)`	
`(proof-to-expr-with-formulas . opt-proof)`	
`(proof-to-expr-with-aconsts . opt-proof)`	
`(display-pterm . opt-proof)`	abbreviated `dpt`
`(display-proof-expr . opt-proof)`	abbreviated `dpe`

We also provide versions which normalize the proof first:

`(display-normalized-proof . opt-proof)`	abbreviated `dnp`
`(display-normalized-pterm . opt-proof)`	abbreviated `dnpt`
`(display-normalized-proof-expr . opt-proof)`	abbreviated `dnpe`

9.6. **Classical logic.** `(proof-of-stab-at formula)` generates a proof of $((A \to \mathbf{F}) \to \mathbf{F}) \to A$. For $\mathbf{F}$, 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`: $((P \to \mathbf{F}) \to \mathbf{F}) \to P$. 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

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)
```

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.

10.1. **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. **normalize-goal.** (normalize-goal . *ng-info*) (short: ng) takes optional arguments ng-info. If there are none, the goal formula and all hypotheses are normalized. Otherwise exactly those among the hypotheses

and the goal formula are normalized whose numbers (or names, or just `#t` for the goal formula) are listed as additional arguments.

10.3. **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.

10.4. **use.** In `(use` *`x . elab-path-and-terms`*`)`, x is one of the following.

(i) A number or string identifying a hypothesis form the context.
(ii) A formula with free variables from the context, generating a new goal.
(iii) The name of a theorem or global assumption.
(iv) A closed proof.

It is checked whether some final part of this used formula has the form of (or "matches") the goal, where if (i) x determines a hypothesis or is the formula for a new goal, then all its free topvars are rigid, and if (ii) x determines a closed proof, then all its (implicitely generalized) tpvars are flexible, except the predicate variable $\bot$ (written `bot`) from `falsity-log`. *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/cterms to be substituted for the variables that cannot be instantiated by matching. Matching is done for type and object variables first (via `match`), and in case this fails with `huet-match` next. There is a similar `(use2` *`x . elab-path-and-terms`*`)`, which only applies `huet-match`.

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

(i) A number or string identifying a hypothesis form the context.
(ii) The name of a theorem or global assumption. If it is a global assumption whose final conclusion is a nullary predicate variable distinct from bot (e.g. `Efq-Log` or `Stab-Log`), this predicate variable is substituted by the goal formula.
(iii) A closed proof.
(iv) A formula with free variables from the context, generating a new goal.

Moreover *x-list* is a list consisting of

(i) a number or string identifying a hypothesis form the context,
(ii) the name of a theorem or global assumption,
(iii) a closed proof,
(iv) the string "?" (value of `DEFAULT-GOAL-NAME`), generating a new goal,
(v) a symbol `left` or `right`,

(vi) a term, whose free variables are added to the context,
(vii) a type, which is substituted for the first type variable,
(viii) 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 $\forall_x Px \to \forall_x \neg Px \to \bot$).

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).

10.6. **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

(i) a number or string identifying a hypothesis form the context,
(ii) the name of a theorem or global assumption,
(iii) a closed proof,
(iv) a formula with free variables from the context, generating a new goal.

and *x-list* is a list consisting of

(i) a number or string identifying a hypothesis form the context,
(ii) the name of a theorem or global assumption,
(iii) a closed proof,
(iv) the string "?" (value of `DEFAULT-GOAL-NAME`), generating a new goal,
(v) a symbol `left` or `right`,
(vi) a term, whose free variables are added to the context,
(vii) a type, which is substituted for the first type variable,
(viii) a comprehension term, which is substituted for the first predicate variable.

10.7. **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.

10.8. **cut.** The command `(cut A)` replaces the goal B by the two new goals A and $A \to B$, with $A \to B$ to be proved first.

10.9. **assert.** The command `(assert A)` replaces the goal B by the two new goals A and $A \to B$, with A to be proved first.

10.10. **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)`.

10.11. **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.

10.12. **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 ith hypothesis.

10.13. **split, msplit.** The command `(split)` expects as goal a conjunction $A \wedge B$ or an `AndConst`-atom, and splits it into two new goals A and B. We allow multiple split `(msplit)` over a conjunctive formula (all conjuncts connected through `&` which are at the same level are split at once).

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

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

10.16. **ind.** `(ind)` expects a goal $\forall_{x^\iota} A(x)$ with x total and ι an algebra, or a goal $\forall_{\hat{x}^\iota}(\texttt{STotal}\ \hat{x} \to A(\hat{x}))$ with $\hat{x}$ partial. Let $c_1, \dots, c_n$ be the constructors of the algebra. In the first case, n new goals $\forall_{\vec{x}_i}(A(x_{1i}) \to \cdots \to A(x_{ki}) \to A(c_i\vec{x}_i)$ are generated. In the second case, for every non-parameter variable $\hat{x}_{ji}$ the new goal has an additional assumption `STotal` $\hat{x}_{ji}$.

`(ind t)` expects a goal $A(t)$. It computes the algebra ι as type of the term t. Then again the n new goals above are generated. If t is partial, another new goal `STotal` t is generated.

10.17. **simind.** `(simind all-formula1 ...)` expects a goal $\forall_{x^\iota} A(x)$ with x total and ι an algebra, or $\forall_{\hat{x}^\iota}(\texttt{STotal}\ \hat{x} \to A(\hat{x}))$ with $\hat{x}$ partial. We have to provide as arguments the other all-formulas to be proved simultaneously with the goal.

10.18. **gind.** `(gind h)` expects a goal $\forall_{\vec{x}} A(\vec{x}\,)$ with $\vec{x}$ total. It generates a new goal $\mathrm{Prog}_h\{\,\vec{x} \mid A(\vec{x}\,)\,\}$ where h is a term of type $\vec{\rho} \to \mathbf{N}$, x_i has type ρ_i and $\mathrm{Prog}_h\{\,\vec{x} \mid A(\vec{x}\,)\,\} := \forall_{\vec{x}}(\forall_{\vec{y}}(h\vec{y} < h\vec{x} \to A(\vec{y})) \to A(\vec{x}\,))$.

`(gind h t1 ...tn)` expects a goal $A(\vec{t}\,)$ and generates the same goal as for `(gind h)` with the formula $\forall_{\vec{x}} A(x)$.

10.19. **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). `(intro-with i . x-list)` does the same, via `use-with`.

10.20. **elim.** Recall that $I\vec{r}$ provides (i) a type substitution, (ii) a predicate instantiation, and (iii) the list $\vec{r}$ of argument terms. In `(elim idhyp)` *idhyp* is, with an inductively defined predicate I,

(i) a number or string identifying a hypothesis $I\vec{r}$ form the context
(ii) the name of a global assumption or theorem $I\vec{r}$;
(iii) a closed proof of a formula $I\vec{r}$;
(iv) a formula $I\vec{r}$ with free variables from the context, generating a new goal.

Then the (strengthened) elimination axiom is used with $\vec{r}$ for $\vec{x}$ and *idhyp* for $I\vec{r}$ to prove the goal $A(\vec{r})$, leaving the instantiated (with $\{\,\vec{x} \mid A(\vec{x})\,\}$) clauses as new goals.

`(elim)` expects a goal $I\vec{r} \to A(\vec{r})$. Then the (strengthened) clauses are generated as new goals, via `use-with`.

In case of simultaneously inductively defined predicate constants we can provide other imp-formulas to be proved simultaneously with the given one. Then the (strengthened) simplified clauses are generated as new goals.

10.21. **inversion, simplified-inversion.** `(inversion x . imp-formulas)` assumes that x is one of the following.

(i) A number or string identifying a hypothesis $I\vec{r}$ form the context.
(ii) The name of a theorem or global assumption stating $I\vec{r}$.
(iii) A closed proof of $I\vec{r}$.
(iv) A formula $I\vec{r}$ with free vars from the context, generating a new goal.

imp-formulas have the form $J\vec{s} \to B$. Here I, J are inductively defined predicates, with clauses $K_1, \dots, K_n$. Now one uses the elim-aconst for $I\vec{x} \to \vec{x} = \vec{r} \to A$ with A the goal formula and the additional implications $J\vec{y} \to \vec{y} = \vec{s} \to B$, with "?" for the clauses, $\vec{r}$ for $\vec{x}$ and proofs for $\vec{r} = \vec{r}$, to obtain the goal. Then many of the generated goals for the clauses will contain false premises, coming from substituted equations $\vec{x} = \vec{r}$, and are proved automatically.

imp-formulas not provided are taken as $J\vec{x} \to J\vec{x}$. Generated clauses for such J are proved automatically from the intro axioms (the rec-prems are not needed).

For simultaneous inductively defined predicates `(simplified-inversion x . imp-formulas)` does not add imp-formulas $J\vec{x} \to J\vec{x}$ to form the elim-aconst. Then the (new) `imp-formulas-to-uninst-elim-formulas-etc` generates simplified clauses. In some special cases this suffices.

10.22. **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.

10.23. **ex-elim.** In `(ex-elim x)`, x is

(i) a number or string identifying an existential hypothesis from the context,
(ii) the name of an existential global assumption or theorem,
(iii) a closed proof on an existential formula,
(iv) 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.

10.24. **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.

10.25. **cases.** `(cases)` expects a goal $\forall_{x^\iota} A(x)$ with x total and ι an algebra, or a goal $\forall_{\hat{x}^\iota}(\texttt{STotal}\ \hat{x} \to A(\hat{x}))$ with $\hat{x}$ partial. Let $c_1, \dots, c_n$ be the constructors of the algebra. In the first case, n new goals $\forall_{\vec{x}_i} A(c_i \vec{x}_i)$ are generated. In the second case, for every non-parameter variable $\hat{x}_{ji}$ the new goal has an additional assumption `STotal` $\hat{x}_{ji}$.

`(cases t)` expects a goal $A(t)$. If t is a total boolean term, the goal $A(t)$ is replaced by the two new goals $\mathrm{atom}(t) \to A(\mathtt{tt})$ and $(\mathrm{atom}(t) \to \mathbf{F}) \to A(\mathtt{ff})$, and if t is not total also `STotal` t. If t is a total non-boolean term, `cases` is called with the all-formula $\forall_x(x = t \to A(x))$, and if t is a non-total non-boolean term, `cases` is called with the all-formula $\forall_{\hat{x}}(\texttt{STotal}\ \hat{x} \to \hat{x} = t \to A(x))$.

`(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.

10.26. **casedist.** `(casedist t)` replaces the goal A containing a boolean term t by two new goals $\mathrm{atom}(t) \to A(\mathtt{tt})$ and $(\mathrm{atom}(t) \to \mathbf{F}) \to A(\mathtt{ff})$, and if t is not total also `STotal` t.

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

(i) a number or string identifying a hypothesis form the context,
(ii) the name of a theorem or global assumption,
(iii) a closed proof,
(iv) 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 as 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.

10.28. **simp-with.** This is a more verbose form of `simp`, where the terms are not inferred via matching, but have to be given explicitely. 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

(i) a number or string identifying a hypothesis form the context,
(ii) the name of a theorem or global assumption,
(iii) a closed proof,
(iv) the string "?" (value of `DEFAULT-GOAL-NAME`), generating a new goal,
(v) a symbol `left` or `right`,
(vi) a term, whose free variables are added to the context,
(vii) a type, which is substituted for the first type variable,
(viii) a comprehension term, which is substituted for the first predicate variable.

10.29. **simphyp-with.** `simphyp-with` does for forward chaining the same as `simp-with` for backward chaining. It replaces the present goal by a new one, with one additional hypothesis obtained by simplifying a previous one. Notice that this effect could also be obtained by `cut` or `assert`. In `(simphyp-with` *`opt-dir hyp . x-list`*`)`, *hyp* is one of the following.

(i) A number or string identifying a hypothesis form the context.
(ii) The name of a theorem or global assumption, but not one whose final conclusion is a predicate variable.
(iii) A closed proof.
(iv) A formula with free variables from the context, generating a new goal.

x-list is a list consisting of

(i) a number or string identifying a hypothesis form the context,
(ii) the name of a theorem or global assumption,
(iii) a closed proof,
(iv) the string "?" (value of `DEFAULT-GOAL-NAME`), generating a new goal,
(v) a symbol `left` or `right`,
(vi) a term, whose free variables are added to the context,
(vii) a type, which is substituted for the first type variable,
(viii) a comprehension term, which is substituted for the first predicate variable.

This generates a used formula, which is to be an atom, a negated atom or $t \approx s$. If it as 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.

10.30. **simphyp-with-to.** `simp-with-to` expects a string as its last argument, which is used (via `name-hyp`) to name the newly introduced simplified hypothesis.

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

(i) a number or string identifying a classical existential hypothesis from the context,
(ii) the name of a classical existential global assumption or theorem,
(iii) a closed proof on a classical existential formula,
(iv) 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.

We also provide `exc-formula-to-min-pr-proof`. It computes first a gind-aconst (an axiom or a theorem) and from this a proof of the minimum principle.

10.32. **by-assume-minimal-with.** For convenience in classical arguments there is `(by-assume-minimal-with exc-hyp . rest)` where *rest* may be called *varnames-and-measure-and-minhyp-and-hyps*. It is meant for the following situation. Suppose we are proving a goal G from a classical existential hypothesis $\tilde{\exists}_{\vec{x}}\vec{A}$. Then by the minimum principle we can assume that we have $\vec{x}$ which are minimal w.r.t. a measure h such that $\vec{A}$ are satisfied.

We also provide `make-gind-aconst`. It takes a positive integer n and returns an assumption constant for general induction w.r.t. a measure function of type $\alpha_1 \to \ldots \to \alpha_n \to \mathbf{N}$.

Finally we provide `make-min-pr-aconst`. It takes positive integers m, n and returns an assumption constant for the minimum principle w.r.t. a measure function of type $\alpha_1 \to \ldots \to \alpha_n \to \mathbf{N}$.

10.33. **exc-intro.** In `(exc-intro terms)`, the user provides terms to be used for the present (classical existential) goal. Moreover we also provide `make-exc-intro-aconst` and `exc-formula-to-exc-intro-aconst`

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

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

Let $\tilde{\exists}_{\vec{y}}\vec{A}$ 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. Moreover we also provide `make-exc-elim-aconst` and `exc-formula-to-exc-elim-aconst`.

10.35. **pair-elim.** In `(pair-elim)`, a goal $\forall_p P(p)$ is replaced by the new goal $\forall_{x_1,x_2} P(\langle x_1, x_2\rangle)$.

10.36. **admit.** `(admit)` temporarily accepts the present goal, by turning it into a global assumption.

11. Unification and proof search

We describe a proof search method suitable for minimal logic with higher order functionals. It is based on Huet's [12] unification algorithm for the simply typed lambda calculus.

Huet's unification algorithm does not terminate in general; this must be the case, since it is well known that higher order unification is undecidable. However, non-termination can be avoided if we restrict ourselves to a certain fragment of higher order (simply typed) minimal logic. This fragment is determined by requiring that every higher order variable Y can only occur in a context $Y\vec{x}$, where $\vec{x}$ are distinct bound variables in the scope of the operator binding Y, and of opposite polarity. Note that for first order logic this restriction does not mean anything, since there are no higher order variables. However, when designing a proof search algorithm for first order logic only, one is naturally led into this fragment of higher order logic, where the algorithm works as well.

In this section we only present the algorithms and state their properties. Proofs can be found in the accompanying document "A Theory of Computable Functionals", also in the Minlog distribution.

11.1. **Huet's unification algorithm.** We work in the simply typed λ-calculus, with the usual conventions. For instance, whenever we write a term we assume that it is correctly typed. *Substitutions* are denoted by φ, ψ, ρ. The result of applying a substitution φ to a term r or a formula A is written as $r\varphi$ or $A\varphi$, with the understanding that after the substitution all terms are brought into long normal form.

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. A variable is called *rigid* if it is either a signature variable or else a forbidden variable.

A *Q-term* is a term with all its free variables in Q, and similarly a *Q-formula* is a formula with all its free variables in Q. 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 now define the unification algorithm. It takes a unification problem $\mathcal{U} = QC$ and produces a not necessarily well-founded tree (called *matching tree* by Huet [12]) with nodes labelled by unification problems and vertices labelled by substitutions.

Definition (Unification algorithm)**.** We distinguish cases according to the form of the unification problem, and either give the transition done by the algorithm, or else state that it fails.

Case identity, i.e., $Q(r = r \wedge C)$. Then

$$Q(r = r \wedge C) \Longrightarrow_\varepsilon QC.$$

Case ξ, i.e., $Q(\lambda_{\vec{x}} r = \lambda_{\vec{x}} s \wedge C)$. We may assume here that the bound variables $\vec{x}$ are the same on both sides.

$$Q(\lambda_{\vec{x}}\, r = \lambda_{\vec{x}}\, s \wedge C) \Longrightarrow_\varepsilon Q\forall_{\vec{x}}(r = s \wedge C).$$

Case rigid-rigid, i.e., $Q(f\vec{r} = g\vec{s} \wedge C)$ with both f and g rigid, that is either a signature variable or else a forbidden variable. If f is different from

g then fail. If f equals g,

$$Q(f\vec{r} = f\vec{s} \wedge C) \Longrightarrow_\varepsilon Q(\vec{r} = \vec{s} \wedge C).$$

Case flex-rigid, i.e., $Q(u\vec{r} = f\vec{s} \wedge C)$ with f rigid. Then the algorithm branches into one *imitation* branch and m *projection* branches, where $r = r_1, \ldots, r_m$. Imitation replaces the flexible head u, using the substitution $\rho = [u := \lambda_{\vec{x}}(f(h_1\vec{x}) \ldots (h_n\vec{x}))]$ with new variables $\vec{h}$ and $\vec{x}$. This is only allowed if f is a signature (and not a forbidden) variable. For r_i we have a projection if and only if the final value type of r_i is the (ground) type of $f\vec{s}$. Then the i-th projections pulls r_i in front, by $\rho = [u := \lambda_{\vec{x}}(x_i(h_1\vec{x}) \ldots (h_{n_i}\vec{x}))]$. In each of these branches we have

$$Q(u\vec{r} = f\vec{s} \wedge C) \Longrightarrow_\rho Q'(u\vec{r} = f\vec{s} \wedge C)\rho,$$

where Q' is obtained from Q by removing $\exists_u$ and adding $\exists_{\vec{h}}$.

Case flex-flex, i.e., $Q(u\vec{r} = v\vec{s} \wedge C)$. If there is a first flex-rigid or rigid-flex equation in C, pull this equation (possibly swapped) to the front and apply case flex-rigid. Otherwise, i.e., if all equations are between terms with flexible heads, pick a new variable z of ground type and let ρ be the substitution mapping each of these flexible heads u to $\lambda_{\vec{x}}z$.

$$Q(u\vec{r} = v\vec{s} \wedge C) \Longrightarrow_\rho Q\emptyset.$$

This concludes the definition of the unification algorithm.

Clearly ρ is defined on flexible variables of Q only, and its value terms have no free occurrences of forbidden variables from Q. One can prove correctness and completeness of this algorithm.

Theorem (Huet)**.** *Let a unification problem $\mathcal{U}$ consisting of a $\forall\exists\forall$-prefix Q and a list $\vec{r} = \vec{s}$ of unification pairs be given. Then either*

(a) *the unification algorithm can make a transition, and*
   (i) *(correctness) for every transition $\mathcal{U} \Longrightarrow_\rho \mathcal{U}'$ and $\mathcal{U}'$-solution φ' the substitution $(\rho \circ \varphi')\restriction Q_\exists$ is a $\mathcal{U}$-solution, and*
   (ii) *(completeness) for every $\mathcal{U}$-solution φ there is a transition $\mathcal{U} \Longrightarrow_\rho \mathcal{U}'$ and $\mathcal{U}'$-solution φ' such that $\varphi = (\rho \circ \varphi')\restriction Q_\exists$, and moreover $\mu(\varphi') \leq \mu(\varphi)$ with $<$ in case flex-rigid, or else*

(b) *the unification algorithm fails, and there is no $\mathcal{U}$-solution, or else*

(c) *the unification algorithm succeeds, and $\vec{r} = \vec{s}$ is empty.*

Here $\mu(\varphi)$ denotes the number of applications in the value terms of φ.

Corollary. *Given a unification problem $\mathcal{U} = QC$, and a success node in the matching tree, labelled with a prefix Q' (i.e., a unification problem $\mathcal{U}'$ with no unification pairs). Then by composing the substitution labels on the branch leading to this node we obtain a pair (Q', ρ) with a "transition" substitution*

ρ and such that for any Q'-substitution φ', $(\rho \circ \varphi')\restriction Q_\exists$ is an $\mathcal{U}$-solution. Moreover, every $\mathcal{U}$-solution can be obtained in this way, for an appropriate success node. Since the empty substitution is a Q'-substitution, $\rho\restriction Q_\exists$ is a $\mathcal{U}$-solution, which is most general in the sense stated.

11.2. **The pattern unification algorithm.** We restrict the notion of a Q-term as follows. *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.

Explicitely, 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 $D \to G$ is a Q-goal.
- If G is a Q-goal and D is a Q-clause, then $G \to D$ 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.

Explicitely, 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-substitution* is a substitution of Q-terms.

A *pattern 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 now define the pattern unification algorithm. It takes a unification problem $\mathcal{U} = QC$ and returns a substitution ρ and another unification problem $\mathcal{U}' = Q'C'$. Note that ρ will be neither a Q-substitution nor a Q'-substitution, but will have the property that

(a) ρ is defined on flexible variables of Q only, and its value terms have no free occurrences of forbidden variables from Q,
(b) if G is a Q-goal, then $G\rho$ is a Q'-goal, and
(c) whenever φ' is a $\mathcal{U}'$-solution, then $(\rho \circ \varphi')\restriction Q_\exists$ is a $\mathcal{U}$-solution.

Definition (Pattern unification algorithm)**.** We distinguish cases according to the form of the unification problem, and either give the transition done by the algorithm, or else state that it fails.

Case identity, i.e., $Q(r = r \wedge C)$. Then

$$Q(r = r \wedge C) \Longrightarrow_\varepsilon QC.$$

Case ξ, i.e., $Q(\lambda_{\vec{x}} r = \lambda_{\vec{x}} s \wedge C)$. We may assume here that the bound variables $\vec{x}$ are the same on both sides.

$$Q(\lambda_{\vec{x}} r = \lambda_{\vec{x}} s \wedge C) \Longrightarrow_\varepsilon Q(\forall_{\vec{x}}(r = s) \wedge C).$$

Case rigid-rigid, i.e., $Q(f\vec{r} = f\vec{s} \wedge C)$ with f either a signature variable or else a forbidden variable.

$$Q(f\vec{r} = f\vec{s} \wedge C) \Longrightarrow_\varepsilon Q(\vec{r} = \vec{s} \wedge C).$$

Case flex-flex with equal heads, i.e., $Q(u\vec{y} = u\vec{z} \wedge C)$.

$$Q(u\vec{y} = u\vec{z} \wedge C) \Longrightarrow_\rho Q'(C\rho)$$

with $\rho = [u := \lambda_{\vec{y}}(u'\vec{w})]$, Q' is Q with $\exists_u$ replaced by $\exists_{u'}$, and $\vec{w}$ an enumeration of those y_i which are identical to z_i (i.e., the variable at the same position in $\vec{z}$). Notice that $\lambda_{\vec{y}}(u'\vec{w}) = \lambda_{\vec{z}}(u'\vec{w})$.

Case flex-flex with different heads, i.e., $Q(u\vec{y} = v\vec{z} \wedge C)$.

$$Q(u\vec{y} = v\vec{z} \wedge C) \Longrightarrow_\rho Q'C\rho,$$

where ρ and Q' are defined as follows. Let $\vec{w}$ be an enumeration of the variables both in $\vec{y}$ and in $\vec{z}$. Then $\rho = [u, v := \lambda_{\vec{y}}(u'\vec{w}\,), \lambda_{\vec{z}}(u'\vec{w}\,)]$, and Q' is Q with $\exists_u, \exists_v$ removed and $\exists_{u'}$ inserted.

Case flex-rigid, i.e., $Q(u\vec{y} = t \wedge C)$ with t rigid, i.e., not of the form $v\vec{z}$ with flexible v.

Subcase occurrence check: t contains (a critical subterm with head) u. Then fail.

Subcase pruning: t contains a subterm $v\vec{w}_1 z\vec{w}_2$ with $\exists_v$ in Q, and z free in t but not in $\vec{y}$.

$$Q(u\vec{y} = t \wedge C) \Longrightarrow_\rho Q'(u\vec{y} = t\rho \wedge C\rho)$$

where $\rho = [v := \lambda_{\vec{w}_1}\lambda_z\lambda_{\vec{w}_2}(v'\vec{w}_1\vec{w}_2)]$, Q' is Q with $\exists_v$ replaced by $\exists_{v'}$.

Subcase pruning impossible: $\lambda_{\vec{y}} t$ (after all pruning steps are done still) has a free occurrence of a forbidden variable z. Then fail.

Subcase explicit definition: otherwise.

$$Q(u\vec{y} = t \wedge C) \Longrightarrow_\rho Q'C\rho$$

where $\rho = [u := \lambda_{\vec{y}} t]$, and Q' is obtained from Q by removing $\exists_u$. This concludes the definition of the pattern unification algorithm.

One can prove that this algorithm indeed has the three properties stated above. The first one (ρ is defined on flexible variables of Q only, and its value terms have no free occurrences of forbidden variables from Q) is obvious from the definition. We now state the second one; the third one will be stated next.

Lemma (Q'-goals)**.** *If $Q \Longrightarrow_\rho Q'$ and G is a Q-goal, then $G\rho$ is a Q'-goal.*

Let $Q \longrightarrow_\rho Q'$ mean that for some C, C' we have $QC \Longrightarrow_\rho Q'C'$. Write $Q \longrightarrow^*_\rho Q'$ if there are $\rho_1, \dots, \rho_n$ and $Q_1, \dots, Q_{n-1}$ such that

$$Q \longrightarrow_{\rho_1} Q_1 \longrightarrow_{\rho_2} \dots \longrightarrow_{\rho_{n-1}} Q_{n-1} \longrightarrow_{\rho_n} Q',$$

and $\rho = \rho_1 \circ \dots \circ \rho_n$.

Corollary. *If $Q \longrightarrow^*_\rho Q'$ and G is a Q-goal, then $G\rho$ is a Q'-goal.*

Lemma. *Let a unification problem $\mathcal{U}$ consisting of a $\forall\exists\forall$-prefix Q and a list $\vec{r} = \vec{s}$ of unification pairs be given. Then either*

(a) *the unification algorithm makes a transition $\mathcal{U} \Longrightarrow_\rho \mathcal{U}'$, and*

$$\Phi' \colon \mathcal{U}'\text{-solutions} \to \mathcal{U}\text{-solutions}$$
$$\varphi' \mapsto (\rho \circ \varphi')\restriction Q_\exists$$

is well-defined and we have $\Phi \colon \mathcal{U}$-solutions $\to \mathcal{U}'$-solutions such that Φ' is inverse to Φ, i.e. $\Phi'(\Phi\varphi) = \varphi$, or else

(b) *the unification algorithm fails, and there is no $\mathcal{U}$-solution.*

It is not hard to see that the unification algorithm terminates, by defining a measure that decreases with each transition.

Corollary. *Given a unification problem $\mathcal{U} = QC$, the unification algorithm either fails, and there is no $\mathcal{U}$-solution, or else returns a pair (Q', ρ) with a "transition" substitution ρ and a prefix Q' (i.e., a unification problem $\mathcal{U}'$ with no unification pairs) such that for any Q'-substitution φ', $(\rho \circ \varphi')\restriction Q_\exists$ is an $\mathcal{U}$-solution, and every $\mathcal{U}$-solution can be obtained in this way. Since the empty substitution is a Q'-substitution, $\rho\restriction Q_\exists$ is a $\mathcal{U}$-solution, which is most general in the sense stated.*

11.3. **Proof search.** 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.

We write $M[\mathcal{P}]$ to indicate that all assumption variables in the derivation M are assumptions of clauses in $\mathcal{P}$.

Write $\vdash^n S$ for a set S of sequents if there are derivations $M_i^{G_i}[\mathcal{P}_i]$ in long normal form for all $(\mathcal{P}_i \Rightarrow G_i) \in S$ such that $\sum \mathrm{dp}(M_i) \le n$. Let $\vdash^{<n} S$ mean $\exists_{m<n} \vdash^m S$.

We prove correctness and completeness of the proof search procedure: correctness is the if-part of the two lemmata to follow, and completeness the only-if-part.

Lemma. *Let Q be a $\forall\exists\forall$-prefix, $\{\mathcal{P} \Rightarrow \forall_{\vec{x}}(\vec{D} \to A)\} \cup S$ Q-sequents with $\vec{x}, \vec{D}$ not both empty. Then we have for every substitution φ:*

$$\varphi \text{ is a } Q\text{-substitution such that } \vdash^n \big(\{\mathcal{P} \Rightarrow \forall_{\vec{x}}(\vec{D} \to A)\} \cup S\big)\varphi$$

if and only if

$$\varphi \text{ is a } Q\forall_{\vec{x}}\text{-substitution such that } \vdash^{<n} \big(\{\mathcal{P} \cup \vec{D} \Rightarrow A\} \cup S\big)\varphi.$$

Proof. "If". Let φ be a $Q\forall_{\vec{x}}$-substitution and $\vdash^{<n} \big(\{\mathcal{P} \cup \vec{D} \Rightarrow A\} \cup S\big)\varphi$. So we have

$$N^{A\varphi}[\vec{D}\varphi \cup \mathcal{P}\varphi].$$

Since φ is a $Q\forall_{\vec{x}}$-substitution, no variable in $\vec{x}$ can be free in $\mathcal{P}\varphi$, or free in $y\varphi$ for some $y \in \mathrm{dom}(\varphi)$. Hence

$$M^{(\forall_{\vec{x}}(\vec{D}\to A))\varphi}[\mathcal{P}\varphi] := \lambda_{\vec{x}}\lambda_{\vec{u}^{\vec{D}\varphi}} N$$

is a correct derivation.

"Only if". Let φ be a Q-substitution and $\vdash^n \big(\{\mathcal{P} \Rightarrow \forall_{\vec{x}}(\vec{D} \to A)\} \cup S\big)\varphi$. This means we have a derivation (in long normal form)

$$M^{(\forall_{\vec{x}}(\vec{D}\to A))\varphi}[\mathcal{P}\varphi] = \lambda_{\vec{x}}\lambda_{\vec{u}^{\vec{D}\varphi}}(N^{A\varphi}[\vec{D}\varphi \cup \mathcal{P}\varphi]).$$

Now $\mathrm{dp}(N) < \mathrm{dp}(M)$, hence $\vdash^{<n} \big(\{\mathcal{P} \cup \vec{D} \Rightarrow A\} \cup S\big)\varphi$, and φ clearly is a $Q\forall_{\vec{x}}$-substitution. □

Lemma. *Let Q be a $\forall\exists\forall$-prefix, $\{\mathcal{P} \Rightarrow P\vec{r}\} \cup S$ Q-sequents and φ a substitution. Then*

$$\varphi \text{ is a } Q\text{-substitution such that } \vdash^n \big(\{\mathcal{P} \Rightarrow P\vec{r}\} \cup S\big)\varphi$$

if and only if there is a clause $\forall_{\vec{x}}(\vec{G} \to P\vec{s})$ in $\mathcal{P}$ such that the following holds. Let $\vec{z}$ be the final universal variables in Q, $\vec{X}$ be new ("raised") variables such that $X_i\vec{z}$ has the same type as x_i, let Q^* be Q with the existential variables extended by $\vec{X}$, and let $*$ indicate the substitution $[x_1, \dots, x_n := X_1\vec{z}, \dots, X_n\vec{z}]$. Then there is a result (Q', ρ) of either Huet's or the pattern

unification algorithm applied to $Q^*(\vec{r} = \vec{s}^*)$ *and a* Q'*-substitution* φ' *such that* $\vdash^{<n} \big(\{\mathcal{P} \Rightarrow \vec{G}^*\} \cup S\big)\rho\varphi'$*, and* $\varphi = (\rho \circ \varphi')\restriction Q_\exists$.

Proof. "If". Let (Q', ρ) be such a result, and assume that φ' is a Q'-substitution such that $N_i \vdash \big(\mathcal{P} \Rightarrow \vec{G}^*\big)\rho\varphi'$. Let $\varphi := (\rho \circ \varphi')\restriction Q_\exists$. From $\mathrm{unif}(Q^*, \vec{r} = \vec{s}^*) = (Q', \rho)$ we know $\vec{r}\rho = \vec{s}^*\rho$, hence $\vec{r}\varphi = \vec{s}^*\rho\varphi'$. Then

$$u^{(\forall_{\vec{x}}.\vec{G} \to P\vec{s})\varphi}((\vec{X}\rho\varphi')\vec{z})\vec{N}^{\vec{G}^*\rho\varphi'}$$

derives $P\vec{s}^*\rho\varphi'$ (i.e., $P\vec{r}\varphi$) from $\mathcal{P}\varphi$.

"Only if". Assume φ is a Q-substitution such that $\vdash (\mathcal{P} \Rightarrow P\vec{r})\varphi$, say by $u^{\forall_{\vec{x}}(\vec{G} \to P\vec{s})\varphi}\vec{t}\vec{N}^{(\vec{G}\varphi)[\vec{x}:=\vec{t}]}$, with $\forall_{\vec{x}}(\vec{G} \to P\vec{s})$ a clause in $\mathcal{P}$, and with additional assumptions from $\mathcal{P}\varphi$ in $\vec{N}$. Then $\vec{r}\varphi = (\vec{s}\varphi)[\vec{x} := \vec{t}]$. Since we can assume that the variables $\vec{x}$ are new and in particular not range variables of φ, with

$$\vartheta := \varphi \cup [\vec{x} := \vec{t}]$$

we have $\vec{r}\varphi = \vec{s}\vartheta$. Let $\vec{z}$ be the final universal variables in Q, $\vec{X}$ be new ("raised") variables such that $X_i\vec{z}$ has the same type as x_i, let Q^* be Q with the existential variables extended by $\vec{X}$, and for terms and formulas let $*$ indicate the substitution $[x_1, \dots, x_n := X_1\vec{z}, \dots, X_n\vec{z}]$. Moreover, let

$$\vartheta^* := \varphi \cup [X_1, \dots, X_n := \lambda_{\vec{z}}t_1, \dots, \lambda_{\vec{z}}t_n].$$

Then $\vec{r}\vartheta^* = \vec{r}\varphi = \vec{s}\vartheta = \vec{s}^*\vartheta^*$, i.e., ϑ^* is a solution to the unification problem given by Q^* and $\vec{r} = \vec{s}$. Hence by the corollary $\mathrm{unif}(Q^*, \vec{r} = \vec{s}^*) = (Q', \rho)$ and there is a Q'-substitution φ' such that $\vartheta^* = (\rho\circ\varphi')\restriction Q^*_\exists$, hence $\varphi = (\rho\circ\varphi')\restriction Q_\exists$. Also, $(\vec{G}\varphi)[\vec{x} := \vec{t}] = \vec{G}\vartheta = \vec{G}^*\vartheta^* = \vec{G}^*\rho\varphi'$. □

A *state* is a pair (Q, S) with Q a prefix and S a finite set of Q-sequents. By the two lemmas just proved we have *state transitions*

$$(Q, \{\mathcal{P} \Rightarrow \forall_{\vec{x}}(\vec{D} \to A)\} \cup S) \mapsto^{\varepsilon} (Q\forall_{\vec{x}}, \{\mathcal{P} \cup \vec{D} \Rightarrow A\} \cup S)$$
$$(Q, \{\mathcal{P} \Rightarrow P\vec{r}\} \cup S) \mapsto^{\rho} (Q', (\{\mathcal{P} \Rightarrow \vec{G}^*\} \cup S)\rho),$$

where in the latter case there is a clause $\forall_{\vec{x}}(\vec{G} \to P\vec{s})$ in $\mathcal{P}$ such that the following holds. Let $\vec{z}$ be the final universal variables in Q, $\vec{X}$ be new ("raised") variables such that $X_i\vec{z}$ has the same type as x_i, let Q^* be Q with the existential variables extended by $\vec{X}$, and let $*$ indicate the substitution $[x_1, \dots, x_n := X_1\vec{z}, \dots, X_n\vec{z}]$, and $\mathrm{unif}(Q^*, \vec{r} = \vec{s}^*) = (Q', \rho)$.

Notice that by the lemma on Q'-goals above, if $\mathcal{P} \Rightarrow P\vec{r}$ is a *Q-sequent* (which means that $\bigwedge\mathcal{P} \to P\vec{r}$ is a Q-goal), then $(\mathcal{P} \Rightarrow \vec{G}^*)\rho$ is a Q'-sequent.

Theorem. *Let Q be a prefix, and S be a set of Q-sequents. For every substitution φ we have: φ is a Q-substitution satisfying $\vdash S\varphi$ iff there is a*

prefix Q', a substitution ρ and a Q'-substitution φ' such that

$$(Q, S) \mapsto^{\rho *} (Q', \emptyset),$$
$$\varphi = (\rho \circ \varphi')\restriction Q_{\exists}.$$

Examples. (a) The sequent $\forall_y(\forall_z Ryz \to Q), \forall_{y_1,y_2} Ry_1y_2 \Rightarrow Q$ leads first to $\forall_{y_1,y_2} Ry_1y_2 \Rightarrow Ryz$ under $\exists_y\forall_z$, then to $y_1 = y \wedge y_2 = z$ under $\exists_y\forall_z\exists_{y_1,y_2}$, and finally to $Y_1z = y \wedge Y_2z = z$ under $\exists_{y,Y_1,Y_2}\forall_z$, which has the solution $Y_1 = \lambda_z y$, $Y_2 = \lambda_z z$.

(b) $\forall_y(\forall_z Ryz \to Q), \forall_{y_1} Ry_1y_1 \Rightarrow Q$ leads first to $\forall_{y_1} Ry_1y_1 \Rightarrow Ryz$ under $\exists_y\forall_z$, then to $y_1 = y \wedge y_1 = z$ under $\exists_y\forall_z\exists_{y_1}$, and finally to $Y_1z = y \wedge Y_1z = z$ under $\exists_{y,Y_1}\forall_z$, which has no solution.

(c) Here is a more complex example (derived from proofs of the Orevkov-formulas), for which we only give the derivation tree.

$$\dfrac{\dfrac{\dfrac{\forall_y(\forall_z(Ryz\to\bot))\to\bot}{\forall_z(R0z\to\bot)\to\bot} \qquad \dfrac{\dfrac{\dfrac{\forall_z(S0z\to\bot)}{S0z_1\to\bot} \quad \dfrac{(*) \quad R0z \quad Rzz_1}{S0z_1}}{\bot}}{R0z\to\bot}}{\forall_z(R0z\to\bot)} \qquad \dfrac{\dfrac{\forall_y(\forall_{z_1}(Ryz_1\to\bot)\to\bot)}{\forall_{z_1}(Rzz_1\to\bot)\to\bot} \qquad \dfrac{\dfrac{\bot}{Rzz_1\to\bot}}{\forall_{z_1}(Rzz_1\to\bot)}}{\bot}}{\bot}$$

where $(*)$ is a derivation from $\mathrm{Hyp}_1 \colon \forall_{z,z_1}(R0z \to Rzz_1 \to S0z_1)$.

11.4. **Extension by $\wedge$ and $\exists$.** The extension by conjunction is rather easy; it is even superfluous in principle, since conjunctions can always be avoided at the expense of having lists of formulas instead of single formulas.

However, having conjunctions available is clearly useful at times, so let's add it. This requires the notion of an *elaboration path* for a formula (cf. [18]). The reason is that the property of a formula to have a unique atom as its *head* is lost when conjunctions are present. An elaboration path is meant to give the directions (left or right) to go when we encounter a conjunction as a strictly positive subformula. For example, the elaboration paths of $\forall_x A \wedge (B \wedge C \to D \wedge \forall_y E)$ are (`left`), (`right`, `left`) and (`right`, `right`). Clearly, a formula is equivalent to the conjunction (over all elaboration paths) of all formulas obtained from it by following an elaboration path (i.e., always throwing away the other part of the conjunction). In our example,

$$\forall_x A \wedge (B \wedge C \to D \wedge \forall_y E) \leftrightarrow \forall_x A \wedge (B \wedge C \to D) \wedge (B \wedge C \to \forall_y E).$$

In this way we regain the property of a formula to have a unique head, and our previous search procedure continues to work.

For the existential quantifier $\exists$ the problem is of a different nature. We chose to introduce $\exists$ by means of axiom schemata. Then the problem is which of such schemes to use in proof search, given a goal G and a set $\mathcal{P}$ of clauses. We might proceed as follows.

List all prime, positive and negative existential subformulas of $\mathcal{P} \Rightarrow G$, and remove any formula from those lists which is of the form of another one[1]. For every positive existential formula – say $\exists_x B$ – add (the generalization of) the existence introduction scheme

$$\exists^+_{x,B} \colon \forall_x(B \to \exists_x B)$$

to $\mathcal{P}$. Moreover, for every negative existential formula – say $\exists_x A$ – and every (prime or existential) formula C in any of those two lists, except the formula $\exists_x A$ itself, add (the generalization of) the existence elimination scheme

$$\exists^-_{x,A,B} \colon \exists_x A \to \forall_x(A \to B) \to B$$

to $\mathcal{P}$. Then start the search algorithm as described in section 11.3. The normal form theorem for the natural deduction system of minimal logic with $\exists$ then guarantees completeness.

However, experience has shown that this complete search procedure tends to be trapped in too large a search space. Therefore in our actual implementation we decided to only take instances of the existence elimination scheme with *existential* conclusions.

Moreover, it seems appropriate that – before the search is started – one eliminates in a preprocessing step as many existential quantifiers as possible.

11.5. **Implementation.** Following Miller [18], Berger and [22], 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 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 $\texttt{star} \circ \rho$, where `star` is the "raising" substitution and ρ is the most general unificator. For this constellation `intro-search` is called again. In case of success, one obtains a Q'-substitution φ' and proofs of the $\texttt{star} \circ \rho \circ \varphi'$ -substituted new sequents. Let $\varphi := (\rho \circ \varphi')\restriction Q_\exists$, and take the

[1] To do this, for patterns the dual of the theory of "most general unifiers", i.e., a theory of "most special generalizations", needs to be developed.

first n proofs of these to build a proof of the φ-substituted (first) sequent originally considered by `intro-search`.

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

(i) numbers of hypotheses from the present context or
(ii) names for theorems or global assumptions,

and for *m1* ... 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.

11.6. **Notes.** I have benefitted from a presentation of Miller's [18] given by Ulrich Berger, in a logic seminar in München in 1991. The type of restriction to higher order terms described in the text has been introduced in [18]; it has been called *patterns* by Nipkow [19]. Miller also noted its relevance for extensions of logic programming, and showed that the unification problem for patterns is solvable and admits most general unifiers. The present treatment was motivated by the desire to use Miller's approach as a basis for an implementation of a simple proof search engine for (first and higher order) minimal logic.

Compared with Miller [18], we make use of several simplifications, optimizations and extensions, in particular the following.

(i) Instead of arbitrarily mixed prefixes we only use those of the form $\forall\exists\forall$. Nipkow in [19] already had presented a version of Miller's pattern unification algorithm for such prefixes, and Miller in [18, 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.
(ii) 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 [18, p.515] that such optimizations are possible.
(iii) The extensions concern the (strong) existential quantifier, which has been left out in Miller's treatment, and also conjunction(cf. 11.4). The latter can be avoided in principle, but of course is a useful thing to have.

12. 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 assigment 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 $\tau(A) \neq$ `nulltype`, its *extracted term* (or *extracted program*) $\mathrm{et}(M)$ 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	$\lambda_x M$	`([x]M)`
pair	$\langle M, N\rangle$	`(M@N)`
left element of a pair	$(M\ 0)$	`left M`
right element of a pair	$(M\ 1)$	`right M`
arrow for types	$\to$	`=>`
product for types	$\times$	`@@`
recursion operator	$\mathcal{R}$	`Rec`

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

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

13. Computational content of classical proofs

13.1. **Refined A-translation.** In this section the connectives $\to$, $\forall$ denote the computational versions $\to^{\mathrm{c}}$, $\forall^{\mathrm{c}}$, unless stated otherwise.

We will concentrate on the question of classical versus constructive proofs. It is known, by the so-called "A-translation" of Friedman [10] and Dragalin [8], that any proof of a specification of the form $\forall_x \tilde{\exists}_y B$ with B quantifier-free and a weak (or "classical") existential quantifier $\tilde{\exists}_y$, can be transformed into a proof of $\forall_x \exists_y B$, now with the constructive existential quantifier $\exists_y$. However, when it comes to extraction of a program from a proof obtained in this way, one easily ends up with a mess. Therefore, some refinements of the standard transformation are necessary. We shall study a refined method of extracting reasonable and sometimes unexpected programs from classical proofs. It applies to proofs of formulas of the form $\forall_x \tilde{\exists}_y B$ where B need not be quantifier-free, but only has to belong to the larger class of *goal formulas*. Furthermore we allow unproven lemmata D to appear in the proof of $\forall_x \tilde{\exists}_y B$, where D is a *definite* formula.

We now describe in more detail what this section is about. It is well known that from a derivation of a classical existential formula $\tilde{\exists}_y A := \forall_y (A \to \bot) \to \bot$ one generally cannot read off an instance. A simple example has been given by Kreisel: let R be a primitive recursive relation such that $\tilde{\exists}_z Rxz$ is undecidable. Clearly – even logically –

$$\vdash \forall_x \tilde{\exists}_y \forall_z (Rxz \to Rxy)$$

but there is no computable f satisfying

$$\forall_x \forall_z (Rxz \to R(x, f(x))),$$

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

However, it is well known that in case $\tilde{\exists}_y G$ with G quantifier-free one *can* read off an instance. Here is a simple idea of how to prove this: replace $\bot$ anywhere in the proof by $\exists_y G$. Then the end formula $\forall_y(G \to \bot) \to \bot$ is turned into $\forall_y(G \to \exists_y G) \to \exists_y G$, and since the premise is trivially provable, we have the claim.

Unfortunately, this simple argument is not quite correct. First, G may contain $\bot$, and hence is changed under the substitution of $\exists_y G$ for $\bot$. Second, we may have used axioms or lemmata involving $\bot$ (e.g., $\bot \to P$), which need not be derivable after the substitution. But in spite of this, the simple idea can be turned into something useful.

Assume that the lemmata $\vec{D}$ and the goal formula G are such that we can derive

$$\vec{D} \to D_i[\bot := \exists_y G], \tag{7}$$

$$G[\bot := \exists_y G] \to \exists_y G. \tag{8}$$

Assume also that the substitution $[\bot := \exists_y G]$ turns any axiom into an instance of the same axiom-schema, or else into a derivable formula. Then from our given derivation (in minimal logic) of $\vec{D} \to \forall_y(G \to \bot) \to \bot$ we obtain

$$\vec{D}[\bot := \exists_y G] \to \forall_y(G[\bot := \exists_y G] \to \exists_y G) \to \exists_y G.$$

Now (7) allows the substitution in $\vec{D}$ to be dropped, and by (8) the second premise is derivable. Hence we obtain as desired

$$\vec{D} \to \exists_y G.$$

We shall identify classes of formulas – to be called *definite* and *goal* formulas – such that slight generalizations of (7) and (8) hold.

This section is based on [3] and particularly [23, 7.3], where the theory is developed in more detail and further references are given. Recall that we restrict to formulas in the language $\{\bot, \to, \forall\}$.

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

```
(atr-relevant? formula)
(atr-definite? formula)
(atr-goal? formula)
```

We need to construct proofs from $\mathbf{F} \to \bot$ of

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

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

$$((R^{\mathbf{F}} \to \mathbf{F}) \to \bot) \to R \quad \text{for } R \text{ relevant and definite,}$$

$I \to I^{\mathbf{F}}$ for I irrelevant and goal.

This is done by

```
(atr-arb-definite-proof formula)
(atr-arb-goal-proof formula)
(atr-rel-definite-proof formula)
(atr-irrel-goal-proof formula)
```

The next task is to generalize $G \to (G^{\mathbf{F}} \to \bot) \to \bot$ and construct a proof of $(G_1^{\mathbf{F}} \to ... \to G_n^{\mathbf{F}} \to \bot) \to G_1 \to ... \to G_n \to \bot$, via

```
(atr-goals-F-to-bot-proof . goals)
```

Given a proof of $\vec{A} \to \vec{D} \to \forall_{\vec{y}}(\vec{G} \to \bot) \to \bot$ with $\vec{A}$ arbitrary, $\vec{D}$ definite and $\vec{G}$ goal formulas, we transform it into a proof of $(\mathbf{F} \to \bot) \to \vec{A} \to \vec{D}^{\mathbf{F}} \to \forall_{\vec{y}}(\vec{G}^{\mathbf{F}} \to \bot) \to \bot$. This is done via

```
(atr-min-excl-proof-to-bot-reduced-proof min-excl-proof)
```

Substituting the formula $\exists_{\vec{y}}\vec{G}^{\mathbf{F}}$ for $\bot$ in the proof given above of $(F \to \bot) \to \vec{A} \to \vec{D}^{\mathbf{F}} \to \forall_{\vec{y}}(\vec{G}^{\mathbf{F}} \to \bot) \to \bot$, both the ex-falso-quodlibet premise and the "wrong formula" $\forall_{\vec{y}}(\vec{G}^{\mathbf{F}} \to \bot)$ become provable and we obtain a proof of $\vec{A}' \to \vec{D}^{\mathbf{F}} \to \exists_{\vec{y}}\vec{G}^{\mathbf{F}}$, where $\vec{A}'$ is defined to be $\vec{A}[\bot := \exists_{\vec{y}}\vec{G}^{\mathbf{F}}]$. The corresponding function is

```
(atr-min-excl-proof-to-ex-proof min-excl-proof)
```

One can test with `min-excl-proof?` whether a given proof indeed is a proof in minimal logic of a classical (i.e., weak) existence formula. Moreover, `atr-expand-theorems` expands all non-definite theorems. This only makes sense before substituting for $\bot$.

See section 12 for an interpretation of the symbols of the extracted terms in Minlog's output.

13.2. **Gödel's Dialectica interpretation.** In his original functional interpretation [11], Gödel assigned to every formula A a new one $\exists_{\vec{x}}\forall_{\vec{y}}A_D(\vec{x},\vec{y})$ with $A_D(\vec{x},\vec{y})$ quantifier-free. Here $\vec{x}$, $\vec{y}$ are lists of variables of finite types; the use of higher types is necessary even when the original formula A is first-order. He did this in such a way that whenever a proof of A say in Peano arithmetic was given, one could produce closed terms $\vec{r}$ such that the quantifier-free formula $A_D(\vec{r},\vec{y})$ is provable in his quantifier-free system T.

In [11] Gödel referred to a Hilbert-style proof calculus. However, since the realizers will be formed in a λ-calculus formulation of system T, Gödel's interpretation becomes more perspicuous when it is done for a natural deduction calculus. The present implementation is based on such a setup.

Then the need for contractions comes up in the (only) logical rule with two premises: modus ponens (or implication elimination $\to^-$). This makes it possible to give a relatively simple proof of the Soundness Theorem.

We assign to every formula A objects $\tau^+(A)$, $\tau^-(A)$ (a type or the "null-type" symbol $\circ$). $\tau^+(A)$ is intended to be the type of a (Dialectica-) realizer to be extracted from a proof of A, and $\tau^-(A)$ the type of a challenge for the claim that this term realizes A.

$$\begin{aligned}
\tau^+(P\vec{s}) &:= \circ, & \tau^-(P\vec{s}) &:= \circ,\\
\tau^+(\forall_{x^\rho} A) &:= \rho \to \tau^+(A), & \tau^-(\forall_{x^\rho} A) &:= \rho \times \tau^-(A),\\
\tau^+(\exists_{x^\rho} A) &:= \rho \times \tau^+(A), & \tau^-(\exists_{x^\rho} A) &:= \tau^-(A),\\
\tau^+(A \wedge B) &:= \tau^+(A) \times \tau^+(B), & \tau^-(A \wedge B) &:= \tau^-(A) \times \tau^-(B),
\end{aligned}$$

and for implication

$$\begin{aligned}
\tau^+(A \to B) &:= (\tau^+(A) \to \tau^+(B)) \times (\tau^+(A) \to \tau^-(B) \to \tau^-(A)),\\
\tau^-(A \to B) &:= \tau^+(A) \times \tau^-(B).
\end{aligned}$$

Recall that $(\rho \to \circ) := \circ$, $(\circ \to \sigma) := \sigma$, $(\circ \to \circ) := \circ$, and $(\rho \times \circ) := \rho$, $(\circ \times \sigma) := \sigma$, $(\circ \times \circ) := \circ$.

In case $\tau^+(A)$ ($\tau^-(A)$) is $\neq \circ$ we say that A has *positive (negative) computational content*. For formulas without positive or without negative content one can give an easy characterization, involving the well-known notion of positive or negative occurrences of quantifiers in a formula.

$$\begin{aligned}
&\tau^+(A) = \circ \leftrightarrow A \text{ has no positive } \exists \text{ and no negative } \forall,\\
&\tau^-(A) = \circ \leftrightarrow A \text{ has no positive } \forall \text{ and no negative } \exists,\\
&\tau^+(A) = \tau^-(A) = \circ \leftrightarrow A \text{ is quantifier-free.}
\end{aligned}$$

Both the positive and the negative type of a formula can be computed by

```
(formula-to-etdp-type formula)
(formula-to-etdn-type formula)
```

For every formula A and terms r of type $\tau^+(A)$ and s of type $\tau^-(A)$ we define a new quantifier-free formula $|A|^r_s$ by induction on A.

$$\begin{aligned}
|P\vec{s}|^r_s &:= P\vec{s},\\
|\forall_x A(x)|^r_s &:= |A(s0)|^{r(s0)}_{s1},\\
|\exists_x A(x)|^r_s &:= |A(r0)|^{r1}_s,
\end{aligned}
\qquad
\begin{aligned}
|A \wedge B|^r_s &:= |A|^{r0}_{s0} \wedge |B|^{r1}_{s1},\\
|A \to B|^r_s &:= |A|^{s0}_{r1(s0)(s1)} \to |B|^{r0(s0)}_{s1}.
\end{aligned}$$

The formula $\exists_x \forall_y |A|^x_y$ is called the *Gödel translation* of A and is often denoted by A^D. Its quantifier-free kernel $|A|^x_y$ is called *Gödel kernel* of A; it is denoted by A_D.

For readability we sometimes write terms of a pair type in pair form:

$$|\forall_z A|^f_{z,y} := |A|^{fz}_y, \qquad |A \wedge B|^{x,z}_{y,u} := |A|^x_y \wedge |B|^z_u,$$
$$|\exists_z A|^{z,x}_y := |A|^x_y, \qquad |A \to B|^{f,g}_{x,u} := |A|^x_{gxu} \to |B|^{fx}_u.$$

`formula-to-d-formula` calculates the Gödel (or Dialectica) translation of a formula.

To answer the question when the Gödel translation of a formula A is equivalent to the formula itself, we need the (constructively doubtful) *Markov principle* (MP), for higher type variables and quantifier-free formulas A_0, B_0.

$$(\forall_{x^\rho} A_0 \to B_0) \to \exists_{x^\rho}(A_0 \to B_0) \quad (x^\rho \notin \mathrm{FV}(B_0)).$$

We also need the (less problematic) *axiom of choice* (AC)

$$\forall_{x^\rho}\exists_{y^\sigma} A(x,y) \to \exists_{f^{\rho\to\sigma}}\forall_{x^\rho} A(x, f(x)).$$

and the *independence of premise* axiom (IP)

$$(A \to \exists_{x^\rho} B) \to \exists_{x^\rho}(A \to B) \quad (x^\rho \notin \mathrm{FV}(A),\ \tau^+(A) = \circ).$$

Notice that (AC) expresses that we can only have continuous dependencies.

Theorem (Characterization)**.**

$$\mathrm{AC} + \mathrm{IP} + \mathrm{MP} \vdash (A \leftrightarrow \exists_x \forall_y |A|^x_y).$$

Let *Heyting arithmetic* HA^ω in all finite types be the fragment of TCF where (i) the only base types are $\mathbf{N}$ and $\mathbf{B}$, and (ii) the only inductively defined predicates are totality, Leibniz equality Eq, the (proper) existential quantifier and conjunction. We can prove soundness of the Dialectica interpretation for $\mathrm{HA}^\omega + \mathrm{AC} + \mathrm{IP} + \mathrm{MP}$, for our natural deduction formulation of the underlying logic.

Theorem (Soundness)**.** *Let M be a derivation*

$$\mathrm{HA}^\omega + \mathrm{AC} + \mathrm{IP} + \mathrm{MP} \vdash A$$

from assumptions $u_i\colon C_i$ $(i = 1, \dots, n)$. Let x_i of type $\tau^+(C_i)$ be variables for realizers of the assumptions, and y be a variable of type $\tau^-(A)$ for a challenge of the goal. Then we can find terms $\mathrm{et}^+(M) =: t$ of type $\tau^+(A)$ with $y \notin \mathrm{FV}(t)$ and $\mathrm{et}^-_i(M) =: r_i$ of type $\tau^-(C_i)$, and a derivation in HA^ω of $|A|^t_y$ from assumptions $\bar{u}_i\colon |C_i|^{x_i}_{r_i}$.

`proof-to-extracted-d-terms` returns the extracted realiser and a list of extracted challenges labelled with their associated assumption variables.

14. Reading formulas in external form

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 brocken into short sequences, called *tokens*.

A token can be one of the following:

(i) An alphabetic symbol: A sequence of letters `a`–`z` and `A`–`Z`. Upper and lower case letters are considered different.
(ii) A number: A sequence of digits `0`–`9`
(iii) A punctuation mark: One of the characters: `( ) [ ] . , ;`
(iv) 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 form 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 σ`;`τ is a type (pair type) and σ`=>`τ 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)=>boole` is again a type. The parentheses in this case are not strictly necessary, since `;` binds stronger than `=>`. 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:

(i) `p` a total variable of type boole with name p and no index.
(ii) `p_` a total variable of type boole with name p and no index.
(iii) `p^` a partial variable of type boole with name p and no index.
(iv) `p2` a total variable of type boole with name p and index 2.
(v) `p_2` a total variable of type boole with name p and index 2.
(vi) `p^2` a partial variable of type boole with name p and index 2.
(vii) `boole` a total anonymous variable of type boole with no index.
(viii) `boole_` a total anonymous variable of type boole with no index.
(ix) `boole^` a partial anonymous variable of type boole with no index.
(x) `boole_2` a total anonymous variable of type boole with index 2.
(xi) `boole2` a total anonymous variable of type boole with index 2.
(xii) `boole^2` a partial anonymous variable of type boole with index 2.
(xiii) `(boole)_2` a total anonymous variable of type boole with index 2.
(xiv) `nat=>boole_2` a total anonymous variable of type function of nat to boole with index 2.
(xv) `nat=>boole^2` a partial anonymous variable of type function of nat to boole with index 2.
(xvi) `(nat=>alpha2)` a total anonymous variable of type function of nat to alpha2 with no index.
(xvii) `(nat=>alpha2)_2` a total anonymous variable of type function of nat to alpha2 with index 2.
(xviii) `(nat=>alpha2)^2` a partial anonymous variable of type function of nat to alpha2 with index 2.

Compare these with the following applicative terms.

(i) `nat=>boole 2` a total anonymous variable of type function of nat to boole with no index applied to the numeric term 2.
(ii) `nat=>boole_ 2` a total anonymous variable of type function of nat to boole with no index applied to the numeric term 2.
(iii) `nat=>boole^ 2` a partial anonymous variable of type function of nat to boole with no index applied to the numeric term 2.
(iv) `nat=>boole_2 2` a total anonymous variable of type function of nat to boole with index 2 applied to the numeric term 2.
(v) `nat=>boole^2 2` a partial anonymous variable of type function of nat to boole with index 2 applied to the numeric term 2.
(vi) `(nat=>alpha)2` a total anonymous variable of type function of nat to alpha with no index applied to the numeric term 2.

(vii) `(nat=>alpha)_ 2` a total anonymous variable of type function of nat to alpha with no index applied to the numeric term 2.
(viii) `(nat=>alpha)^ 2` a partial anonymous variable of type function of nat to alpha with no index applied to the numeric term 2.
(ix) `(nat=>alpha)_2 2` a total anonymous variable of type function of nat to alpha with index 2 applied to the numeric term 2.
(x) `(nat=>alpha)^2 2` a partial anonymous variable of type function of nat to alpha with index 2 applied to the numeric term 2.
(xi) `(nat=>alpha2)_2 2` a total anonymous variable of type function of nat to alpha2 with index 2 applied to the numeric term 2.
(xii) `(nat=>alpha2)^2 2` a partial anonymous variable of type function of nat to alpha2 with index 2 applied to the numeric term 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:

(i) multiplicative operators, leftassociative, token type `mul-op`;
(ii) additive operators, leftassociative, token type `add-op`;
(iii) relational operators, not associative, token type `rel-op`;
(iv) boolean and operators, leftassociative, token type `and-op`;
(v) boolean or operators, leftassociative, token type `or-op`;
(vi) boolean implication operators, rightassociative, token type `imp-op`;
(vii) 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:

(i) conjunction junctors, leftassociative, token type `and-jct`;
(ii) disjunction junctors, leftassociative, token type `or-jct`;
(iii) tensor junctors, rightassociative, token type `tensor-jct`;
(iv) 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] Pater Aczel, Harold Simmons, and Stanley S. Wainer (eds.), *Proof theory. a selection of papers from the leeds proof theory programme 1990*, Cambridge University Press, 1992. 21

[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. iii

[3] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg, *Refined program extraction from classical proofs*, Annals of Pure and Applied Logic **114** (2002), 3–25. iii, 13.1

[4] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg, *Term rewriting for normalization by evaluation*, Information and Computation **183** (2003), 19–42. 1.1, 2.3, 4.1, 4.1, 6.2

[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. 6.2

[6] Stefan Berghofer, *Proofs, programs and executable specifications in higher order logic*, Ph.D. thesis, Institut für Informatik, TU München, 2003. 1.6

[7] Coq Development Team, *The Coq Proof Assistant Reference Manual – Version 8.2*, Inria, 2009. 1.6

[8] Albert Dragalin, *New kinds of realizability*, Abstracts of the 6th International Congress of Logic, Methodology and Philosophy of Sciences (Hannover, Germany), 1979, pp. 20–24. 1, 13.1

[9] Yuri L. Ershov, *Model C of partial continuous functionals*, Logic Colloquium 1976 (R. Gandy and M. Hyland, eds.), North–Holland, Amsterdam, 1977, pp. 455–467. 1

[10] Harvey Friedman, *Classically and intuitionistically provably recursive functions*, Higher Set Theory (D.S. Scott and G.H. Müller, eds.), Lecture Notes in Mathematics, vol. 669, Springer Verlag, Berlin, Heidelberg, New York, 1978, pp. 21–28. 1, 13.1

[11] Kurt Gödel, *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts*, Dialectica **12** (1958), 280–287. 1, 4.2, 5.1, 13.2

[12] Gérard Huet, *A unification algorithm for typed λ-calculus*, Theoretical Computer Science **1** (1975), 27–57. 11, 11.1

[13] 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. 6.1

[14] Alberto Martelli and Ugo Montanari, *An efficient unification algorithm*, ACM Transactions on Programming Languages and Systems **4** (1982), no. 2, 258–282. 2.2

[15] Per Martin-Löf, *Hauptsatz for the intuitionistic theory of iterated inductive definitions*, Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, ed.), North–Holland, Amsterdam, 1971, pp. 179–216. 5.3

[16] ———, *Intuitionistic type theory*, Bibliopolis, 1984. 1

[17] 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

[18] 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. v, 11.4, 11.5, 11.6, i, ii

[19] 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. 11.6, i

[20] Diana Ratiu and Helmut Schwichtenberg, *Decorating proofs*, Proofs, Categories and Computations. Essays in honor of Grigori Mints (S. Feferman and W. Sieg, eds.), College Publications, 2010, pp. 171–188. 9.2

[21] Helmut Schwichtenberg, *Proofs as programs*, in Aczel et al. [1], pp. 81–113. (document)

[22] ———, *Proof search in minimal logic*, Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 2004, Proceedings (B. Buchberger and J.A. Campbell, eds.), LNAI, vol. 3249, Springer Verlag, Berlin, Heidelberg, New York, 2004, pp. 15–25. 11.5

[23] Helmut Schwichtenberg and Stanley S. Wainer, *Proofs and computations*, Perspectives in Mathematical Logic, Cambridge University Press, to appear 2010. 13.1

[24] Dana Scott, *Outline of a mathematical theory of computation*, Technical Monograph PRG–2, Oxford University Computing Laboratory, 1970. 1

[25] 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

[26] Anne S. Troelstra and Helmut Schwichtenberg, *Basic proof theory*, 2nd ed., Cambridge University Press, 2000. 1.5

Index