# MINLOG REFERENCE MANUAL

HELMUT SCHWICHTENBERG

Contents



*Date*: April 8, 2024.

*Acknowledgement.* The Minlog system has been under development since around 1990; its first appearance in print is in [29]. 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),
- Fredrik Nordvall Forsberg (Haskell translation),
- Valentin Herrmann (Minlogpad),
- 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),
- Nils Köpp (many contributions, in particular corecursion, coinduction, lookahead for stream-represented real numbers),
- Ralph Matthes (documentation),
- Kenji Miyamoto (corecursion, coinduction),
- 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),
- Franziskus Wiesnet (constructive analysis with exact real numbers, documentation),
- 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 [38] and Ershov [12]. 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 [14]. 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 $\mathbf{F} := \mathtt{ff} \equiv \mathtt{tt}$, where EqD 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 [14] Dialectica interpretation and the so-called $A$-translation of Friedman [13] and Dragalin [10]. The difference of TCF to well-established theories like Martin-Löf's [23] 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. [4]).
(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 explicitly refer to the generation tree. Such trees are quite often countably branching, and hence we allow infinitary free algebras from the outset.

The freeness of the constructors is expressed by requiring that their ranges are disjoint and that they are injective. Moreover, we view the free algebra as a domain and require that its bottom element is not in the range of the constructors. Hence the constructors are total and non-strict. For the notion of totality cf. [39, 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. [5]) 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

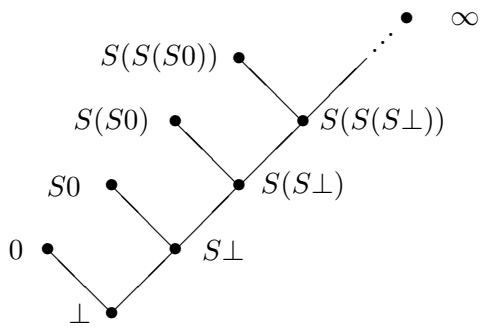


FIGURE 1. The domain of natural numbers

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. [39]. 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 $T_\rho$ with the total functionals of type $\rho$ 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, $\mathtt{n}, \mathtt{n0}, \mathtt{n1}, \mathtt{n2}, \ldots$ range over total natural numbers, and $\mathtt{n\hat{}}, \mathtt{n\hat{}0}, \mathtt{n\hat{}1}, \mathtt{n\hat{}2}, \ldots$ are general variables. This amounts to an abbreviation of

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

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 represented by means of boolean valued functions, hence the rewrite mechanism applies to them as well. Equality is decidable for finitary algebras only; for infinitary algebras one uses the inductively defined Leibniz equality instead.

1.5. **Minimal logic, proof transformation.** For generalities about minimal logic cf. [41] or [40]. A description of the theory behind the present implementation can be found in [35].

1.6. **Comparison with Coq and Isabelle.** Coq [9] 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 [7].

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 explicitly via assumptions.

(v) Minlog's automated proof search `search` tool is based on [25]; it produces proofs in minimal logic. In addition, Coq has many strong tactics, for instance `Omega` for quantifier free Presburger arithmetic, `Arith` for proving simple arithmetic properties and `Ring` for proving consequences of the ring axioms. Similar tactics exist in Isabelle. These tactics tend to produce rather long proofs, which is due to the fact that equality arguments are carried out explicitly. This is avoided in Minlog by relativizing every proof to a set of rewrite rules, and 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, for efficiency reasons[1] Minlog also has a primitive product type formation.

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` (written $\circ$ in text and displayed `eps` in Minlog) 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`.

[1]Usage of a primitive product will increase efficiency when normalizing proofs via normalization-by-evaluation. This is done by first translating a proof into a term, then normalizing the term and finally translating it back into a proof. When translating a proof into a term, for instance at existence introduction a term of product type is formed, whose components need to be accessed. Such computations will be done at the object level (since we use normalization-by-evaluation) and therefore be faster when primitive pairing is used(pp.

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 $\vartheta$ 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 $\vartheta$ whose domain consists of type variables $\alpha$, 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 $\vartheta$ *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 $\vartheta$ in a term $r$, provided $\vartheta$ is admissible for all $x \in \mathrm{FV}(r)$ (in short: $\vartheta$ 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 $\vartheta$ is admissible for $\lambda_x r$, then $\vartheta^y_x$ 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 $\vartheta$ be admissible for the term $r$. Then $\overline{r\vartheta} = \overline{r}\vartheta$.*

*Proof. Case* $x$. $\overline{x\vartheta} = \overline{\vartheta(x)}$ holds since $\vartheta$ 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 $\vartheta$ is admissible for $r$ and $\eta$ 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 $\eta$ 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. □

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

Now consider a type-object-predicate-assumption substitution $\vartheta$ with type variables $\alpha$, object variables $x$, predicate variables $P$ and assumption variables $u$ in its domain. Again we allow that the type $\sigma$ 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 $\vartheta$ in a derivation $M$ is defined as follows, provided $\vartheta$ is admissible for $M$. We define $M\vartheta$ by induction on $M$.

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

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 [21] 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.

In our constructors of an algebra we allow a certain "nesting" w.r.t. already generated algebras $\bar{\iota}$. Then the cototal ideals of this algebra will "incorporate" total ideals of type $\bar{\iota}$. An example are finitely branching non-wellfounded trees, where $\bar{\iota}$ is the list type. Such cototal ideals will be useful as witnesses of "nested coinductive/inductive definitions".

We inductively define *type forms*

$$\rho, \sigma ::= \alpha \mid \rho \to \sigma \mid \mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k}$$

with $\alpha, \xi$ type variables and $k \geq 1$ (since we want our algebras to be inhabited). Note that $(\rho_\nu)_{\nu<n} \to \sigma$ means $\rho_0 \to \ldots \to \rho_{n-1} \to \sigma$, associated to the right.

Let $\mathrm{FV}(\rho)$ denote the set of type variables free in $\rho$. We define $\mathrm{SP}(\alpha, \rho)$ "$\alpha$ occurs at most *strictly positive* in $\rho$" by induction on $\rho$.

$$\mathrm{SP}(\alpha, \beta) \qquad \frac{\alpha \notin \mathrm{FV}(\rho) \quad \mathrm{SP}(\alpha, \sigma)}{\mathrm{SP}(\alpha, \rho \to \sigma)} \qquad \frac{\mathrm{SP}(\alpha, \rho_{i\nu}) \text{ for all } i < k,\ \nu < n_i}{\mathrm{SP}(\alpha, \mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k})}$$

Now we can define $\mathrm{Ty}(\rho)$ "$\rho$ is a *type*", again by induction on $\rho$.

$$\mathrm{Ty}(\alpha) \qquad \frac{\mathrm{Ty}(\rho) \quad \mathrm{Ty}(\sigma)}{\mathrm{Ty}(\rho \to \sigma)}$$

$$\frac{\mathrm{Ty}(\rho_{i\nu}) \text{ and } \mathrm{SP}(\xi, \rho_{i\nu}) \text{ for all } i < k,\ \nu < n_i \quad \xi \notin \mathrm{FV}(\rho_{0\nu}) \text{ for all } \nu < n_0}{\mathrm{Ty}(\mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k})}$$

We call

$$\iota := \mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k}$$

an *algebra*. Sometimes it is helpful to display the type parameters and write $\iota(\vec{\alpha}, \vec{\beta})$, where $\vec{\alpha}, \vec{\beta}$ are all type variables except $\xi$ free in some $\rho_{i\nu}$, and $\vec{\alpha}$ are the ones occuring only strictly positive. If we write the $i$-th component of $\iota$ in the form $(\rho_\nu(\xi))_{\nu<n} \to \xi$, then we call

$$(\rho_\nu(\iota))_{\nu<n} \to \iota$$

the $i$-th *constructor type* of $\iota$.

In $(\rho_\nu(\xi))_{\nu<n} \to \xi$ we call $\rho_\nu(\xi)$ a *parameter* argument type if $\xi$ does not occur in it, and a *recursive* argument type otherwise. A recursive argument type $\rho_\nu(\xi)$ is *nested* if it has an occurrence of $\xi$ in a strictly positive parameter position of another (previously defined) algebra, and unnested otherwise. An algebra $\iota$ is called *nested* if it has a constructor with at least one nested recursive argument type, and *unnested* otherwise.

Every type $\rho$ should have a *total inhabitant*, i.e., a closed term of this type built solely from constructors, variables and assumed total inhabitants of some of its (type) variables. To ensure this we have required that for every algebra $\mu_\xi((\rho_{i\nu})_{\nu<n_i} \to \xi)_{i<k}$ the initial $(\rho_{0\nu})_{\nu<n_0} \to \xi$ has no recursive argument types. Note that it might not be necessary to actually use assumed total inhabitants for all variables of a type. An example is the list type $\mathbf{L}(\alpha)$, which has the Nil constructor as a total inhabitant. However, for the type $\mathbf{L}(\alpha)^+$ $(:= \mu_\xi(\alpha \to \xi, \alpha \to \xi \to \xi))$ we need to assume a total inhabitant of $\alpha$.

Here are some examples of algebras.

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

Examples of algebras strictly positive in their type parameters are

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

An example of a nested algebra is

$$\mathbf{T} := \mu_\xi(\mathbf{L}(\xi) \to \xi) \quad \text{(finitely branching trees)}.$$

Note that $\mathbf{T}$ has a total inhabitant since $\mathbf{L}(\alpha)$ has one (given by the Nil constructor).

*Remark* (Substitution for type parameters). Let $\rho \in \mathrm{Ty}(\vec{\alpha})$; we write $\rho(\vec{\alpha})$ for $\rho$ 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* $\mathrm{C}_i$ 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,

$(\mathrm{Inl}_{\rho\sigma})^{\rho\to\rho+\sigma}, (\mathrm{Inr}_{\sigma\rho})^{\sigma\to\rho+\sigma}$ for the sum type $\rho + \sigma$,

$\mathrm{Branch}\colon \mathbf{L}(\mathbf{T}) \to \mathbf{T}$ for the type $\mathbf{T}$ of finitely branching trees.

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). Another example uses the parametrized algebra

$$\mathbf{R}(\alpha) := \mu_\xi(\alpha \to \xi, \alpha \to \xi, \alpha \to \xi, \xi \to \xi \to \xi \to \xi)$$

(labelled read-and-finally-write-one-digit trees), whose constructors we name

$\mathrm{Put}_d\colon \alpha \to \mathbf{R}(\alpha)$ $(d \in \{-1, 0, 1\})$ finally write $d$ and continue,

$\mathrm{Get}\colon \mathbf{R}(\alpha) \to \mathbf{R}(\alpha) \to \mathbf{R}(\alpha) \to \mathbf{R}(\alpha)$ read.

Using $\mathbf{R}(\alpha)$ we then define

$$\mathbf{W} := \mu_\xi(\xi, \mathbf{R}(\xi) \to \xi) \quad \text{(nested alternating read-write trees)}$$

with constructors

$\mathrm{Stop}\colon \mathbf{W}$ stop,

$\mathrm{Cont}\colon \mathbf{R}(\mathbf{W}) \to \mathbf{W}$ branch by applying a read-write instruction, and continue.

Later we will consider ideals built with finite read-write instructions, but infinitely many alternations, via a "nested inductive/coinductive" definition.

One can extend the definition of algebras and types to *simultaneously defined algebras*. Instead of $\xi$ we consider a list $\vec{\xi} = \xi_0, \dots, \xi_{N-1}$ of type-variables and generate $\vec{\xi}, j$-constructor types $(j < N)$ by

$$\frac{(\rho_{i\nu}(\vec{\beta}) \in \mathrm{Ty}(\vec{\alpha}, \vec{\beta}))_{i<k;\nu<n_i}}{((\rho_{i\nu}(\vec{\xi}))_{\nu<n_i} \to \xi_j) \in \mathrm{KT}_{\vec{\xi},j}(\vec{\alpha})}.$$

Let $k = \sum_{j<N} k_j$ with $k_j \geq 1$ and $m_j := \sum_{l<j} k_j$, hence $\mathrm{m}_j + k_j = m_{j+1}$. For $m_j \leq i < m_{j+1}$ let $\kappa_i \in \mathrm{KT}_{\vec{\xi},j}(\vec{\alpha})$. Then all $\iota_j$ in $\vec{\iota} := \mu_{\vec{\xi}}(\kappa_0, \dots, \kappa_{k-1})$ are in $\mathrm{Alg}(\vec{\alpha})$. To ensure total inhabitants of the algebra we require that the initial constructor type for $\iota_j$ has argument types involving $\iota_i$ for $i < j$ only. — 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 $\rho$ 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 form $\iota$ is *structure-finitary* if in its generation the rule leading to $\sigma \to \rho$ has not been used. It is *finitary* if in addition it has no type

variables. 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. The nested algebra $\mathbf{T}$ above is finitary. 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 $\mathbf{N}$, $\mathbf{P}$, $\mathbf{L}(\rho)$, $\mathbf{D}$, $\mathbf{O}$, $\mathbf{T}_{n+1}$ and $\mathbf{T}$ are not.

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

$$\begin{aligned}
&\mathrm{lev}(\iota) := 0,\\
&\mathrm{lev}(\rho\to\sigma) := \max(\mathrm{lev}(\sigma), 1+\mathrm{lev}(\rho)),\\
&\mathrm{lev}(\rho\times\sigma) := \max(\mathrm{lev}(\sigma),\mathrm{lev}(\rho)) \qquad \text{for the primitive product.}
\end{aligned}$$

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

For a type $\rho(\vec\alpha\,)\in\mathrm{Ty}(\vec\alpha\,)$ (hence also for an algebra $\iota(\vec\alpha\,)\in\mathrm{Alg}(\vec\alpha\,)$) we define the *map* operator

$$\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\rho(\vec\alpha\,)}\colon \rho(\vec\sigma\,)\to(\vec\sigma\to\vec\tau\,)\to\rho(\vec\tau\,)$$

(where $(\vec\sigma\to\vec\tau)\to\rho(\vec\tau\,)$ means $(\sigma_1\to\tau_1)\to\ldots\to(\sigma_n\to\tau_n)\to\rho(\vec\tau\,)$). If none of $\vec\alpha$ appears free in $\rho(\vec\alpha\,)$ let

$$\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\rho(\vec\alpha\,)}x\vec f := x.$$

Otherwise we use an outer recursion on $\rho(\vec\alpha\,)$ and if $\rho(\vec\alpha\,)$ is $\iota(\vec\alpha\,)$ an inner one on $x$. In case $\rho(\vec\alpha\,)$ is $\iota(\vec\alpha\,)$ we abbreviate $\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\iota(\vec\alpha\,)}$ by $\mathcal{M}^{\vec\sigma\to\vec\tau}_{\iota}$ or $\mathcal{M}^{\vec\tau}_{\iota(\vec\sigma\,)}$.

The immediate cases for the outer recursion are

$$\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\alpha_i}x\vec f := f_i x, \qquad \mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}(\sigma\to\rho)}h\vec f x := \mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\rho}(hx)\vec f.$$

It remains to consider $\iota(\vec\pi(\vec\alpha\,))$. In case $\vec\pi(\vec\alpha\,)$ is not $\vec\alpha$ let

$$\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\iota(\vec\pi(\vec\alpha\,))}x\vec f := \mathcal{M}^{\vec\pi(\vec\sigma\,)\to\vec\pi(\vec\tau\,)}_{\iota}x(\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\pi_i(\vec\alpha\,)}\cdot\vec f\,)_{i<|\vec\pi\,|}$$

with $\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\pi_i(\vec\alpha\,)}\cdot\vec f := \lambda_x\mathcal{M}^{\vec\sigma\to\vec\tau}_{\lambda_{\vec\alpha}\pi_i(\vec\alpha\,)}x\vec f$. In case $\vec\pi(\vec\alpha\,)$ is $\vec\alpha$ we use recursion on $x$ and define for a constructor $\mathrm{C}_i\colon(\rho_\nu(\vec\sigma,\iota(\vec\sigma\,)))_{\nu<n}\to\iota(\vec\sigma\,)$

$$\mathcal{M}^{\vec\sigma\to\vec\tau}_{\iota}(\mathrm{C}_i\vec x\,)\vec f$$

to be the result of applying $\mathrm{C}'_i$ of type $(\rho_\nu(\vec\tau,\iota(\vec\tau\,)))_{\nu<n}\to\iota(\vec\tau\,)$ (the same constructor as $\mathrm{C}_i$ with only the type changed) to, for each $\nu<n$,

$$\mathcal{M}^{\vec\sigma,\iota(\vec\sigma\,)\to\vec\tau,\iota(\vec\tau\,)}_{\lambda_{\vec\alpha,\beta}\rho_\nu(\vec\alpha,\beta)}x_\nu\vec f(\mathcal{M}^{\vec\sigma\to\vec\tau}_{\iota}\cdot\vec f\,).$$

Note that the final function argument provides the recursive call w.r.t. the recursion on $x$.

Here are some examples.

$$\mathcal{M}^{\tau}_{\mathbf{L}(\sigma)}\mathrm{Nil}f^{\sigma\to\tau} := \mathrm{Nil},$$

$$\mathcal{M}^{\tau}_{\mathbf{L}(\sigma)}(x^{\sigma} :: l^{\mathbf{L}(\sigma)})f^{\sigma\to\tau} := (fx) :: (\mathcal{M}\, l\, f),$$

$$\mathcal{M}^{\tau}_{\mathbf{R}(\sigma)}(\mathrm{Put}_d^{\sigma\to\mathbf{R}(\sigma)}x^{\sigma})f^{\sigma\to\tau} := \mathrm{Put}_d^{\tau\to\mathbf{R}(\tau)}(fx) \quad (d \in \{-1,0,1\}),$$

$$\mathcal{M}^{\tau}_{\mathbf{R}(\sigma)}(\mathrm{Get}\, y_1^{\mathbf{R}(\sigma)} y_2^{\mathbf{R}(\sigma)} y_3^{\mathbf{R}(\sigma)})f^{\sigma\to\tau} := \mathrm{Get}(\mathcal{M}\, y_1\, f)(\mathcal{M}\, y_2\, f)(\mathcal{M}\, y_3\, f).$$

A slightly more complex example is the nested algebra $\mathbf{T}(\alpha)$ with the single constructor $\mathrm{C}\colon \mathbf{L}(\alpha + \mathbf{T}(\alpha)) \to \mathbf{T}(\alpha)$:

$$\mathcal{M}^{\tau}_{\mathbf{T}(\sigma)}(\mathrm{C}x^{\mathbf{L}(\sigma+\mathbf{T}(\sigma))})f := \mathrm{C}'(\mathcal{M}^{\sigma,\mathbf{T}(\sigma)\to\tau,\mathbf{T}(\tau)}_{\lambda_{\alpha,\beta}\mathbf{L}(\alpha+\beta)}xfg)$$

with $g\colon \mathbf{T}(\sigma) \to \mathbf{T}(\tau)$ defined by $\mathcal{M}^{\tau}_{\mathbf{T}(\sigma)} \cdot f$.

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)`.

To introduce (possibly simultaneous) free algebras we use

`add-algs`.

Examples are

```
(add-algs "nat" '("Zero" "nat") '("Succ" "nat=>nat"))

(add-algs "list" 'prefix-typeop
  '("list" "Nil")
  '("alpha=>list=>list" "Cons"))

(add-algs (list "ltlist" "ltree") 'prefix-typeop
  '("ltlist" "LEmpty")
  '("ltree=>ltlist=>ltlist" "LTcons")
  '("alpha=>ltree" "LLeaf")
  '("ltlist=>ltree" "LBranch"))
```

The final example simultaneously introduces the two free algebras . The constructors are introduced as "self-evaluating" constants; they play a special role in our semantics for normalization by evaluation.

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, |
| (nested-alg-name? *name*) | complete test, |
| (ground-type? *x*) | incomplete test. |

To remove names for algebras we use

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

Standard examples for finitary free algebras are the type `nat` of unary natural numbers, and the algebra of binary trees.

Minlog initially provides the finitary free algebra `unit` consisting of exactly one element, and `boole` of booleans; objects of the latter type are (cf. [5]) `true`, `false` and families of terms of this type, and in addition the bottom object of type `boole`. Moreover, Minlog initially has the structure-finitary algebras `yprod` for product types $\rho\times\sigma$ and `ysum` for sum types $\rho+\sigma$. For convenience and readability there are also structure-finitary algebras for sum types where one component is the unit type: `uysum` for $\mathbf{U}+\sigma$ and `ysumu` for $\rho+\mathbf{U}$.

Tests:

(arrow-form? *type*),
(star-form? *type*) for the primitive product,
(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, i.e., the primitive product,

`(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.

A test function for types is

`(type?` *x*`)`.

For displaying types we have

`(type-to-string` *type*`)`,

which is defined by

`(token-tree-to-string (type-to-token-tree` *type*`))`.

For better line breaks in the display one can use

`(pp` *type*`)`,

which is defined by

`(token-tree-to-pp-tree (type-to-token-tree` *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 [39, 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, `n`, `n0`, `n1`, `n2`, ... range over total natural numbers, and `n^`, `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. One can view the already chosen default variable names for some types by

`(display-default-varnames .` *types*`).`

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: for arbitrary algebras we have recursion, (guarded) general recursion and corecursion operators and also destructors, and for finitary algebras equality, existence and structural existence operators. We also need ex-falso-quodlibet and existence elimination operators. 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 $\rho$ 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. **Structural recursion operators and Gödel's T.** Recall the definition of types and constructor types in section 2, and the examples given there. The (structural) higher type *recursion operators* $\mathcal{R}^\tau_\iota$ (introduced by Gödel [14]) are used to construct maps from the algebra $\iota$ to $\tau$, by recursion on the structure of $\iota$. For instance, $\mathcal{R}^\tau_{\mathbf{N}}$ 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 on $n$.

Generally, we define the type of the recursion operator for the algebra $\iota = \mu_\xi\,(\kappa_0, \dots, \kappa_{k-1})$ and result type $\tau$. Let the $i$-th $\xi$-constructor type for $\iota$ be

$$\kappa_i(\xi) = (\rho_{i\nu}(\xi))_{\nu<n_i} \to \xi.$$

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

$$\iota \to (\kappa_i(\iota, \tau))_{i<k} \to \tau$$

with *step types* (w.r.t. the result type $\tau$)

$$\kappa_i(\iota, \tau) := (\rho_{i\nu}(\iota \times \tau))_{\nu<n_i} \to \tau.$$

The recursion argument is of type $\iota$.

*Remark.* Usage of $\iota \times \tau$ rather than $\tau$ in the step types can be seen as a "strengthening", since then one has more data available to construct the value of type $\tau$. Moreover, for unnested recursive argument types $\vec\sigma \to \tau$ we avoid the product type in $\vec\sigma \to \iota \times \tau$ and take the two argument types $\vec\sigma \to \iota$ and $\vec\sigma \to \tau$ instead ("duplication").

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

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

$\mathcal{R}^\tau_{\rho+\sigma}\colon \rho+\sigma \to (\rho\to\tau)\to(\sigma\to\tau)\to\tau$,
$\mathcal{R}^\tau_{\rho\times\sigma}\colon \rho\times\sigma \to (\rho\to\sigma\to\tau)\to\tau$,
$\mathcal{R}^\tau_{\mathbf{T}}\colon \mathbf{T} \to (\mathbf{L}(\mathbf{T}\times\tau)\to\tau)\to\tau$,
$\mathcal{R}^\tau_{\mathbf{W}}\colon \mathbf{W} \to \tau\to(\mathbf{R}(\mathbf{W}\times\tau)\to\tau)\to\tau$.

One can extend the definition of the (structural) recursion operators to simultaneously defined algebras $\vec{\iota} = \mu_{\vec{\xi}}(\kappa_0,\dots,\kappa_{k-1})$ and result types $\vec{\tau}$. Pick $k_j, m_j$ as above (for simultaneously defined algebras). For $m_j \le i < m_{j+1}$ let $\kappa_i \in \mathrm{KT}_{\vec{\xi},j}(\vec{Y})$ be the $\vec{\xi}, j$-constructor type

$$(\rho_{i\nu}(\vec{\xi}))_{\nu<n_i} \to \xi_j.$$

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

$$\iota_j \to (\kappa_i(\vec{\iota},\vec{\tau}))_{i<k} \to \tau_j,$$

with step types

$$\kappa_i(\vec{\iota},\vec{\tau}) := (\rho_{i\nu}(\vec{\iota}\times\vec{\tau}))_{\nu<n_i} \to \tau_l \quad (m_l \le i < m_{l+1}).$$

Here $\vec{\iota}\times\vec{\tau}$ is the component-wise product. Again for an unnested recursive argument type $\vec{\sigma}\to\tau_i$ we use duplication to avoid the product type in $\vec{\sigma}\to\iota_i\times\tau_i$ and take the two argument types $\vec{\sigma}\to\iota_i$ and $\vec{\sigma}\to\tau_i$ instead.

Note that $k$ is the *total* number of constructors, and that the recursion argument is of type $\iota_j$. We will often omit the upper indices $\vec{\iota},\vec{\tau}$ when they are clear from the context. In case of a non-simultaneous free algebra we write $\mathcal{R}^\tau_\iota$ for $\mathcal{R}^{\iota,\tau}_1$. o— An example of a simultaneous recursion on tree lists and trees will be given below.

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

4.2. **Conversion.** We define a *conversion relation* $\mapsto_\rho$ between terms of type $\rho$ by

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

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

(3) $\quad \mathcal{R}^\tau_\iota(\mathrm{C}^\iota_i\vec{N})\vec{M} \mapsto M_i(\mathcal{M}^{\iota\to\iota\times\tau}_{\lambda_\alpha\rho_\nu(\alpha)}N_\nu\lambda_x\langle x^\iota, \mathcal{R}^\tau_\iota x\vec{M}\rangle)_{\nu<n}$.

where in (3) for simplicity we have spelled out the non-simultaneous case only; the $i$-th $\xi$-constructor type is assumed to be $(\rho_\nu(\xi))_{\nu<n}\to\xi$. Note

that (3) uses the map operator defined above. In the special case $\rho_\nu(\alpha) = \alpha$ we can avoid the product type and instead of the pair

$$\mathcal{M}^{\iota\to\iota\times\tau}_{\lambda_\alpha\alpha} N_\nu \lambda_x \langle x^\iota, \mathcal{R}^\tau_\iota x\vec{M}\rangle \quad \text{i.e.,} \quad \langle N^\iota_\nu, \mathcal{R}^\tau_\iota N_\nu \vec{M}\rangle$$

take its two components $N^\iota_\nu$ and $\mathcal{R}^\tau_\iota N_\nu \vec{M}$ as separate arguments of $M_i$.

The rule (1) is called $\beta$-conversion, and (2) $\eta$-conversion; their left hand sides are called *$\beta$-redexes* or *$\eta$-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. 4.4), 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* $\varepsilon^\rho$ of a type $\rho \in \mathrm{Ty}$:

$$\varepsilon^{\iota_j} := \mathrm{C}^{\vec{\iota}}_{i_j} \varepsilon^{\vec{\rho}} (\lambda_{\vec{x}_1} \varepsilon^{\iota_{j_1}}) \dots (\lambda_{\vec{x}_n} \varepsilon^{\iota_{j_n}}), \quad \varepsilon^{\rho\to\sigma} := \lambda_x \varepsilon^\sigma.$$

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

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

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 $(\mathbf{Ts}(\alpha), \mathbf{T}(\alpha))$ simultaneously defined in 2.3 (we write them without the parameter $\alpha$ here), whose constructors $\mathrm{C}_i^{(\mathbf{Ts},\mathbf{T})}$ 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., $\mathbf{T}(\alpha)$) are just the finitely branching trees, which carry objects of type $\alpha$ on their leaves.

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

$$\begin{aligned} &\delta_0 := \sigma, && \delta_2 := \alpha \to \tau, \\ &\delta_1 := \mathbf{T} \to \tau \to \mathbf{Ts} \to \sigma \to \sigma, && \delta_3 := \mathbf{Ts} \to \sigma \to \tau. \end{aligned}$$

Hence the types of the recursion operators are

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

The recursion operator $\mathcal{R}_{\mathbf{T}}$ or explicitly $\mathcal{R}_{\mathbf{T}}^{(\mathbf{Ts},\mathbf{T}),(\sigma,\tau)}$ is displayed as

$$\texttt{(Rec } \mathbf{T} \to \tau \ \ \mathbf{Ts} \to \sigma\texttt{)},$$

where the first arrow type indicates the type of the recursion and its value type, and the remaining arrow types provide the value types for the simultaneously defined algebras.

We now introduce some special cases of structural recursion and also a generalization; both will be important later on.

*Simplified simultaneous recursion.* In a recursion on simultaneously defined algebras one may need to recur on some of those algebras only. Then we can simplify the type of the recursion operator accordingly, as follows.

(i) Only consider the relevant constructors, i.e., those mapping into relevant algebras.
(ii) Shorten their types by omitting all argument types containing irrelevant algebras.
(iii) Let $(\rho_\nu(\vec{\xi}))_{\nu<n} \to \xi_j$ be a shortened $\vec{\xi}, j$-constructor type. Form its simplified step type as $(\rho_\nu(\vec{\iota} \times \vec{\tau}))_{\nu<n} \to \tau_j$ where $\vec{\iota}$ are the relevant algebras and $\vec{\tau}$ the assigned value types.

If in the $(\mathbf{Ts}, \mathbf{T})$-example we want to recur on $\mathbf{Ts}$ only, the step types are

$$\delta_0 := \sigma, \quad \delta_1 := \mathbf{Ts} \to \sigma \to \sigma.$$

Hence the type of the simplified recursion operator is

$$\mathcal{R}_{\mathbf{Ts}} \colon \mathbf{Ts} \to \delta_0 \to \delta_1 \to \sigma;$$

It is displayed as `(Rec Ts → σ)`, where the missing arrow type $\mathbf{T} \to \tau$ indicates that we have a simplified simultaneous recursion.

An example is the recursive definition of the length of a tree list. The recursion equations are

$$\mathrm{lh}(\mathrm{Empty}) = 0, \quad \mathrm{lh}(\mathrm{Tcons}\, b\, bs) = \mathrm{lh}(bs) + 1.$$

This length function can be defined by an ordinary (i.e., non-simplified) simultaneous recursion operator as

$$\lambda_{as}\mathcal{R}^{(\mathbf{Ts},\mathbf{T}),(\mathbf{N},\tau)}_{\mathbf{Ts}}\, as^{\mathbf{Ts}} 0(\lambda_{a,y,bs,n} n + 1)(\lambda_x e^\tau)(\lambda_{as,n} e^\tau).$$

This simultaneous recursion simplifies to

$$\lambda_{as}\mathcal{R}^{\mathbf{N}}_{\mathbf{Ts}}\, as^{\mathbf{Ts}} 0(\lambda_{bs,n} n + 1).$$

*Cases.* There is an important variant of recursion, where no recursive calls occur. This variant is called the *cases operator*; it distinguishes cases according to the outer constructor form. Here all step types have the form

$$\delta_i^{\vec{\iota},\vec{\tau}} := \vec{\rho} \to (\vec{\sigma}_\nu \to \iota_{j\nu})_{\nu<n} \to \tau_j.$$

The intended meaning of the cases operator is given by the conversion rule

$$\mathcal{C}_j(\mathrm{C}_i^{\vec{\iota}}\vec{N})\vec{M} \mapsto M_i\vec{N}. \tag{4}$$

Notice that only those step terms are used whose value type is the present $\tau_j$; this is due to the fact that there are no recursive calls. Therefore the type of the cases operator is

$$\mathcal{C}^{\vec{\iota}}_{\iota_j\to\tau_j} \colon \iota_j \to \delta_{i_0} \to \ldots \to \delta_{i_{q-1}} \to \tau_j,$$

where $\delta_{i_0}, \ldots, \delta_{i_{q-1}}$ consists of all $\delta_i$ with value type $\tau_j$. We write $\mathcal{C}^{\tau_j}_{\iota_j}$ or even $\mathcal{C}_j$ for $\mathcal{C}^{\vec{\iota}}_{\iota_j\to\tau_j}$.

The simplest example (for type $\mathbf{B}$) is *if-then-else*. Another example is

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

It can be used to define the *predecessor* function on $\mathbf{N}$, i.e., $\mathrm{P}0 := 0$ and $\mathrm{P}(\mathrm{S}n) := n$, by the term

$$\mathrm{P}m := \mathcal{C}^{\mathbf{N}}_{\mathbf{N}} m 0(\lambda_n n).$$

In the $(\mathbf{Ts}, \mathbf{T})$-example we have

$$\mathcal{C}^{\tau_0}_{\mathbf{Ts}} \colon \mathbf{Ts} \to \tau_0 \to (\mathbf{T} \to \mathbf{Ts} \to \tau_0) \to \tau_0.$$

When computing the value of a cases term, we do not want to (eagerly) evaluate all arguments, but rather compute the test argument first and depending on the result (lazily) evaluate at most one of the other arguments. This phenomenon is well known in functional languages; for instance, in

SCHEME the `if`-construct is called a *special form* (as opposed to an operator). Therefore instead of taking the cases operator applied to a full list of arguments, one rather uses a `if`-construct to build this term; it differs from the former only in that it employs lazy evaluation. Hence the predecessor function is written [`if` $m\ 0\ \lambda_n n$] (which is often written in the form [**case** $m$ **of** $0 \mid \lambda_n n$]).

*General recursion with respect to a measure.* In practice it often happens that one needs to recur to an argument which is not an immediate component of the present constructor object; this is not allowed in structural recursion. Of course, in order to ensure that the recursion terminates we have to assume that the recurrence is w.r.t. a given well-founded set; for simplicity we restrict ourselves to the algebra $\mathbf{N}$. However, we do allow that the recurrence is with respect to a measure function $\mu$, with values in $\mathbf{N}$. The operator $\mathcal{F}$ of *general recursion* then is defined by

$$\mathcal{F}\mu x G = Gx(\lambda_y[\mathbf{if}\ \mu y < \mu x\ \mathbf{then}\ \mathcal{F}\mu y G\ \mathbf{else}\ \varepsilon]), \tag{5}$$

where $\varepsilon$ denotes a canonical inhabitant of the range. One can see easily that $\mathcal{F}$ is definable from an appropriate structural recursion operator.

4.3. **Corecursion.** It is well known that an arbitrary "reduction sequence" beginning with a term in Gödel's T terminates. For this to hold it is essential that the constants allowed in T are restricted to constructors C and recursion operators $\mathcal{R}$. A consequence will be that every closed term of a base type denotes a total ideal. The conversion rules for $\mathcal{R}$ (cf. 4.2) work from the leaves towards the root, and terminate because total ideals are well-founded. If however we deal with cototal ideals (infinitary derivations for example), then a similar operator is available to define functions with cototal ideals as values, namely "corecursion".

To understand the type of a corecursion operator recall the constructor types $\kappa_i(\iota)$ of an algebra $\iota = \mu_\xi(\kappa_0, \dots, \kappa_{k-1})$:

$$(\rho_{i\nu}(\iota))_{\nu<n_i} \to \iota \quad (i < k).$$

The product of these $k$ constructor types is isomorphic to

$$\sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota) \to \iota$$

and the type of the recursion operator $\mathcal{R}^\tau_\iota$ is isomorphic to

$$\iota \to (\sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota \times \tau) \to \tau) \to \tau.$$

Dually for the algebra $\iota$ the type of its *destructor* $D_\iota$ (defined below) is

$$\iota \to \sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota).$$

The corecursion operator ${}^{\mathrm{co}}\mathcal{R}^\tau_\iota$ is used to construct a mapping from $\tau$ to $\iota$ by "corecursion" on the structure of $\iota$. Its type is

$$\tau \to (\tau \to \sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota+\tau)) \to \iota.$$

We list the types of the corecursion operators for some algebras:

$$\begin{aligned}
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{B}} \colon \tau \to (\tau \to \mathbf{U}+\mathbf{U}) \to \mathbf{B},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{N}} \colon \tau \to (\tau \to \mathbf{U}+(\mathbf{N}+\tau)) \to \mathbf{N},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{P}} \colon \tau \to (\tau \to \mathbf{U}+(\mathbf{P}+\tau)+(\mathbf{P}+\tau)) \to \mathbf{P},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{D}} \colon \tau \to (\tau \to \mathbf{U}+(\mathbf{D}+\tau)\times(\mathbf{D}+\tau)) \to \mathbf{D},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{L}(\rho)} \colon \tau \to (\tau \to \mathbf{U}+\rho\times(\mathbf{L}(\rho)+\tau)) \to \mathbf{L}(\rho),\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{I}} \colon \tau \to (\tau \to \mathbf{U}+(\mathbf{I}+\tau)+(\mathbf{I}+\tau)+(\mathbf{I}+\tau)) \to \mathbf{I}.
\end{aligned}$$

The conversion relation for each of these is defined below. For $f\colon \rho \to \tau$ and $g\colon \sigma \to \tau$ we denote $\lambda_x(\mathcal{R}^\tau_{\rho+\sigma} x f g)$ of type $\rho+\sigma \to \tau$ by $[f,g]$, and similary for ternary sumtypes etcetera. $x_1$, $x_2$ are shorthand for the two projections of $x$ of type $\rho\times\sigma$. The identity functions id below are of type $\iota \to \iota$ with $\iota$ the respective algebra.

$$\begin{aligned}
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{B}} NM &\mapsto [\lambda_\mathtt{tt}, \lambda_\mathtt{ff}](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{N}} NM &\mapsto [\lambda_0, \lambda_x(\mathrm{S}([\mathrm{id}^{\mathbf{N}\to\mathbf{N}}, \lambda_y({}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{N}} yM)]x))](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{P}} NM &\mapsto [\lambda_1, \lambda_x(S_0([\mathrm{id}, P_{\mathbf{P}}]x)), \lambda_x(S_1([\mathrm{id}, P_{\mathbf{P}}]x))](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{D}} NM &\mapsto [\lambda_0, \lambda_x(\mathrm{C}([\mathrm{id}, P_{\mathbf{D}}]x_1)([\mathrm{id}, P_{\mathbf{D}}]x_2))](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{L}(\rho)} NM &\mapsto [\lambda_\mathrm{Nil}, \lambda_x(x_1 :: [\mathrm{id}, \lambda_y({}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{L}(\rho)} yM)]x_2)](MN),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{I}} NM &\mapsto [\lambda_\mathbb{I}, \lambda_x(\mathrm{C}_{-1}([\mathrm{id}, P_{\mathbf{I}}]x)), \lambda_x(\mathrm{C}_0([\mathrm{id}, P_{\mathbf{I}}]x)), \lambda_x(\mathrm{C}_1([\mathrm{id}, P_{\mathbf{I}}]x))]\\
&\quad (MN)
\end{aligned}$$

with $P_\alpha := \lambda_y({}^{\mathrm{co}}\mathcal{R}^\tau_\alpha yM)$ for $\alpha \in \{\mathbf{P}, \mathbf{D}, \mathbf{I}\}$.

The types of the corecursion operators for $\mathbf{T}$ and $\mathbf{W}$ are

$$\begin{aligned}
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{T}} \colon \tau \to (\tau \to \mathbf{L}(\mathbf{T}+\tau)) \to \mathbf{T},\\
&{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{W}} \colon \tau \to (\tau \to \mathbf{U}+\mathbf{R}(\mathbf{W}+\tau)) \to \mathbf{W}.
\end{aligned}$$

The conversion relation for each of these is defined by

$$\begin{aligned}
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{T}} NM &\mapsto \mathrm{Branch}(\mathcal{M}^{\mathbf{T}}_{\mathbf{L}(\mathbf{T}+\tau)}[\mathrm{id}^{\mathbf{T}\to\mathbf{T}}, \lambda_z {}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{T}} zM](MN)^{\mathbf{L}(\mathbf{T}+\tau)}),\\
{}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{W}} NM &\mapsto [\lambda_W_0, \lambda_x(W(\mathcal{M}^{\mathbf{W}}_{\mathbf{R}(\mathbf{W}+\tau)}[\mathrm{id}^{\mathbf{W}\to\mathbf{W}}, \lambda_z {}^{\mathrm{co}}\mathcal{R}^\tau_{\mathbf{W}} zM)]x))](MN).
\end{aligned}$$

An alternative notation for the former term is

$$\begin{aligned}{}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{T}}NM \mapsto \mathrm{Branch}(\mathcal{M}^{\mathbf{T}}_{\mathbf{L}(\mathbf{T}+\tau)}(\lambda_p[&\mathbf{case}\ p^{\mathbf{T}+\tau}\ \mathbf{of}\\ &\mathrm{Inl}\ a^{\mathbf{T}} \mapsto a \mid\\ &\mathrm{Inr}\ z^{\tau} \mapsto {}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{T}}zM])\\ &(MN)^{\mathbf{L}(\mathbf{T}+\tau)})\end{aligned}$$

and for the latter

$$\begin{aligned}{}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{W}}NM \mapsto [&\mathbf{case}\ (MN)^{\mathbf{U}+\mathbf{R}(\mathbf{W}+\tau)}\ \mathbf{of}\\ &\mathrm{Inl}\ \_ \mapsto W_0 \mid\\ &\mathrm{Inr}\ x \mapsto W(\mathcal{M}^{\mathbf{W}}_{\mathbf{R}(\mathbf{W}+\tau)}(\lambda_p[&\mathbf{case}\ p^{\mathbf{W}+\tau}\ \mathbf{of}\\ &&\mathrm{Inl}\ y^{\mathbf{W}} \mapsto y \mid\\ &&\mathrm{Inr}\ z^{\tau} \mapsto {}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{W}}zM])\\ &x^{\mathbf{R}(\mathbf{W}+\tau)}].\end{aligned}$$

The conversion rule for ${}^{\mathrm{co}}\mathcal{R}^{\tau}_{\iota}NM$ in the general case is defined similarly: we distinguish cases on $MN$ of type $\sum\big(\prod\vec{\rho}(\iota+\tau)\big)$. Suppose we are in the case of the $i$-th injection of a term $x$ of product type $\prod\vec{\rho}(\iota+\tau)$. Then we apply the $i$-th constructor $\mathrm{C}_i$ of type $\vec{\rho}(\iota)\to\iota$ as follows. The $\nu$-th argument of type $\rho_\nu(\iota)$ is obtained from the $\nu$-th component $x_\nu$ of $x$ as

$$\begin{aligned}\mathcal{M}^{\iota+\tau\to\iota}_{\lambda_\alpha\rho_\nu(\alpha)}x_\nu^{\rho_\nu(\iota+\tau)}(\lambda_p[&\mathbf{case}\ p^{\iota+\tau}\ \mathbf{of}\\ &\mathrm{Inl}\ y^{\iota} \mapsto y \mid\\ &\mathrm{Inr}\ z^{\tau} \mapsto {}^{\mathrm{co}}\mathcal{R}^{\tau}_{\iota}zM]).\end{aligned}$$

As an example of a function defined by corecursion (due to [3]) consider the transformation of an "abstract" real in the interval $[-1,1]$ into a stream representation using signed digits from $\{-1,0,1\}$. Assume that we work in an abstract (axiomatic) theory of reals, having an unspecified type $\rho$, and that we have a type $\sigma$ for rationals as well. Assume that the abstract theory provides us with a function $g\colon \rho\to\sigma\to\sigma\to\mathbf{B}$ comparing a real $x$ with a proper rational interval $p<q$:

$$\begin{aligned}g(x,p,q) &= \mathtt{tt}\to x\le q,\\ g(x,p,q) &= \mathtt{ff}\to p\le x.\end{aligned}$$

From $g$ we define a function $h\colon \rho\to\mathbf{U}+(\mathbf{I}+\rho)+(\mathbf{I}+\rho)+(\mathbf{I}+\rho)$ by

$$h(x) := \begin{cases} 2x+1 \text{ in rhs of left } \mathbf{I}+\rho & \text{if } g(x,-\frac{1}{2},0)=\mathtt{tt},\\ 2x \text{ in rhs of middle } \mathbf{I}+\rho & \text{if } g(x,-\frac{1}{2},0)=\mathtt{ff},\ g(x,0,\frac{1}{2})=\mathtt{tt},\\ 2x-1 \text{ in rhs of right } \mathbf{I}+\rho & \text{if } g(x,0,\frac{1}{2})=\mathtt{ff}.\end{cases}$$

$h$ is definable by a closed term $M$ in Gödel's T. Then the desired function $f\colon \rho \to \mathbf{I}$ transforming an abstract real $x$ into a cototal ideal (i.e., a stream) in $\mathbf{I}$ can be defined by

$$f(x) := {}^{\mathrm{co}}\mathcal{R}^{\rho}_{\mathbf{I}} x M.$$

This $f(x)$ will thus be a stream of digits $-1, 0, 1$.

We give another example of a function defined by corecursion, this time on the nested algebra $\mathbf{W}$. It uses ${}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{W}}\colon \tau \to (\tau \to \mathbf{U} + \mathbf{R}(\mathbf{W} + \tau)) \to \mathbf{W}$ with $\tau$ an abstract type of continuous functions. We assume that for every $f^{\tau}$ we have $\omega(f)\colon \mathbf{N} \to \mathbf{N}$ (the uniform modulus of continuity) and $h(f)\colon \mathbf{N} \to \mathbb{Q} \to \mathbb{Q}$ (the approximating function). Our example is the computational content of the proof of proposition (a) below, assigning to every continuous $f$ an ideal in $\mathbf{W}$. This ideal is given as ${}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{W}} f M$ with $M$ of type $\tau \to \mathbf{U} + \mathbf{R}(\mathbf{W} + \tau)$ defined by

$$Mf := \mathrm{Inr}(\Phi(\omega f 2) f(hf0)).$$

Here $\Phi\colon \mathbf{N} \to \tau \to (\mathbb{Q} \to \mathbb{Q}) \to \mathbf{R}(\mathbf{W} + \tau)$ is recursively defined by

$$\begin{aligned} \Phi 0 f g &:= R_d(\mathrm{Inr}(\mathrm{out}_d \circ f)) \quad \text{with } d := \mathrm{head}(g(\tfrac{1}{2^2})), \\ \Phi l f g &:= R(\Phi(l-1)(f \circ \mathrm{in}_d)(g \circ \mathrm{in}_d))_{d \in \{-1,0,1\}}, \end{aligned}$$

where $\mathrm{head}\colon \mathbb{Q} \to \mathbf{SD}$ is defined by

$$\mathrm{head}(q) := \begin{cases} -1 & \text{if } q < -\frac{1}{4} \\ 0 & \text{if } -\frac{1}{4} \le q \le \frac{1}{4} \\ 1 & \text{if } \frac{1}{4} < q. \end{cases}$$

$\Phi$ can be explicitly defined using $\mathcal{R}^{\tau \to (\mathbb{Q}\to\mathbb{Q}) \to \mathbf{R}(\mathbf{W}+\tau)}_{\mathbf{N}}$: as base term take

$$\lambda_{f,g} R_d(\mathrm{Inr}(\mathrm{out}_d \circ f)) \quad \text{with } d := \mathrm{head}(g(\tfrac{1}{2^2}))$$

and as step term

$$\lambda_{\_,p,f,g} R(p(f \circ \mathrm{in}_d)(g \circ \mathrm{in}_d))_{d \in \{-1,0,1\}}.$$

Simultaneous corecursion operators can be introduced similarly. For $\vec{\iota} := \mu_{\xi}(\kappa_0, \dots, \kappa_{k-1})$ let $k = \sum_{j<N} k_j$ with $k_j \ge 1$ and $m_j := \sum_{l<j} k_j$, hence $\mathrm{m}_j + k_j = m_{j+1}$. Recall the constructor type

$$(\rho_{i\nu}(\vec{\iota}))_{\nu<n_i} \to \iota_j \quad (m_j \le i < m_{j+1}).$$

The product of these $k$ constructor types is isomorphic to

$$\sum_{m_j \le i < m_{j+1}} \prod_{\nu<n_i} \rho_{i\nu}(\iota) \to \iota_j$$

and the type of the recursion operator $\mathcal{R}_j^{\vec{\iota},\vec{\tau}}$ is isomorphic to

$$\iota_j \to (\sum_{m_l \le i < m_{l+1}} \prod_{\nu<n_i} \rho_{i\nu}(\vec{\iota} \times \vec{\tau}) \to \tau_l)_{i<k} \to \tau_j.$$

Dually for the algebras $\vec{\iota}$ the types of the *destructor* $D_j^{\vec{\iota}}$ (defined below) for the algebra $\iota_j$ is

$$\iota_j \to \sum_{m_j \le i < m_{j+1}} \prod_{\nu<n_i} \rho_{i\nu}(\vec{\iota}).$$

The $j$-th simultaneous corecursion operator ${}^{\mathrm{co}}\mathcal{R}_j^{\vec{\iota},\vec{\tau}}$ is used to construct a mapping from $\tau_j$ to $\iota_j$ by "corecursion" on the structure of $\iota$. Its type is

$$\tau_j \to (\tau_l \to \sum_{m_l \le i < m_{l+1}} \prod_{\nu<n_i} \rho_{i\nu}(\vec{\iota} + \vec{\tau}))_{l<N} \to \iota_j.$$

We give an example of a simultaneous corecursion on tree lists and trees. Recall the simultaneously defined algebras $(\mathbf{Ts}(\alpha), \mathbf{T}(\alpha))$ (we write them without the parameter $\alpha$ here), whose constructors $\mathrm{C}_i^{(\mathbf{Ts},\mathbf{T})}$ for $i \in \{0,\dots,3\}$ are

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

The elements of the algebra $\mathbf{T}$ (i.e., $\mathbf{T}(\alpha)$) are just the finitely branching trees, which carry objects of type $\alpha$ on their leaves.

We compute the types of the corecursion operators w.r.t. the argument types $\sigma, \tau$, i.e., of ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{Ts}}^{(\mathbf{Ts},\mathbf{T}),(\sigma,\tau)}$ and ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{T}}^{(\mathbf{Ts},\mathbf{T}),(\sigma,\tau)}$, or shortly ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{Ts}}$ and ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{T}}$. The step types are

$$\delta_0 := \sigma \to \mathbf{U} + (\mathbf{T} + \tau) \times (\mathbf{Ts} + \sigma), \qquad \delta_1 := \tau \to \alpha + (\mathbf{Ts} + \sigma).$$

Hence the types of the corecursion operators are

$${}^{\mathrm{co}}\mathcal{R}_{\mathbf{Ts}} \colon \sigma \to \delta_0 \to \delta_1 \to \mathbf{Ts},$$
$${}^{\mathrm{co}}\mathcal{R}_{\mathbf{T}} \colon \tau \to \delta_0 \to \delta_1 \to \mathbf{T}.$$

The corecursion operator ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{T}}$ or explicitly ${}^{\mathrm{co}}\mathcal{R}_{\mathbf{T}}^{(\mathbf{Ts},\mathbf{T}),(\sigma,\tau)}$ is displayed as

$$\texttt{(CoRec } \tau \to \mathbf{T} \ \ \sigma \to \mathbf{Ts}\texttt{)},$$

where the first arrow type indicates the type of the corecursion and its argument type, and the remaining arrow types provide the argument types for the simultaneously defined algebras.

*Simplified simultaneous corecursion.* In a corecursion on simultaneously defined algebras one may need to recur on some of those algebras only. Then we can simplify the type of the corecursion operator accordingly, as follows.

(i) Only consider the relevant constructors, i.e., those mapping into relevant algebras.

(ii) Shorten their types by omitting all argument types containing irrelevant algebras.
(iii) Out of these shortened relevant constructor types form the dual type

$$\iota_j \to \sum_{m_j \le i < m_{j+1}} \prod \vec{\rho}(\vec{\iota})$$

with $\vec{\iota}$ the relevant algebras. The simplified step type then is

$$\tau_j \to \sum_{m_j \le i < m_{j+1}} \prod \vec{\rho}(\vec{\iota} + \vec{\tau})$$

with $\vec{\tau}$ corresponding to $\vec{\iota}$.

In the $(\mathbf{Ts}, \mathbf{T})$-example, if we want to do corecursion on $\mathbf{Ts}$ only, then there is a single step type

$$\delta_0 := \sigma \to \mathbf{U} + (\mathbf{Ts} + \sigma),$$

and the type of the simplified corecursion operator is

$${}^{\mathrm{co}}\mathcal{R}_{\mathbf{Ts}} \colon \mathbf{Ts} \to \delta_0 \to \sigma.$$

*Remark.* There is yet another situation where one might want to simplify the type of corecursion, namely when the argument type $\tau$ is the unit type $\mathbf{U}$. For instance for ${}^{\mathrm{co}}\mathcal{R}^{\tau}_{\mathbf{N}}$ its type

$$\tau \to (\tau \to \mathbf{U} + (\mathbf{N} + \tau)) \to \mathbf{N}$$

can then be simplified to

$$(\mathbf{U} + (\mathbf{N} + \mathbf{U})) \to \mathbf{N}.$$

4.4. **A common extension $\mathrm{T}^+$ of Gödel's T and Plotkin's PCF.** *Terms* of $\mathrm{T}^+$ are built from (typed) variables and (typed) constants (constructors C or defined constants $D$, see below) by (type-correct) application and abstraction:

$$M, N ::= x^\rho \mid \mathrm{C}^\rho \mid D^\rho \mid (\lambda_{x^\rho} M^\sigma)^{\rho\to\sigma} \mid (M^{\rho\to\sigma} N^\rho)^\sigma.$$

**Definition** (Computation rule)**.** Every defined constant $D$ comes with a system of *computation rules*, consisting of finitely many equations

$$D\vec{P}_i(\vec{y}_i) = M_i \qquad (i = 1, \dots, n) \tag{6}$$

with free variables of $\vec{P}_i(\vec{y}_i)$ and $M_i$ among $\vec{y}_i$, where the arguments on the left hand side must be "constructor patterns", i.e., lists of applicative terms built from constructors and distinct variables. To ensure consistency of the defining equations, we require that for $i \neq j$ either $\vec{P}_i$ and $\vec{P}_j$ are non-unifiable (i.e., there is no substitution which identifies them), or else $\vec{P}_i$ and $\vec{P}_j$ have disjoint free variables, and for the most general unifier $\xi$ of $\vec{P}_i$ and $\vec{P}_j$ we have $M_i\xi = M_j\xi$. Notice that the substitution $\xi$ assigns to the variables

$\vec{y}_i$ in $M_i$ constructor patterns $\vec{R}_k(\vec{z})$ $(k = i, j)$. A further requirement on a system of computation rules $D\vec{P}_i(\vec{y}_i) = M_i$ is that the lengths of all $\vec{P}_i(\vec{y}_i)$ are the same; this number is called the *arity* of $D$, denoted by $\mathrm{ar}(D)$. A substitution instance of a left hand side of (6) is called a *$D$-redex*.

More formally, constructor patterns are defined inductively by (we write $\vec{P}(\vec{x}\,)$ to indicate all variables in $\vec{P}$)

(a) $x$ is a constructor pattern.
(b) The empty list $\langle\rangle$ is a constructor pattern.
(c) If $\vec{P}(\vec{x}\,)$ and $Q(\vec{y}\,)$ are constructor patterns whose variables $\vec{x}$ and $\vec{y}$ are disjoint, then $(\vec{P}, Q)(\vec{x}, \vec{y}\,)$ is a constructor pattern.
(d) If C is a constructor and $\vec{P}$ a constructor pattern, then so is $\mathrm{C}\vec{P}$, provided it is of ground type.

*Remark.* The requirement of disjoint variables in unifiable constructor patterns $\vec{P}_i$ and $\vec{P}_j$ used in computation rules of a defined constant $D$ is needed to ensure that applying the most general unifier produces constructor patterns again. However, for readability we take this as an implicit convention, and write computation rules with possibly non-disjoint variables.

Examples of constants $D$ defined by computation rules are abundant. The defining equations in 4.2 can all be seen as computation rules, for

(i) the append-function $*$,
(ii) list reversal Rev,
(iii) the simultaneously defined functions $\mathrm{even}, \mathrm{odd}\colon \mathbf{N} \to \mathbf{B}$ and
(iv) the two simultaneously defined functions $\oplus\colon \mathbf{Ts} \to \mathbf{T} \to \mathbf{Ts}$ and $+\colon \mathbf{T} \to \mathbf{T} \to \mathbf{T}$.

Moreover, the structural recursion operators themselves can be viewed as defined by computation rules, which in this case are called *conversion* rules; cf. 4.2.

The boolean connectives andb, impb and orb are defined by

$$\begin{array}{lll} \mathtt{tt}\ \mathrm{andb}\ y = y, & \mathtt{ff}\ \mathrm{impb}\ y = \mathtt{tt}, & \mathtt{tt}\ \mathrm{orb}\ y = \mathtt{tt}, \\ x\ \mathrm{andb}\ \mathtt{tt} = x, & \mathtt{tt}\ \mathrm{impb}\ y = y, & x\ \mathrm{orb}\ \mathtt{tt} = \mathtt{tt}, \\ \mathtt{ff}\ \mathrm{andb}\ y = \mathtt{ff}, & x\ \mathrm{impb}\ \mathtt{tt} = \mathtt{tt}, & \mathtt{ff}\ \mathrm{orb}\ y = y, \\ x\ \mathrm{andb}\ \mathtt{ff} = \mathtt{ff}, & & x\ \mathrm{orb}\ \mathtt{ff} = x. \end{array}$$

Notice that when two such rules overlap, their right hand sides are equal under any unifier of the left hand sides.

Decidable *equality* $=_\iota\colon \iota \to \iota \to \mathbf{B}$ for a finitary algebra $\iota$ is defined by

$$(\mathrm{C}_i\vec{x} =_\iota \mathrm{C}_j\vec{y}) = \mathtt{ff} \quad \text{if } i \neq j,$$
$$(\mathrm{C}_i\vec{x} =_\iota \mathrm{C}_i\vec{y}) = (\vec{x}^P =_{\vec{\rho}} \vec{y}^P\ \mathrm{andb} \bigwedge_{\nu<n} (\vec{x}^R_{m+\nu} =_{\iota_{j_\nu}} \vec{y}^R_{m+\nu})).$$

(For a constructor term $\mathrm{C}\vec{r}$ we denote by $\vec{r}^{\,P}$ its parameter arguments and by $\vec{r}^{\,R}$ its recursive arguments.) For example,

$$(0 =_{\mathbf{N}} 0) = \mathtt{tt}, \qquad (\mathrm{S}m =_{\mathbf{N}} 0) = \mathtt{ff},$$
$$(0 =_{\mathbf{N}} \mathrm{S}n) = \mathtt{ff}, \qquad (\mathrm{S}m =_{\mathbf{N}} \mathrm{S}n) = (m =_{\mathbf{N}} n).$$

The *predecessor* functions introduced in 4.1 by means of the cases-operator $\mathcal{C}$ can also be viewed as defined constants:

$$\mathrm{P}0 = 0, \qquad \mathrm{P}(\mathrm{S}n) = n.$$

Another example is the *destructor* function, disassembling a constructor-built argument into its parts. For the type $\mathbf{T}_1 := \mu_\xi(\xi, (\mathbf{N} \to \xi) \to \xi)$ the destructor $D_{\mathbf{T}_1}$ has type

$$D_{\mathbf{T}_1} : \mathbf{T}_1 \to \mathbf{U} + (\mathbf{N} \to \mathbf{T}_1)$$

and is defined by the computation rules

$$D_{\mathbf{T}_1} 0 = \mathrm{Inl}(\mathbf{u}), \qquad D_{\mathbf{T}_1}(\mathrm{Sup}(f)) = \mathrm{Inr}(f).$$

Generally, the type of the destructor $D_\iota$ function for $\iota := \mu_\xi(\kappa_0, \ldots, \kappa_{k-1})$ with $\kappa_i = \vec{\rho}_i \to \iota$ is

$$\iota \to \sum_{i<k} \prod_{\nu<n_i} \rho_{i\nu}(\iota).$$

Its conversion rules map $D_\iota(\mathrm{C}_i\vec{x}\,)$ to the $i$-th injection into the sum type of the product of the $\vec{x}$.

4.5. **Implementation.** Every object constant has the internal representation

`(const` *object-or-arity name kind 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`. An examples for an object constant is

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

*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, its denoted objects (as association list with type substitutions as keys) and possibly (as an optinal final entry) the (Scheme) code of an *external* function mapping a type substitution and an object list to either an object to be returned immediately, or else to `#f`, in which case the rules are tried next. 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 "ExIntro" axiom. One uses

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

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

`ex-formula-and-concl-to-ex-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*),

From these we can define

(const-to-type *const*),

```
(const-to-tvars const).
```

The test functions are

```
(const-form? x),
(check-const x),
(const? x),
(const=? x y).
```

`check-const` assumes that the constant is not one of those used during proof normalization. Hence `repro-data` must be empty.

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 `(ExIntro 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-alg alg-name1 ...).
```

Recall that program constants allow user defined rules, and that 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`. We have procedures recovering information from the string denoting a program constant (via `PROGRAM-CONSTANTS`):

```
(pconst-name? string),
(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),
(pconst-name-to-external-code name).
```

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

```
(display-pconst name1 ...).
```

To add and remove program constants we use

```
(add-program-constant name type <rest>),
(remove-program-constant string1 ...);
```

*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).

The degree of totality of a program constant can be changed from 0 to 1 provided we have proved that the program constant is in fact total. This change is done by calling `change-t-deg-to-one` with the name of the program constant.

To make program constants more readable we provide

```
(add-prefix-display-string name1 name2),
(add-postfix-display-string name1 name2),
(add-infix-display-string name1 name2).
```

To add and remove computation and rewrite rules and also external code we have

```
(add-computation-rule lhs rhs),
(add-rewrite-rule lhs rhs),
(add-external-code name code),
(remove-computation-rules-for lhs),
(remove-rewrite-rules-for lhs),
(remove-external-code name).
```

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,
(alg-to-destr-const alg)                    destructor,
(ex-formula-and-concl-to-ex-elim-const
    ex-formula concl).
```

Corecursion will be treated below.

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

Similarly we have `arrow-type-to-cases-const` and on the proof level `all-formula-to-cases-const`. For elimination axioms we have

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

*General recursion and induction* (work of Simon Huber)

```
(GRecGuard rhos tau) :
(rhos=>nat)=>rhos=>(rhos=>(rhos=>tau)=>tau)=>boole=>tau
GRecGuard mu xs G True ->
   G xs([ys]GRecGuard mu ys G(mu ys<mu xs))
GRecGuard mu xs G False -> Inhab
```

For convenience we add GRec with

```
GRec mu xs G -> GRecGuard mu xs G True
```

There is also a variant with type parameters:

```
(GRecGuard m alphas rhos tau) :
alphas=>(rhos=>nat)=>rhos=>(rhos=>(rhos=>atomic=>tau)=>tau)=>
boole=>atomic=>tau
GRecGuard ts mu xs G True u ->
G xs ([ys,atomic]GRecGuard ts mu ys G (mu ys<mu xs) atomic)
GRecGuard ts mu xs G False u -> Efq u
```

Note that this variant is only used to normalize proofs. Here we need that Efq is a constant. Induction:

```
GInd : allnc zs all mu,xs(Prog_mu{xs|A(xs)} ->
         all boole(atom(boole) -> A(xs))), where
Prog_mu{xs|A(xs)} =
all xs(all ys(mu ys<mu xs -> A(ys)) -> A(xs))
```

We get the ordinary general induction `GInd'` by:

```
GInd' ts mu xs M = GInd ts mu xs M True Truth
```

Internally we have

`(type-info-to-grecguard-const type-info)`,

`(type-info-to-grec-const type-info)`

`all-formula-and-number-to-grecguard-const` is used to achieve normalization of proofs via translating them in terms, to translate a gind-aconst. In addition we need the number m of quantifiers used for the axioms.

*Corecursion.* To generate the corecursion constants we use

`(alg-or-arrow-types-to-corec-consts . arrow-types)`,

`(alg-or-arrow-types-to-corec-const . arrow-types)`.

To avoid being trapped in non-termination of the conversion rule for corecursion, we now aim at a bounded reduction of corec constants.

In `corec-const-and-bound-to-bcorec-term` we begin with constructing `corec-consts` (as in `corec-const-to-corec-consts` above), for the base case of the `bcorec-term`. The product of their types is the value type of the recursion operator (over **N**). Next the step-term

`lambda (n prev)(abstr-if-term1 pair .. pair abstr-if-termN)`

is built. We need variables `us` for the covals and `vs` for the steps. Each `(vi ui)` has type `ysum-without-unit-of-product-types`. For each product type we introduce a product-variable $y$. The components of the term corresponding to $y$ are called `param-comps` and `test-comps`. Using the instantiated constructors we can build the `abstr-constr-terms` for the constructors of the $i$-th algebra, and using these the $i$-th if-term is constructed via `corec-test-and-abstr-constr-terms-to-if-term`.

Finally `undelay-delayed-corec` takes a term and a non-negative integer (a bound) as arguments. It replaces every corecursion constant in the given term by the result of applying `corec-const-and-bound-to-bcorec-term` to it and the given bound.

## 5. Predicates

Every predicate has an arity (i.e., a list of types) and denotes a property of tuples of functionals of these types. We have the following three kinds of predicates:

(i) predicate variables;
(ii) predicate constants;
(iii) inductively and coinductively defined predicate constants.

`(predicate-to-arity predicate)` returns the arity of a predicate. A test for equality is `(predicate-equal? pred1 pred2)`.

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 $\{ x_1, \dots, x_n \mid A \}$. Totality matters for the abstracted variables of a comprehension term, because of the inductively defined existential quantifier. The default is the use of partial variables.

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 [14] 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*). | |

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* it 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 want axiomatized predicates, which are *not* placeholders for formulas. The main example is the totality predicate constant, intended to denote the set of total objects of a given type. We will see below (in section 5.3) that in case this type is (i) an algebra we can define the totality predicate inductively, and (ii) an arrow or a pair type we can define it explicitly. However, we also allow type variables $\alpha$ (and substitutions for them), and certainly cannot know what property the "total" elements of type $\alpha$ should have. Therefore we provide a totality predicate constant $T_\rho$ of arity $(\rho)$ at an arbitrary type $\rho$; this is necessary for to allow a type substitution $\alpha \mapsto \rho$ in formulas involving $T_\alpha$. However, a formula $T_\rho r$ can be "unfolded" in case $\rho$ is an algebra, an arrow or a pair type: $T_\iota r$ unfolds by means of the inductively defined totality predicate for the algebra $\iota$, and

$$T_{\rho\to\sigma} r := \forall^{\mathrm{nc}}_{\hat{x}}(T_\rho \hat{x} \to T_\sigma(r\hat{x})),$$
$$T_{\rho\times\sigma} r := T_\rho r_0 \wedge T_\sigma r_1.$$

This unfolding is done by means of `(unfold-formula formula)` (which also unfolds classical existential quantifiers). We also provide

`(term-to-totality-formula term)`,

which when applied to a term $r$ of type $\rho$ returns the result of unfolding $T_\rho r$.

The inductively defined totality predicate for an algebra $\iota$ is computationally relevant (c.r.) and has its witnesses in the same algebra $\iota$. Therefore it is mandatory to consider $T_\rho$ as c.r. as well, and let $\rho$ be the type of its witnesses.

When later (in section 13) we consider realizability it will be necessary to define what $t\ \mathbf{r}\ T_\rho s$ means. Since again $T_\alpha$ is unknown we provide another predicate constant $T^{\mathbf{r}}_\rho$ and define $t\ \mathbf{r}\ T_\rho s := T^{\mathbf{r}}_\rho ts$. Clearly $T^{\mathbf{r}}_\rho$ has arity $(\rho,\rho)$ and is computationally irrelevant. Again $T^{\mathbf{r}}_\rho ts$ unfolds by means of an inductively defined predicate for the algebra $\iota$, and

$$T^{\mathbf{r}}_{\rho\to\sigma} ts := \forall^{\mathrm{nc}}_{\hat{x},\hat{y}}(T^{\mathbf{r}}_\rho \hat{x}\hat{y} \to T^{\mathbf{r}}_\sigma(t\hat{x}, s\hat{y})),$$
$$T^{\mathbf{r}}_{\rho\times\sigma} ts := T^{\mathbf{r}}_\rho t_0 s_0 \wedge T^{\mathbf{r}}_\sigma t_1 s_1.$$

This unfolding is done by calling

```
(terms-to-mr-totality-formula term1 term2),
```

which when applied to terms $t$ and $s$ of type $\rho$ returns the result of unfolding $T^{\mathbf{r}}_{\rho} ts$.

It is also possible to add (and later remove) further computationally irrelevant predicate constants via

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

We have 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 provide

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

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 $P$.

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 or coinductively. However, for simplicity we omit the treatment of coinductive definitions and deal with inductive definitions only. For example, in the algebra $\mathbf{N}$ we can inductively define *totality* by the clauses

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

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

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

The reason for writing it in this way is that it fits more conveniently with the logical elimination rules, which will be useful in the proof of the soundness theorem. It expresses that every "competitor" $\{\, n \mid A(n) \,\}$ satisfying the same clauses contains $T_{\mathbf{N}}$. This is the usual induction schema for natural numbers, which clearly only holds for "total" numbers (i.e., total ideals in the information system for $\mathbf{N}$). Notice that we have used a "strengthened" form of the "step formula", namely $\forall_n(T_{\mathbf{N}}n \to A(n) \to A(\mathrm{S}n))$ 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 $T_{\mathbf{N}}(n)$ available. Totality for an arbitrary algebra can be defined similarly. Consider for example the non-finitary algebra $\mathbf{O}$ (cf. 2.3), with constructors 0, successor S of type $\mathbf{O} \to \mathbf{O}$ and supremum Sup of type $(\mathbf{N} \to \mathbf{O}) \to \mathbf{O}$. Its clauses are

$$T_{\mathbf{O}}0, \qquad \forall_x(T_{\mathbf{O}}x \to T_{\mathbf{O}}(\mathrm{S}x)), \qquad \forall_f(\forall_{n\in T_{\mathbf{N}}} T_{\mathbf{O}}(fn) \to T_{\mathbf{O}}(\mathrm{Sup}(f))),$$

and its least-fixed-point scheme is

$$\begin{aligned}\forall_x(T_{\mathbf{O}}x \to A(0) \to\ & \\ & \forall_x(T_{\mathbf{O}}x \to A(x) \to A(\mathrm{S}x)) \to \\ & \forall_f(\forall_{n\in T} T_{\mathbf{O}}(fn) \to \forall_{n\in T} A(fn) \to A(\mathrm{Sup}(f))) \to \\ & A(x)).\end{aligned}$$

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

$$K_i := \forall_{\vec{x}}((A_\nu(I))_{\nu<n} \to I\vec{r}) \quad (i < k).$$

It is not required that all universal quantifiers precede all implications.

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". Every predicate has an *arity*, which is a possibly empty list of types.

**Definition** (Formulas and predicates)**.** By simultaneous induction we define formula forms

$$A, B ::= P\vec{r} \mid A \to B \mid \forall_x A$$

and predicate forms

$$P, Q ::= X \mid \{\, \vec{x} \mid A \,\} \mid \mu_X(\forall_{\vec{x}_i}((A_{i\nu})_{\nu<n_i} \to X\vec{r}_i))_{i<k}$$

with $X$ a predicate variable, $k \geq 1$ and $\vec{x}_i$ all free variables in $(A_{i\nu})_{\nu<n_i} \to X\vec{r}_i$ (it is not necessary to allow object parameters in inductively defined predicates, since they can be taken as extra arguments). Let $C$ denote both formula and predicate forms. Let $\mathrm{FPV}(C)$ denote the set of free predicate

variables in $C$. We define $\mathrm{SP}(Y, C)$ "$Y$ occurs at most strictly positive in $C$" by induction on $C$.

$$\frac{\mathrm{SP}(Y,P)}{\mathrm{SP}(Y,P\vec{r}\,)} \qquad \frac{Y \notin \mathrm{FPV}(A) \quad \mathrm{SP}(Y,B)}{\mathrm{SP}(Y,A\to B)} \qquad \frac{\mathrm{SP}(Y,A)}{\mathrm{SP}(Y,\forall_x A)}$$

$$\mathrm{SP}(Y,X) \qquad \frac{\mathrm{SP}(Y,A)}{\mathrm{SP}(Y,\{\,\vec{x}\mid A\,\})} \qquad \frac{\mathrm{SP}(Y,A_{i\nu}) \text{ for all } i{<}k,\ \nu{<}n_i}{\mathrm{SP}(Y,\mu_X(\forall_{\vec{x}_i}((A_{i\nu})_{\nu<n_i}\to X\vec{r}_i))_{i<k})}$$

Now we can define $\mathrm{F}(A)$ "$A$ is a formula" and $\mathrm{Preds}(P)$ "$P$ is a predicate", again by simultaneous induction.

$$\frac{\mathrm{Preds}(P)}{\mathrm{F}(P\vec{r}\,)} \qquad \frac{\mathrm{F}(A) \quad \mathrm{F}(B)}{\mathrm{F}(A\to B)} \qquad \frac{\mathrm{F}(A)}{\mathrm{F}(\forall_x A)}$$

$$\mathrm{Preds}(X) \qquad \frac{\mathrm{F}(A)}{\mathrm{Preds}(\{\,\vec{x}\mid A\,\})}$$

$$\frac{\mathrm{F}(A_{i\nu}) \text{ and } \mathrm{SP}(X,A_{i\nu}) \text{ for all } i{<}k,\ \nu{<}n_i \qquad X\notin \mathrm{FPV}(A_{0\nu}) \text{ for all } \nu{<}n_0}{\mathrm{Preds}(\mu_X(\forall_{\vec{x}_i}((A_{i\nu})_{\nu<n_i}\to X\vec{r}_i))_{i<k})}$$

We call

$$I := \mu_X(\forall_{\vec{x}_i}((A_{i\nu})_{\nu<n_i}\to X\vec{r}_i))_{i<k}$$

an inductive (or inductively defined) predicate. Sometimes it is helpful to display the predicate parameters and write $I(\vec{Y},\vec{Z}\,)$, where $\vec{Y},\vec{Z}$ are all predicate variables free in some $A_{i\nu}$ except $X$, and $\vec{Y}$ are the ones occuring only strictly positive. If we write the $i$-th component of $I$ in the form $\forall_{\vec{x}}((A_\nu(X))_{\nu<n}\to X\vec{r}\,)$, then we call

$$K_i := \forall_{\vec{x}}((A_\nu(I))_{\nu<n}\to I\vec{r}\,) \tag{7}$$

the $i$-th *clause* (or *introduction axiom*) of $I$, denoted $I_i^+$.

Here $\vec{A}\to B$ means $A_0\to\cdots\to A_{n-1}\to B$, associated to the right. The terms $\vec{r}$ are those introduced in section 6, i.e., typed terms built from variables and constants by abstraction and application, and (importantly) those with a common reduct are identified. In $\forall_{\vec{x}}((A_\nu(X))_{\nu<n}\to X\vec{r}\,)$ we call $A_\nu(X)$ a *parameter* premise if $X$ does not occur in it, and a *recursive* premise otherwise. A recursive premise $A_\nu(X)$ is *nested* if it has an occurrence of $X$ in a strictly positive parameter position of another (previously defined) inductive predicate, and unnested otherwise. An inductive predicate $I$ is called *nested* if it has a clause with at least one nested recursive premise, and *unnested* otherwise.

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 $C(\vec{r}\,)$. An inductively defined predicate is *finitary* if its clauses have recursive premises of the form $X\vec{s}$ only.

**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 the introduction axiom (7). Moreover, we have an *elimination axiom* $I^-$:

$$\forall_{\vec{x}}(I\vec{x} \to (\forall_{\vec{x}_i}((A_{i\nu}(I \cap X))_{\nu<n_i} \to X\vec{r}_i))_{i<k} \to X\vec{x}\,) \tag{8}$$

where $I \cap X$ abbreviates $\{\, \vec{x} \mid I\vec{x} \wedge X\vec{x}\,\}$ with $\wedge$ defined (inductively) below. Here $X$ can be thought of as a "competitor" predicate.

5.4. **Examples of inductive predicates.** 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 EqD 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 section 6 as a boolean-valued binary function.

*Leibniz equality.* We define Leibniz equality by

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

The introduction axiom is

$$\forall_x(x^\rho \equiv x^\rho)$$

and the elimination axiom

$$\forall_{x,y}(x \equiv y \to \forall_x Xxx \to Xxy),$$

where $x \equiv y$ abbreviates $\mathrm{EqD}(\rho)(x^\rho, y^\rho)$. In Minlog this is displayed as `x eqd y`.

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

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

Using compatibility of EqD one easily proves symmetry and transitivity. Define *falsity* by $\mathbf{F} := (\mathtt{ff} \equiv \mathtt{tt})$. 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 $\mathbf{F} \to x^\rho \equiv y^\rho$. To see this, we first obtain $\mathcal{R}^\rho_{\mathbf{B}}\mathtt{ff}xy \equiv \mathcal{R}^\rho_{\mathbf{B}}\mathtt{ff}xy$ from the introduction axiom. Then from $\mathtt{ff} \equiv \mathtt{tt}$ we get $\mathcal{R}^\rho_{\mathbf{B}}\mathtt{tt}xy \equiv \mathcal{R}^\rho_{\mathbf{B}}\mathtt{ff}xy$ by compatibility. Now $\mathcal{R}^\rho_{\mathbf{B}}\mathtt{tt}xy$ converts to $x$ and $\mathcal{R}^\rho_{\mathbf{B}}\mathtt{ff}xy$ converts to $y$. Hence $x^\rho \equiv y^\rho$, since we identify terms with a common reduct.

The claim can now be proved by induction on $A \in \mathrm{F}$. *Case* $I\vec{r}$. By definition the clause $K_0$ is "nullary", i.e., of the form $\forall_{\vec{x}}((A_\nu)_{\nu<n} \to I\vec{s})$ with no occurrence of $I$ in the $A_\nu$. By induction hypothesis from $\mathbf{F}$ we can derive all premises $A_\nu$. Hence $I\vec{s}$. From $\mathbf{F}$ we also obtain $r_i \equiv s_i$, by the remark above. Hence $I\vec{r}$ by compatibility. The cases $A \to B$ and $\forall_x A$ are obvious. □

A crucial use of Leibniz equality is that it allows to lift a boolean term $r^{\mathbf{B}}$ to a formula, by considering $r^{\mathbf{B}} \equiv \mathtt{tt}$ instead. For convenience we introduce a new predicate constant atom of arity $(\mathbf{B})$ and define $\mathrm{atom}(r^{\mathbf{B}})$ as an abbreviation of $r^{\mathbf{B}} \equiv \mathtt{tt}$. Formally, we use the axioms

$$\mathtt{AtomToEqDTrue}\colon \forall_{p^{\mathbf{B}}}(\mathrm{atom}(p^{\mathbf{B}}) \to p^{\mathbf{B}} \equiv \mathtt{tt}),$$
$$\mathtt{EqDTrueToAtom}\colon \forall_{p^{\mathbf{B}}}(p^{\mathbf{B}} \equiv \mathtt{tt} \to \mathrm{atom}(p^{\mathbf{B}})).$$

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 $(\mathrm{S}r =_{\mathbf{N}} \mathrm{S}s) \equiv \mathtt{tt}$, 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 $\mathrm{C}_i$ of an algebra $\iota$ we have an introduction axiom

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

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

$$\forall_{n_1,n_2}(n_1 =_{\mathbf{N}} n_2 \to X00 \to \forall_{n_1,n_2}(n_1 =_{\mathbf{N}} n_2 \to Xn_1n_2 \to X(\mathrm{S}n_1, \mathrm{S}n_2)) \to Xn_1n_2).$$

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)**.** $\forall_{x_1,x_2}(x_1 =_\rho x_2 \leftrightarrow x_1 \equiv x_2)$.

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 [22].

*Existential quantifier.*

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

("D" indicates that the existential quantifier is inductively defined; it also reminds on "double", since both parts – the variable $x$ and the kernel $A$ – are of computational significance. Later when considering decorations we will define other computational variants of the existential quantifier).

The introduction axiom is

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

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

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

*Conjunction.* We define

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

The introduction axiom is

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

where $A \wedge B$ (displayed `A andd B`) abbreviates $\mathrm{AndD}(\{\mid A\,\}, \{\mid B\,\})$, and the elimination axiom is

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

*Remark.* In addition to the inductively defined existential quantifier and conjunction, in Minlog there are also "primitive" variants, displayed `ex x A` and `A & B`. Both make use of a (again "primitive") version of the product type (displayed `rho@@sigma`), which is based on the pairing operation of the underlying progamming language (Scheme). The reason to have them is

that sometimes this allows a more efficient evaluation (i.e., normalization) of extracted terms.

*Disjunction.* We define

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

The introduction axioms are

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

where $A \vee B$ (displayed `A ord B`) abbreviates $\mathrm{OrD}(\{ \mid A \}, \{ \mid B \})$, and the elimination axiom is

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

*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

$$\forall_n(\mathrm{Even}(n) \to X0 \to \forall_n(\mathrm{Even}(n) \to Xn \to X(\mathrm{S}(\mathrm{S}n))) \to Xn).$$

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

$$\begin{aligned}&\forall_x \mathrm{TC}(x, x)),\\ &\forall_{x,y,z}(y \prec z \to \mathrm{TC}(x, y) \to \mathrm{TC}(x, z))\end{aligned}$$

and the elimination axiom is

$$\begin{aligned}\forall_{x,y}(\mathrm{TC}(x, y) \to \forall_x Xxx \to\\ \forall_{x,y,z}(y \prec z \to \mathrm{TC}(x, y) \to Xyz \to Xxz) \to\\ Xxy).\end{aligned}$$

*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

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

5.5. **Totality and induction.** We now inductively define general totality predicates. Let us first look at some examples. The clauses defining totality for the algebra $\mathbf{N}$ are

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

The least-fixed-point axiom is is according to (8)

$$\forall_n(T_{\mathbf{N}}n \to X0 \to \forall_n((T_{\mathbf{N}} \wedge X)n \to X(\mathrm{S}n)) \to Xn).$$

Written differently (with "duplication") we obtain

$$\forall_n(T_{\mathbf{N}}n \to X0 \to \forall_n(T_{\mathbf{N}}n \to Xn \to X(\mathrm{S}n)) \to Xn).$$

We call this least-fixed-point axiom an *induction* axiom, and write $\mathrm{Ind}^{n,X}_{\mathbf{N}}$ or $\mathrm{Ind}_{n,X}$ for $T^-_{\mathbf{N}}$. The indices $n, X$ are omitted when they can be inferred from the context. Clearly the partial continuous functionals with $T_{\mathbf{N}}$ interpreted as the total ideals for $\mathbf{N}$ provide a model of TCF extended by these axioms.

For the algebra $\mathbf{D}$ of derivations totality is inductively defined by the clauses

$$T_{\mathbf{D}}0^{\mathbf{D}}, \qquad \forall_x(T_{\mathbf{D}}x \to \forall_y(T_{\mathbf{D}}y \to T_{\mathbf{D}}(\mathrm{C}^{\mathbf{D}\to\mathbf{D}\to\mathbf{D}}xy))),$$

with least-fixed-point axiom

$$\forall_x(T_{\mathbf{D}}x \to X0^{\mathbf{D}} \to \forall_x(T_{\mathbf{D}}x \to Xx \to \forall_y(T_{\mathbf{D}}y \to Xy \to X(\mathrm{C}^{\mathbf{D}\to\mathbf{D}\to\mathbf{D}}xy))) \to Xx).$$

Again, the partial continuous functionals with $T_{\mathbf{D}}$ interpreted as the total ideals for $\mathbf{D}$ (i.e., the finite derivations) provide a model.

Generally we define $\mathrm{RT}_\rho$ called *relative totality*, and its special case $T_\rho$ called (absolute) *totality*. The definition of $\mathrm{RT}_\rho$ is relative to an assigment of predicate variables $Y$ of arity $(\alpha)$ to type variables $\alpha$.

**Definition** (Relative totality RT)**.** Let $\iota = \mu_\xi(\kappa_0, \dots, \kappa_{k-1}) \in \mathrm{Alg}(\vec{\alpha}\,)$ with $\kappa_i = (\rho_\nu(\vec{\alpha}, \xi))_{\nu<n} \to \xi$. Then $\mathrm{RT}_\iota := \mu_X(K_0, \dots, K_{k-1})$, with

$$K_i := \forall_{\vec{x}}((\mathrm{RT}_{\rho_\nu}(\vec{Y}, X)x_\nu)_{\nu<n} \to X(\mathrm{C}_i\vec{x}\,))$$

and

$$\begin{aligned}\mathrm{RT}_{\alpha_j}(\vec{Y}, X) &:= Y_j,\\ \mathrm{RT}_\xi(\vec{Y}, X) &:= X,\\ \mathrm{RT}_{\sigma\to\rho}(\vec{Y}, X) &:= \{\, f \mid \forall_{\vec{x}}(\mathrm{RT}_\sigma\vec{x} \to \mathrm{RT}_\rho(\vec{Y}, X)(f\vec{x}\,))\,\}.\end{aligned}$$

As an example of a finitary algebra with parameters consider lists $\mathbf{L}(\alpha)$. The clauses for the predicate $\mathrm{RT}_{\mathbf{L}(\alpha)}(Y)$ expressing relative totality w.r.t. the predicate variable $Y$ are

$$\mathrm{RT}_{\mathbf{L}(\alpha)}(Y)(\mathrm{Nil}), \qquad \forall_x(Yx \to \forall_l(\mathrm{RT}_{\mathbf{L}(\alpha)}(Y)l \to \mathrm{RT}_{\mathbf{L}(\alpha)}(Y)(x :: l))),$$

and the least-fixed-point axiom is

$$\begin{aligned}\forall_l(\mathrm{RT}_{\mathbf{L}(\alpha)}(Y)l \to{}& X(\mathrm{Nil}) \to\\ &\forall_x(Yx \to \forall_l(\mathrm{RT}_{\mathbf{L}(\alpha)}(Y)l \to Xl \to X(x :: l))) \to\\ &Xl^{\mathbf{L}(\alpha)}).\end{aligned}$$

For important special cases of the parameter predicates $\vec{Y}$ we introduce a separate notation. Suppose we want to argue about total ideals only. Note that this only makes sense when when no type variables occur. However, to allow a certain amount of abstract reasing (involving type variables to be substituted later by concrete closed types), we introduce special predicate variables $T_\alpha$ which under a substitution $\alpha \mapsto \rho$ with $\rho$ closed turn into the inductively defined predicate $T_\rho$. Using this convention we define totality for an arbitrary algebra by specializing $Y$ of arity $(\rho)$ to $T_\rho$.

**Definition** (Absolute totality $T$)**.** Let $\iota = \mu_\xi(\kappa_0, \dots, \kappa_{k-1}) \in \mathrm{Alg}(\vec{\alpha}\,)$ with $\kappa_i = (\rho_\nu(\vec{\alpha}, \xi))_{\nu<n} \to \xi$. Then $T_\iota := \mu_X(K_0, \dots, K_{k-1})$, with

$$K_i := \forall_{\vec{x}}((T_{\rho_\nu}(X)x_\nu)_{\nu<n} \to X(\mathrm{C}_i\vec{x}\,))$$

and

$$\begin{aligned}T_{\alpha_j}(X) &:= T_{\alpha_j},\\ T_\xi(X) &:= X,\\ T_{\sigma\to\rho}(X) &:= \{\, f \mid \forall_{\vec{x}}(T_\sigma\vec{x} \to T_\rho(X)(f\vec{x}\,))\,\}.\end{aligned}$$

Another important special case occurs when we substitute the predicate variables $Y$ by truth predicates $\{\, y \mid \top\,\}$. The resulting totality predicate is called *structural totality*. For example, the clauses for the predicate

$\mathrm{RT}_{\mathbf{L}(\alpha)}(\{\, y \mid \top \}) =: \mathrm{ST}_{\mathbf{L}(\alpha)}$ expressing structural totality are

$$\mathrm{ST}_{\mathbf{L}(\alpha)}(\mathrm{Nil}), \qquad \forall_x(\{\, y \mid \top \}x \to \forall_l(\mathrm{ST}_{\mathbf{L}(\alpha)}l \to \mathrm{ST}_{\mathbf{L}(\alpha)}(x :: l))),$$

and the least-fixed-point axiom is

$$\begin{aligned}\forall_l(\mathrm{ST}_{\mathbf{L}(\alpha)}l \to & X(\mathrm{Nil}) \to \\ & \forall_x(\{\, y \mid \top \}x \to \forall_l(\mathrm{ST}_{\mathbf{L}(\alpha)}l \to Xl \to X(x :: l))) \to \\ & Xl^{\mathbf{L}(\alpha)}).\end{aligned}$$

Here the premises $\{\, y \mid \top \}x$ can clearly be omitted, and the least-fixed-point turns into

$$\forall_l(\mathrm{ST}_{\mathbf{L}(\alpha)}l \to X(\mathrm{Nil}) \to \forall_{x,l}(\mathrm{ST}_{\mathbf{L}(\alpha)}l \to Xl \to X(x :: l)) \to Xl^{\mathbf{L}(\alpha)}),$$

called *structural induction* on lists.

Note that we allow usage of totality predicates for previously introduced algebras $\iota'$. An example is totality $T_{\mathbf{T}}$ for the algebra $\mathbf{T}$ of finitely branching trees. It is defined by the single clause

$$\forall^{\mathrm{nc}}_{as}(\mathrm{RT}_{\mathbf{L}(\mathbf{T})}(T_{\mathbf{T}})(as) \to^{\mathrm{c}} T_{\mathbf{T}}(\mathrm{Branch}(as))).$$

In practice one often wants to reason about total objects only. To make this more convenient, Minlog distinguishes between *general* variables (written `x^`) and *total* variables (written `x`, without a hat). The latter are (implicitly) restricted to the relative totality predicate of the respective type. Formally, these conventions appear as abbreviating axioms

$$\begin{aligned}&\forall_x Px \to \forall_{\hat{x}}(T_\rho \hat{x} \to P\hat{x}) \quad \texttt{AllTotalElim},\\ &\forall_{\hat{x}}(T_\rho \hat{x} \to P\hat{x}) \to \forall_x Px \quad \texttt{AllTotalIntro}\end{aligned}$$

where $T_\rho$ is the absolute totality predicate defined above, which depends on the type $\rho$ of $x$. For instance, $T_{\mathbf{L}(\mathbf{N})}$ is $\mathrm{RT}_{\mathbf{L}(\mathbf{N})}(T_{\mathbf{N}})$, and $T_{\mathbf{L}(\alpha)}$ is $\mathrm{RT}_{\mathbf{L}(\alpha)}(T_\alpha)$.

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" $\mu$. 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 $\omega$, so the ordering we refer to is just the standard $<$-relation on the natural numbers. The principle of general induction then is

$$\forall_{\mu,x\in T}(\mathrm{Prog}^\mu_x Px \to Px) \tag{9}$$

where $\mathrm{Prog}^\mu_x Px$ expresses "progressiveness" w.r.t. the measure function $\mu$ 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* (9) from ordinary induction. However, it will be convenient to use general induction as a primitive axiom.

5.6. **Coinductive definitions.** We now extend TCF by allowing coinductive definitions as well as inductive ones. For instance, in the algebra $\mathbf{N}$ we can coinductively define *cototality* by the clause

$$^{\mathrm{co}}T_{\mathbf{N}}n \to n \equiv 0 \vee \exists_m(^{\mathrm{co}}T_{\mathbf{N}}m \wedge n \equiv \mathrm{S}m).$$

Its greatest-fixed-point axiom is

$$Xn \to \forall_n(Xn \to n \equiv 0 \vee \exists_m((^{\mathrm{co}}T_{\mathbf{N}}m \vee Xm) \wedge n \equiv \mathrm{S}m) \to {}^{\mathrm{co}}T_{\mathbf{N}}n.$$

It expresses that every "competitor" $X$ satisfying the same clause is a subset of $^{\mathrm{co}}T_{\mathbf{N}}$. The partial continuous functionals with $^{\mathrm{co}}T_{\mathbf{N}}$ interpreted as the cototal ideals for $\mathbf{N}$ provide a model of TCF extended by these axioms. The greatest-fixed-point axiom is called the *coinduction* axiom for natural numbers.

Similarly, for the algebra $\mathbf{D}$ of derivations with constructors $0^{\mathbf{D}}$ and $\mathrm{C}^{\mathbf{D}\to\mathbf{D}\to\mathbf{D}}$ cototality is coinductively defined by the clause

$$^{\mathrm{co}}T_{\mathbf{D}}x \to x \equiv 0 \vee \exists_y(^{\mathrm{co}}T_{\mathbf{D}}y \wedge \exists_z(^{\mathrm{co}}T_{\mathbf{D}}z \wedge x \equiv \mathrm{C}yz)).$$

Its greatest-fixed-point axiom is

$$\begin{aligned} Xx \to \forall_x(Xx \to x \equiv 0 \vee \exists_y((^{\mathrm{co}}T_{\mathbf{D}}x \vee Xy) \wedge \\ \exists_z((^{\mathrm{co}}T_{\mathbf{D}}x \vee Xz) \wedge x \equiv \mathrm{C}yz))) \to {}^{\mathrm{co}}T_{\mathbf{D}}x. \end{aligned}$$

The partial continuous functionals with $^{\mathrm{co}}T_{\mathbf{D}}$ interpreted as the cototal ideals for $\mathbf{D}$ (i.e., the finite or infinite locally correct derivations) provide a model.

For the algebra $\mathbf{I}$ of standard rational intervals cototality is defined by

$$\begin{aligned} ^{\mathrm{co}}T_{\mathbf{I}}x \to x \equiv \mathbb{I} \vee \exists_y(^{\mathrm{co}}T_{\mathbf{I}}y \wedge x \equiv \mathrm{C}_{-1}y) \vee \\ \exists_y(^{\mathrm{co}}T_{\mathbf{I}}y \wedge x \equiv \mathrm{C}_0y) \vee \\ \exists_y(^{\mathrm{co}}T_{\mathbf{I}}y \wedge x \equiv \mathrm{C}_1y). \end{aligned}$$

A model is provided by the set of all finite or infinite streams of signed digits from $\{-1, 0, 1\}$, i.e., the well-known (non-unique) stream representation of real numbers.

Generally, every inductive predicate $I$ gives rise to a coinductive predicate, its *dual* or *companion* $^{\mathrm{co}}I$. Let $I$ be inductively defined by the clauses

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

The conjunction of these $k$ clauses is equivalent to

$$\forall_{\vec{x}}(\bigvee_{i<k} \exists_{\vec{x}_i}((\bigwedge_{\nu<n_i} A_{i\nu}(I) \wedge \vec{x} \equiv \vec{t}_i) \to I\vec{x}).$$

Now the dual ${}^{\mathrm{co}}I$ of $I$ is coinductively defined by its closure axiom ${}^{\mathrm{co}}I^-$:

$$\forall_{\vec{x}}({}^{\mathrm{co}}I\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i}(\bigwedge_{\nu<n_i} A_{i\nu}({}^{\mathrm{co}}I) \land \vec{x} \equiv \vec{t}_i)).$$

Its greatest-fixed-point axiom ${}^{\mathrm{co}}I^+$ is

$$\forall_{\vec{x}}(X\vec{x} \to \forall_{\vec{x}}(X\vec{x} \to \bigvee_{i<k} \exists_{\vec{x}_i}(\bigwedge_{\nu<n_i} A_{i\nu}({}^{\mathrm{co}}I \cup X) \land \vec{x} \equiv \vec{t}_i)) \to {}^{\mathrm{co}}I\vec{x}\,)$$

where ${}^{\mathrm{co}}I \cup X$ abbreviates $\{\,\vec{x} \mid {}^{\mathrm{co}}I\vec{x} \lor X\vec{x}\,\}$.

Notice that the proof of the Ex-Falso-Quodlibet theorem above can easily be extended by a case ${}^{\mathrm{co}}I\vec{r}$: use the greatest-fixed-point axiom for ${}^{\mathrm{co}}I$ with $X\vec{x} := \mathbf{F}$. Since we have a nullary clause $\forall_{\vec{x}}((A_\nu)_{\nu<n} \to I\vec{s}\,)$ with no occurrence of $I$ in the $A_\nu$, it suffices to prove $\mathbf{F} \to \exists_{\vec{y}_i} \bigwedge_{\nu<n} \vec{A}_\nu$. But this follows from the induction hypothesis.

We extend this to the simultaneous case. For $\vec{I} := \mu_{\vec{X}}(K_0, \dots, K_{k-1})$ let $k = \sum_{j<N} k_j$ with $k_j \geq 1$ and $m_j := \sum_{l<j} k_j$, hence $\mathrm{m}_j + k_j = m_{j+1}$. Recall the clauses or introduction axioms $I_i^+$:

$$\forall_{\vec{x}_i}((A_{i\nu}(\vec{I}))_{\nu<n_i} \to I_j\vec{t}_i) \quad (m_j \leq i < m_{j+1}).$$

The conjunction of these $k_j$ clauses is equivalent to

$$\forall_{\vec{x}}(\bigvee_{m_j \leq i < m_{j+1}} \exists_{\vec{x}_i}(\bigwedge_{\nu<n_i} A_{i\nu}(\vec{I}) \land \vec{x} \equiv \vec{t}_i) \to I_j\vec{x}\,).$$

The dual ${}^{\mathrm{co}}I_j$ of $I_j$ is coinductively defined by its closure axiom ${}^{\mathrm{co}}I_j^-$:

$$\forall_{\vec{x}}({}^{\mathrm{co}}I_j\vec{x} \to \bigvee_{m_j \leq i < m_{j+1}} \exists_{\vec{x}_i}(\bigwedge_{\nu<n_i} A_{i\nu}(\vec{{}^{\mathrm{co}}I}) \land \vec{x} \equiv \vec{t}_i)).$$

Its greatest-fixed-point axiom ${}^{\mathrm{co}}I_j^+$ is

$$\forall_{\vec{x}}(X_j\vec{x} \to (\forall_{\vec{x}}(X_j\vec{x} \to \bigvee_{m_j \leq i < m_{j+1}} \exists_{\vec{x}_i}(\bigwedge_{\nu<n_i} A_{i\nu}(\vec{{}^{\mathrm{co}}I} \lor \vec{X}) \land \vec{x} \equiv \vec{t}_i)))_{j<N} \to {}^{\mathrm{co}}I_j\vec{x}\,).$$

The most important coinductively defined predicates for us will be those of cototality; we have seen some examples above. Generally, for a finitary algebra $\iota$ cototality is coinductively defined by

$${}^{\mathrm{co}}T_\iota x \to \bigvee_{i<k} \exists_{\vec{y}_i}({}^{\mathrm{co}}T_\iota\vec{y}_i \land x \equiv \mathrm{C}_i\vec{y}_i).$$

5.7. **Implementation.** We maintain an association list `IDS` (a global variable), which assigns all relevant information to the name of an inductively defined predicate constant. This information consists of

(i) the names of idpredconsts simultaneously defined with the present one,
(ii) an algebra name (for computational content, in case there is one),

(iii) the clauses with their names.

These data can be read off by

```
(idpredconst-name-to-simidpc-names name),
(idpredconst-name-to-alg-name name),
(idpredconst-name-to-clauses name).
```

Every inductively defined predicate constant has the internal representation

```
(idpredconst name types cterms).
```

*types* and *cterms* are to be substituted for the type and predicate variables in the clauses. To create this substitution use `idpredconst-to-tpsubst`.

We provide a constructor, accessors and a test:

| | |
|---|---|
| `(make-idpredconst name types cterms)` | constructor, |
| `(idpredconst-to-name idpredconst)` | accessor, |
| `(idpredconst-to-types idpredconst)` | accessor, |
| `(idpredconst-to-cterms idpredconst)` | accessor, |
| `(idpredconst? x).` | |

To introduce inductively defined predicates we use `add-ids`, for example

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

This 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 has computational content. If this is an already known algebra, the clauses with this constant in the conclusion must have the same types for their extracted terms as the constructors of the algebra. If no such algebra is known, we can also write `algEven` (instead of `nat`) to create one. The clauses can be given names (here `InitEven`, `GenEven`), and are saved as theorems under these names.

For the inductive definition of the reflexive transitive closure of a binary relation $\prec$ we have two variants of this definition, depending on whether possible computational content of the relation $\prec$ is taken into account or not. If not we take

```
(add-ids
 (list (list "RTClNc" (make-arity (py "alpha") (py "alpha"))
       "nat"))
 '("allnc x^(RTClNc x^ x^)" "InitRTClNc")
 '("allnc x^,y^,z^(R y^ z^ --> RTClNc x^ y^ -> RTClNc x^ z^)"
   "GenRTClNc"))
```

and otherwise

```
(add-ids
 (list (list "TClCr" (make-arity (py "alpha") (py "alpha"))
       "list"))
 '("allnc x^(TClCr x^ x^)" "InitTClCr")
 '("allnc x^,y^,z^(R y^ z^ -> TClCr x^ y^ -> TClCr x^ z^)"
   "GenTClCr"))
```

The difference between the "non-computational implication" (displayed `-->`) and the computational one (displayed `->`) is explained in section 7.2.

For an inductive definition of the accessible part of a binary relation P we consider the case that the relation $\prec$ is decidable, i.e., given by a boolean-valued binary function `r^`

```
(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 may also have the string `identity` in the field where an algebra name is expected. This is allowed if and only if there is exactly one clause where the type of its extracted term is essentially the identity. Then no new algebra is created. Later $\lambda_x x$ will be taken as realizer for the (single) clause, and $\lambda_{x,f}(fx)$ as realizer for the elimination axiom. Examples are computational variants `ExL`, `ExR` and `AndR` of the (inductively defined) existential quantifier and conjunction.

We also allow non-computational (n.c.) inductively defined predicates. Then no algebra name is provided. Important special cases are:

(i) For every `I` its witnessing predicate `IMR`. It is special in the sense that `(IMR t ss)` just states the fact that `t` is a realizer for `I ss`.

(ii) By providing just one nullary clause with $\forall^{\mathrm{nc}}$, $\to^{\mathrm{nc}}$ only and no algebra name one can introduce a "uniform one clause defined" idpredconst which is n.c. Examples are Leibniz equality `EqD`, and uniform variants `ExNc` and `AndNc` of the existential quantifier and conjunction.

In all other cases the elimination scheme must be restricted to n.c. formulas. Also, all (n.c.) clauses must be invariant. This ensures that the soundness theorem holds: every introduction and elimination axiom is invariant, i.e., $\varepsilon\,\mathbf{r}\,A$ is the same as $A$.

It is also possible to introduce simultaneously inductively defined predicates:

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

However, for simplicity we have restricted the discussion above to the non-simultaneous case.

An important example for an inductively defined predicate is the totality predicate for an algebra, for instance `TotalNat` for the algebra `nat`. It can be created by calling `(add-totality` *`alg-name`*`)`. The same can be done for relative totality by calling `(add-rtotality` *`alg-name`*`)`, yielding for instance `RTotalList` for the algebra `list`.

To remove a name for an inductively defined predicate constant (and also the ones defined simultaneously with it), we use

`(remove-idpc-name` *`name1`* `...)`.

Coinductively defined predicates are in many aspects similar to inductively defined ones, and it seems easiest to use most of the functions with `idpredconst` in their name for both. We even insert the names of the coinductively defined predicates in `IDS`; however, there is also a global variable `COIDS` (with the same format) for the coinductively defined ones only. `add-co` adds dualized "companions" for inductively defined predicate constants to `COIDS`. Examples are cototality predicates for the corresponding total ones, but also for instance `CoEv`, `CoOd` for `Ev`, `Od`. The optional algebra names are the same as for the corresponding inductively defined predicate constants. Realizers for cototality predicates are the cototal ideals of the algebra.

The Minlog command for coinduction is `coind` (cf. section 11.22).

## 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 algebra 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. |

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 [18]). Moreover we provide

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

For displaying terms we have

`(term-to-string` *term*`)`,

which is defined by

`(token-tree-to-string (term-to-token-tree` *term*`))`.

For better line breaks in the display one can use

`(pp` *term*`)`,

which is defined by

`(token-tree-to-pp-tree (term-to-token-tree` *term*`))`.

Sometimes for readability it is helpful to have special support for definitions by cases. Then it is advisable to use

`(pretty-print-with-case-display` *term*`)`,

abbreviated `(ppc` *term*`)`. Moreover we provide

`(term-to-scheme-expr` *term*`)`,

`(term-to-haskell-expr` *term*`)`,

`term-to-expr` is used as abbreviation for `term-to-scheme-expr`. These functions aim at producing a readable Scheme / Haskell expression that can be evaluated. For instance `term-to-expr` transforms an application of a program constant `c` to `args`, where `c` has a corresponding built-in Scheme operator written in uncurried form with length of `args` many arguments, into the corresponding Scheme expression. If however `c` is applied to fewer arguments, then the default translation of `c` is used. Equality with name "=" requires a special treatment: if there are exactly two arguments, it is transformed into an =-expression if the type of = refers to a number type (`nat`, `pos`, `int` or `rat`), and to an `equal?`-expression otherwise. If it is applied to fewer arguments, then one needs `FinAlg=` as a special default name, since the internal name = cannot be used.

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 [6], and extended to arbitrary computation and rewrite rules in [5].

For normalization by evaluation we need semantical *objects*. For an arbitrary ground type every term family of that type is an object. For an algebra 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. [5]):

`(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-without-eta` *term*`).`

The result is a term in long normal form; to transform it into $\eta$-normal form one can use

`(term-to-eta-nf` *term*`).`

We now aim at a full normalization of terms, including permutative conversions. Here the `if`-form needs a special treatment. In a preprocessing step, we $\eta$-expand the alternatives of `if`-terms, using

`(term-to-term-with-eta-expanded-if-terms` *term*`).`

The result contains `if`-terms with ground type alternatives only. Then permutative conversions for `if`-terms can be performed. Notice that this is not possible for recursion terms, but is is if we have recursion terms with no recursive calls, i.e., essentially cases terms: they can be replaced by `if`-terms. The relevant function is

`(normalize-term-pi-with-rec-to-if` *term*`).`

Using these (and some other) auxiliary functions we finally define

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

abbreviated `nt`.

We also provide `term-to-term-without-predecided-ifs`. It simplifies all `if`-terms whose branch is known because we are in a branch of an outer `if`-term with the same test term.

As an alternative to normalization by evaluation, we can also normalize "by hand". This is done via

`(term-to-one-step-beta-reduct` *term*`)`,
`(term-in-beta-normal-form?` *term*`)`,
`(term-to-beta-nf` *term*`)`,
`(term-to-beta-eta-nf` *term*`)`,
`(term-to-beta-pi-eta-nf` *term*`)`

abbreviated `bpe-nt`.

We also provide some auxiliary functions to analyze terms. In

```
(term-in-rec-normal-form? term)
```

we assume that *term* is not one of those appearing during proof normalization. This means that all recursion constants are without repro data.

```
(term-to-consts term)
```

returns a list of all constants in a term (without repetitions). For tests it can be useful to have a level-wise decomposition of terms into subterms: one level transforms a term $N\lambda_{\vec{u}}(vM_1 \dots M_n)$ into the list $N, v, M_1, \dots, M_n$. The general function is

```
(term-to-subterms term opt-level).
```

Example (let introduction). In practice it often happens that an extracted term contains multiple occurrences of the same subterm. One can (and should) avoid this by using the "identity theorem" `Id` (proving $P \to P$ with a predicate variable $P$) at appropriate places in the underlying proof. This amounts to the introduction of a "let" in the term, which is also displayed in this form. Here is an example: let $f$ be variable of type $\mathbf{N} \to \mathbf{N}$ and $g$ of type $\mathbf{N} \to \mathbf{B}$. Consider the proof

```
(set-goal "all f,g,n ex boole(
  (boole -> ex m(m<f n & g m) -> F) &
  ((ex m(m<f n & g m) -> F) -> boole))")
(assume "f" "g" "n")
(ex-intro (pt "NatLeast(f n)g=f n"))
(split) ;4,5
(assume "EqHyp" "ExHyp")
(by-assume "ExHyp" "m" "mProp")
(assert "NatLeast(f n)g<f n")
 (use "NatLeLtTrans" (pt "m"))
 (use "NatLeastLeIntro")
 (use "mProp")
 (use "mProp")
 (simp "EqHyp")
(assume "Absurd")
(use "Absurd")
;; Goal 5
(assume "NegExHyp")
(use "NatLeGeToEq")
(use "NatLeastBound")
(use "NatNotLtToLe")
```

```
(assume "LtHyp")
(use "NegExHyp")
(ex-intro (pt "NatLeast(f n)g"))
(split)
(use "LtHyp")
(use "NatLeastLtElim")
(use "LtHyp")
;; Proof finished.
(pp (nt (proof-to-extracted-term)))
;; [f0,g1,n2]NatLeast(f0 n2)g1=f0 n2
```

A problem is that when evaluating this term one needs to compute `f0 n2` twice. To introduce the desired "let", at a place where the term to be taken out can be constructed (here: `f n`) one cuts in the formula $E :=$ `ex n0 n0=f n`. This generates two new goals: an implication $E \to A$ (where A is the present goal), and $A$, with the implication to be proved first. Now here one uses the identity theorem `Id`, and then carries on with assuming the existential hypothesis, and taking an `n0` with the definition `n0=f n` into the context. In the extracted term this will yield the constant `cId` (evaluating to $\lambda_f f$) applied to $\lambda_{n_0} r(n_0)$ and $fn$, displayed as `[let n0 (f n) r(n0)]`. It is only after "animating" `Id` (i.e., adding the computation rule `cId` $\mapsto$ $\lambda_f f$) that this term evaluates to `r(f n)`, as desired.

```
(set-goal "all f,g,n ex boole(
  (boole -> ex m(m<f n & g m) -> F) &
  ((ex m(m<f n & g m) -> F) -> boole))")
(assume "f" "g" "n")
(cut "ex n0 n0=f n")
;; (use "Id") ;can be slow.  Use use-with instead:
(use-with
  "Id" (make-cterm (goal-to-formula (current-goal))) "?")
(assume "Exn0")
(by-assume "Exn0" "n0" "n0=f n")
(ex-intro (pt "NatLeast n0 g=n0"))
(split) ;11,12
(assume "EqHyp" "ExHyp")
(by-assume "ExHyp" "m" "mProp")
(assert "NatLeast(f n)g<f n")
 (use "NatLeLtTrans" (pt "m"))
 (use "NatLeastLeIntro")
 (use "mProp")
 (use "mProp")
 (simp "<-" "n0=f n")
```

```
 (simp "EqHyp")
(assume "Absurd")
(use "Absurd")
;; Goal 5
(assume "NegExHyp")
(use "NatLeGeToEq")
(use "NatLeastBound")
(use "NatNotLtToLe")
(assume "LtHyp")
(use "NegExHyp")
(ex-intro (pt "NatLeast(f n)g"))
(split)
(simp "<-" "n0=f n")
(use "LtHyp")
(use "NatLeastLtElim")
(simp "<-" "n0=f n")
(use "LtHyp")
;; Now we prove the formula cut in above.
(ex-intro (pt "f n"))
(use "Truth")
;; Proof finished.
(pp (nt (proof-to-extracted-term)))
;; [f0,g1,n2][let n3 (f0 n2) (NatLeast n3 g1=n3)]
```

However, sometimes it is not easy to find the right places for introducting a cut. For such situations it can be helpful to hand optimize a term by searching for its longest duplicate subterm, and taking that subterm out via a "let". The relevant function is

(term-to-term-with-let *term*).

As test functions we provide

(term-form? *x*),
(term? *x*),
(check-term *x*)

abbreviated `ct`. Here `term?` returns `#t` or `#f`, and `check-term` is a complete test returning an error if the argument is not a term.

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*).

We also provide

(term-gen-substitute *term gen-subst*),
(term-gen-subst *term term1 term2*).

`term-gen-substitute` substitutes simultaneously the left hand sides of the association list *gen-subst* (associating terms to terms) at all occurrences in *term* with no free variables captured by the corresponding right hand sides. Renaming takes place if and only if a free variable would become bound.

6.4. **Unification and matching.** For first order unification we use `unify`. It checks whether two terms can be unified, returns `#f` if this is impossible, and a most general unifier otherwise. `unify-list` does the same for lists of terms. The implemented algorithm makes use of disagreement pairs, and does not yield idempotent unifiers (as opposed to the Martelli-Montanari algorithm [21], implemented in `modules/type-inf.scm`).

For first order matching we use `match`. It checks whether a given pattern (term or formula with type variables in its types) can be transformed by a tosubst - respecting totality constraints - into a given instance, such that (i) no type variable from a given set of identity variables, and (ii) no object variable from a given set of signature variables gets substituted. It returns `#f`, if this is impossible, and the tosubst otherwise.

For higher-order unification we use Huet's [17] unification algorithm, and (for the approriate fragment) also Miller's [25] pattern unification algorithm; both are discussed in 12. Higher-order matching is implemented as `huet-match`, which is defined as a special case of `huet-unifiers`: no flexible variables are allowed in the instance. `huet-match` picks a most detailed substitution. Higher-order matching for type substitutions is implemented as `pattern-and-instance-to-tsubst`.

## 7. Formulas and comprehension terms

7.1. **Constructors and accessors.** 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 $\mathbf{F}$),
(vi) `(excl (`*`x1 ...xn`*`)` *`formula`*`)` classical existential quantification (with the logical form of falsity $\bot$),
(vii) `(excu (`*`x1 ...xn`*`)` *`formula`*`)` classical existential quantification (with the logical form of falsity $\bot$, and using $\forall^{\mathrm{nc}}$ rather than $\forall$ in the unfolded form),
(viii) `(tensor` *`formula1 formula2`*`)` tensor, for proper unfolding of formulas containing `exca`, `excl` or `excu`.

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

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

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. We also have the built-in versions:

(i) `(and` *`formula1 formula2`*`)` conjunction
(ii) `(ex` *`x formula`*`)` existential quantification

We also allow prime formulas of the form `(atom r)` with a term `r` of type boole. They are just shorthand for 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*),
(impnc-form? *formula*),
(and-form? *formula*),
(tensor-form? *formula*),
(all-form? *formula*),
(allnc-form? *formula*),
(ex-form? *formula*),
(exca-form? *formula*),
(excl-form? *formula*),
(excu-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-eqd *term1 term2*) | constructor for Leibniz equalities, |
| (make-= *term1 term2*) | constructor for equalities (atomic or eqd), |
| (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, |

non-computational implications ($\to^{\mathrm{nc}}$; displayed `-->`)

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

universally quantified formulas without computational content ($\forall^{\mathrm{nc}}$)

`(make-allnc` *`var formula`*`)` constructor,
`(allnc-form-to-var` *`allnc-formula`*`)` accessor,
`(allnc-form-to-kernel` *`allnc-formula`*`)` accessor,

existentially quantified formulas

`(make-ex` *`var formula`*`)` constructor,
`(ex-form-to-var` *`ex-formula`*`)` accessor,
`(ex-form-to-kernel` *`ex-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,

existentially quantified formulas in the sense of classical logic w.r.t. $\forall^{nc}$

| | |
|---|---|
| `(make-excu` *var formula*`)` | constructor, |
| `(excu-form-to-var` *excu-formula*`)` | accessor, |
| `(excu-form-to-kernel` *excu-formula*`)` | accessor. |

By means of inductively defined predicate constants, we have defined computationally sensitive forms of existential quantification `exd`, `exl`, `exr`, `exnc` written $\exists^{d}$, $\exists^{l}$, $\exists^{r}$, $\exists^{u}$ and also their total versions `exdt`, `exlt`, `exrt`, `exnct`, conjunction `andd`, `andr`, `andnc` written $\wedge^{d}$, $\wedge^{r}$, $\wedge^{u}$ (`andb` is used for the boolean operator), disjunction `or`, `orl`, `orr`, `oru`, `ornc` written $\vee^{d}$, $\vee^{l}$, $\vee^{r}$, $\vee^{u}$, $\vee^{nc}$ (`orb` is used for the boolean operator). For all these we have similar constructors and accessors. There is also some mild form of automatization for these computationally sensitive forms of logical connectives, which provides for computational content only if there is some:

`(make-exi` *var formula*`)`,
`(make-exnci` *var formula*`)`,
`(make-andi` *formula1 formula2*`)`,
`(make-ori` *formula1 formula2*`)`.

For convenience we have as generalized constructors

| | |
|---|---|
| `(mk-imp` *formula formula1 ...*`)` | implication, |
| `(mk-impnc` *formula formula1 ...*`)` | n.c. 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-allnc` *var1 ... formula*`)` | allnc-formula, |
| `(mk-ex` *var1 ... formula*`)` | ex-formula, |
| `(mk-exca` *var1 ... formula*`)` | classical ex-formula (arithmetical), |
| `(mk-excl` *var1 ... formula*`)` | classical ex-formula (logical), |
| `(mk-excu` *var1 ... formula*`)` | classical ex-formula (logical, n.c.). |

and similar for the computationally sensitive logical connectives: `mk-exd`, `mk-exl`, `mk-exr`, `mk-exnc`, `mk-exdt`, `mk-exlt`, `mk-exrt`, `mk-exnct`, `mk-andd`, `mk-andr`, `mk-andnc`, `mk-ord`, `mk-orl`, `mk-orr`, `mk-oru`, `mk-exi`, `mk-exnci`, `mk-andi`, `mk-ori`. As generalized accessors we have

`(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 again similar for `impnc-`, `allnc-`, `exca-` and `excl`-forms, and the computationally sensitive logical connectives. 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 (similar for `impnc`-forms).

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

| | |
|---|---|
| `(make-bicon` *bicon formula1 formula2*`)` | constructor, |
| `(bicon-form-to-bicon` *bicon-form*`)` | accessor, |
| `(bicon-form-to-left` *bicon-form*`)` | accessor, |
| `(bicon-form-to-right` *bicon-form*`)` | accessor, |
| `(bicon-form?` *x*`)` | test, |
| `(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*`).`

We also provide

`(prime-predicate-form?` *x*`)`,
`(prime-form?` *x*`)`,
`(quant-prime-form?` *x*`)`,
`(quant-free?` *x*`)`

and for prime, imp, impnc, all or allnc formulas

`(formula-to-head` *formula*`)`.

To fold and unfold (classical existential quantifiers in) formulas we have

`(fold-formula` *formula*`)`,

(unfold-formula *formula*).

To test equality of formulas up to normalization and $\alpha$-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*),
(formula-to-tvars *formula*),
(formula-to-pvars *formula*),
(ex-free-formula? *formula*),
(nbe-formula-to-type *formula*),
(formula-to-prime-subformulas *formula*).

nbe-formula-to-type needs a procedure associating type variables to predicate variables, which remembers the assignment done so far. Therefore it refers to the global variable PVAR-TO-TVAR. This machinery will be used to assign recursion constants to induction constants. There we need to associate type variables with predicate variables, in such a way that we can later refer to this assignment.

We also provide

(formula-to-prime-subformulas *formula*).

As an alternative to normalization by evaluation, we can also normalize "by hand". This is done via

(formula-to-beta-nf *formula*),
(cterm-to-beta-nf *cterm*),
(formula-to-eta-nf *formula*),
(cterm-to-eta-nf *cterm*),
(formula-to-beta-eta-nf *formula*),
(cterm-to-beta-eta-nf *cterm*).

Clearly every quantifier-free formula can be converted into a term of type boole; this is done by

(qf-to-term *formula*).

We also provide

(alpha-equal-formulas-to-renaming *formula1 formula2*).

The constructor and accessors for comprehension terms are

| | |
|---|---|
| `(make-cterm` *var1 ... formula*`)` | constructor, |
| `(cterm-to-vars` *cterm*`)` | accessor, |
| `(cterm-to-formula` *cterm*`)` | accessor. |

Moreover we need

`(cterm-to-arity` *cterm*`)`,
`(cterm-to-free` *cterm*`)`,
`(cterm-to-bound` *cterm*`)`,
`(fold-cterm` *cterm*`)`,
`(unfold-cterm` *cterm*`)`,
`(pvar-cterm-to-pvar` *cterm*`)`,
`(pvar-cterm?` *cterm*`)`.

7.2. **Decoration.** We think of (computationally relevant) implication $\to$ and universal quantification $\forall$ as "decorated" versions of their non computational counterparts $\to^{\mathrm{nc}}$ and $\forall^{\mathrm{nc}}$ (cf. 7.1). Moreover, existential quantifiers $\exists^{\mathrm{d}}$, $\exists^{\mathrm{l}}$, $\exists^{\mathrm{r}}$ can be seen as decorated versions of the non computational quantifier $\exists^{\mathrm{u}}$, and similar for conjunction and disjunction. To "undecorate" formulas and comprehension terms we use

`(formula-to-undec-formula` *formula id-deco?*`)`,
`(cterm-to-undec-cterm` *cterm id-deco?*`)`.

Both change all occurrences of $\to$, $\forall$ into $\to^{\mathrm{nc}}$, $\forall^{\mathrm{nc}}$, and in case `id-deco?` is true (`id` for "inductively defined logical connective"),

(i) existential quantification $\exists^{\mathrm{d}}$, $\exists^{\mathrm{l}}$, $\exists^{\mathrm{r}}$ into $\exists^{\mathrm{u}}$,
(ii) conjunction $\wedge^{\mathrm{d}}$, $\wedge^{\mathrm{r}}$ into $\wedge^{\mathrm{u}}$, and
(iii) disjunction $\vee^{\mathrm{d}}$, $\vee^{\mathrm{l}}$, $\vee^{\mathrm{r}}$ into $\vee^{\mathrm{u}}$.

They do not touch formulas of nulltype under extension, and in case `id-deco?` is false do not touch any formula of nulltype.

Conversely, `formula-to-dec-formula` changes all occurrences of $\to^{\mathrm{nc}}$, $\forall^{\mathrm{nc}}$ to $\to$, $\forall$. We say that `formula1` *extends* `formula2` if some $\to^{\mathrm{nc}}$, $\forall^{\mathrm{nc}}$ have been changed into $\to$, $\forall$; then `formula1` has a more complex type:

`(extending-dec-variants?` *formula1 formula2 id-deco?*`)`.

In case `id-deco?` is true "extension" is transferred in the expected way to $\exists^{\mathrm{d}}$, $\exists^{\mathrm{l}}$, $\exists^{\mathrm{r}}$, $\exists^{\mathrm{u}}$, $\wedge^{\mathrm{d}}$, $\wedge^{\mathrm{r}}$, $\wedge^{\mathrm{u}}$, $\vee^{\mathrm{d}}$, $\vee^{\mathrm{l}}$, $\vee^{\mathrm{r}}$ and $\vee^{\mathrm{u}}$.

7.3. **Normalization.** Normalization of formulas is done with

(normalize-formula *formula*),
(normalize-cterm *cterm*).

The former is abbreviated by `nf`.

7.4. **Alpha-equality.** To check equality of formulas and comprehension terms we use

(classical-formula=? *formula1 formula2 opt-ignore-deco-flag*),
(classical-cterm=? *cterm1 cterm2 opt-ignore-deco-flag*),
(formula=? *formula1 formula2 opt-ignore-deco-flag*),
(cterm=? *cterm1 cterm2 opt-ignore-deco-flag*)

where the "classical" variants unfold classical existential quantifiers and normalize all subterms in its formulas.

`rename-variables` renames bound variables in terms, formulas and comprehension terms.

7.5. **Display.** For a readable display of formulas we normally use

(pp *formula*)

which is implemented using as auxiliary functions

(predicate-to-token-tree *pred*),
(formula-to-token-tree *formula*).

Alternative display functions for formulas and comprehension terms are

(formula-to-string *formula*),
(cterm-to-string *cterm*).

7.6. **Check.** As test functions we provide

(formula-form? *x*),
(cterm-form? *x*),
(formula? *x*),
(cterm? *x*),
(check-formula *x*),

abbreviated `cf`. Here `formula?` returns `#t` or `#f`, and `check-formula` is a complete test returning an error if the argument is not a formula.

7.7. **Substitution.** 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`*`)`.

In a simultaneous substitution topsubst for type, object and predicate variables in a formula or a comprehension term it is allowed that the substitution affects variables whose type is changed by topsubst, provided topsubst is admissible for the formula or the comprehension term.

We also provide

`(formula-gen-substitute` *`formula gen-subst`*`)`,
`(formula-gen-subst` *`formula term1 term2`*`)`.

The former substitutes simultaneously the left hand sides of the association list *gen-subst* at all occurrences in the formula with no free variables captured by the corresponding right hand sides. *gen-subst* is an association list associating terms to terms. Renaming takes place if and only if a free variable would become bound.

As display functions for substitutions we again use

`(pp-subst` *`topsubst`*`)`
`(display-substitutions` *`topsubst`*`)`.

## 8. 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 [24]) 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. |

Testing equality of assumption variables is often used, and it is expensive since it involves an equality test for formulas (which includes normalization). To allow a more efficient equality test, we maintain an *avar-convention*: whenever two assumption variables have the same identifier, their formulas are equal as well; therefore `avar=?` only checks equality of identifiers. For a full test one can use

`(avar-full=?` *avar1 avar2 opt-ignore-deco-flag*`)`.

For convenience we have the function

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

The formula is a required argument; the remaining arguments are optional. The default for the name string is `u`. For display we use

`(avar-to-string` *avar*`)`.

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.

## 9. Assumption constants

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. An exception are the elimination and greatest-fixed-point axioms, where the argument variables of the (co)inductively defined predicate are formally free in the uninstantiated formula; however, they are considered bound as well. In the proof the bound type variables are implicitly instantiated by types, and the bound predicate variables by comprehension terms (the arity of a comprehension term is the type-instantiated arity of the corresponding predicate variable). Since we do not have explicit type and predicate quantification in formulas, the assumption constant contains these parts left implicit in the proof, as `tpsubst`.

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 "reproduction data" to be used when after normalization the axiom in question is to be reconstructed: `all-formulas` for induction, a number `i` and an inductively defined predicate constant `idpc` for its `i`-th clause, `imp-formulas` for elimination, 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 reproduction data 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

$$\texttt{(aconst ExIntro } \forall_{\hat{x}^\alpha}(P\hat{x} \to \exists_{\hat{x}^\alpha} P\hat{x}) \ (\alpha \mapsto \tau, P^{(\alpha)} \mapsto \{\, \hat{z}^\tau \mid A \,\})\texttt{)},$$

$$\texttt{(aconst ExElim } \exists_{\hat{x}^\alpha} P\hat{x} \to \forall_{\hat{x}^\alpha}(P\hat{x} \to Q) \to Q \quad (\alpha \mapsto \tau, P^{(\alpha)} \mapsto \{\, \hat{z}^\tau \mid A \,\}, Q \mapsto \{ \mid B \,\})\texttt{)}.$$

Interface for general assumption constants:

| | |
|---|---|
| `(make-aconst` *`name kind uninst-formula tpsubst`* | |
| *`repro-data1 ...`*`)` | 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-data` *`aconst`*`)` | accessor, |
| `(aconst-form?` *`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`*`)`.

The reproduction data can be computed from the name, the uninstantiated formula, the `tpsubst` of an axiom, by

`(aconst-to-computed-repro-data` *`aconst`*`)`.

However, to avoid recomputations we carry them along.

We also provide

`(check-aconst x)`,
`(aconst=? aconst1 aconst2)`,
`(aconst-without-rules? aconst)`,
`(aconst-to-string aconst)`.

9.1. **Axioms.** In TCF the only axioms are the clauses and the least- or greatest-fixed-point axioms of inductively or coinductively defined predicates, and equality axioms stating the (Leibniz) equality of both sides of a computation rule. However, as long as (i) we allow free type parameters and (ii) make use of convenient abbreviations, we need to allow corresponding additional axioms.

Recall that we require nullary constructors in every free algebra; hence, it has a "canonical inhabitant". Since we allow free type parameters $\alpha$, we provide a constant $\texttt{Inhab}_\alpha$ intended to denote the canonical inhabitant of the (unknown) type $\alpha$. When finally we substitute a closed type $\rho$ for the type parameter $\alpha$, we can give a value to $\texttt{Inhab}_\rho$ by adding an appropriate computation rule; an example is

```
(add-computation-rule (pt "(Inhab nat)") (pt "0")).
```

Since for closed types the canonical inhabitant is total, we add an axiom stating the totality of $\texttt{Inhab}_\alpha$; it appears among the initial theorems under the name InhabTotal.

Recall that in order 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, $\texttt{n}, \texttt{n0}, \texttt{n1}, \texttt{n2}, \ldots$ range over total natural numbers, and $\texttt{n\^{}}, \texttt{n\^{}0}, \texttt{n\^{}1}, \texttt{n\^{}2}, \ldots$ are general variables. Formally this is done by providing the abbreviating axioms

$$\texttt{AllTotalElim}\colon \forall_x Px \to \forall^{\mathrm{nc}}_{\hat{x}}(T\hat{x} \to P\hat{x}),$$
$$\texttt{AllncTotalElim}\colon \forall^{\mathrm{nc}}_x Px \to \forall^{\mathrm{nc}}_{\hat{x}}(T\hat{x} \to^{\mathrm{nc}} P\hat{x})$$

and their converses `AllTotalIntro` and `AllncTotalIntro`. For the inductively defined (decorated) existential quantifiers we have the abbreviating axioms

$$\texttt{ExDTotalElim}\colon \exists^{\mathrm{d}}_x Px \to \exists^{\mathrm{r}}_{\hat{x}}(T\hat{x} \wedge^{\mathrm{d}} P\hat{x}),$$
$$\texttt{ExLTotalElim}\colon \exists^{\mathrm{l}}_x Px \to \exists^{\mathrm{r}}_{\hat{x}}(P\hat{x} \wedge^{\mathrm{r}} T\hat{x}),$$
$$\texttt{ExRTotalElim}\colon \exists^{\mathrm{r}}_x Px \to \exists^{\mathrm{r}}_{\hat{x}}(T\hat{x} \wedge^{\mathrm{r}} P\hat{x}),$$
$$\texttt{ExNcTotalElim}\colon \exists^{\mathrm{u}}_x Px \to \exists^{\mathrm{u}}_{\hat{x}}(T\hat{x} \wedge^{\mathrm{u}} P\hat{x})$$

and their converses `ExDTotalIntro`, `ExLTotalIntro`, `ExRTotalIntro` and `ExNcTotalIntro`. For the primitive existential quantifier we have the abbreviating axiom

$$\texttt{ExTotalElim}\colon \exists_x Px \to \exists^{\mathrm{r}}_{\hat{x}}(T\hat{x} \land P\hat{x})$$

and its converse `ExTotalIntro`.

Recall the treatment in 5.3 of induction axioms, viewed as elimination axioms for inductively defined totality predicates. We can also use the predicate constant $T$ instead, which gives essentially the same axioms; this is what `ind` calls. In more detail, the command (`ind r`) expects a goal $A(r)$ with a syntactically total term $r$. Then the induction axiom

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

is used with the term $r$ and two new goals for the base and the step case. Similarly, (`elim Hyp`) with a hypothesis `Hyp`: $Tr$ and a goal $A(r)$ uses the elimination axiom

$$\forall^{\mathrm{nc}}_{\hat{n}}(T\hat{n} \to A(0) \to \forall^{\mathrm{nc}}_{\hat{n}}(A(\hat{n}) \to A(\mathrm{S}\hat{n})) \to A(\hat{n}))$$

with the term $r$, the hypothesis `Hyp` and two new goals for the base and the step case. The resulting proofs clearly can be transformed into each other using the abbreviating axioms above dealing with total variables.

We now spell out what in detail we mean by induction over *simultaneous* free algebras $\vec{\iota} = \mu_{\vec{\xi}}\vec{\kappa}$, with goal formulas $\forall^{\iota_j}_{x_j} P_j x_j$. 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*

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

Here $\vec{y} = y_1^{\rho_1},\dots,y_m^{\rho_m},y_{m+1}^{\vec{\sigma}_1\to\iota_{j_1}},\dots,y_{m+n}^{\vec{\sigma}_n\to\iota_{j_n}}$ are the *components* of the object $\mathrm{C}^{\vec{\iota}}_i(\vec{y}\,)$ of type $\iota_j$ 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 $\mathrm{Ind}_{\iota_j}$ then proves the formula

$$\mathrm{Ind}_{\iota_j}\colon D_1 \to \cdots \to D_k \to \forall^{\iota_j}_{x_j} P_j x_j.$$

We will often write $\mathrm{Ind}_j$ for $\mathrm{Ind}_{\iota_j}$.

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

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

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

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

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

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

with

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

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`. |

`all-formula-and-number-to-gind-aconst` takes an all-formula, a number $n$ for the arity of the measure function and an optional argument for the name of a theorem proving general induction from induction. If `opt-gindthmname` is not present, general induction is viewed as an axiom (and `GRec` will be extracted). Otherwise general induction is viewed as proved from structural induction (and `Rec` is extracted).

We also provide

`(formula-to-efq-aconst` *formula*`)`,

which is conceived as a global assumption (of $\mathbf{F} \to A$). As we have seen in section 5.3, it can be proved if the formula $A$ has no stricly positive occurrences of predicate variables (cf. section 10.8).

For the introduction and elimination axioms `ExIntro` and `ExElim` of the primitive existential quantifier we provide

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

To deal with inductively defined predicate constants, we need additional axioms with names "Intro" and "Elim", which are 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 $I\vec{x} \to A$. (For simultaneously inductively defined predicates we need many such `imp-formulas`).

For coinductively defined predicate constants we need additional axioms, with names "Closure" and "Gfp". They are generated by

```
(coidpredconst-to-closure-aconst coidpc),
(imp-formulas-to-gfp-aconst imp-formula1 ...).
```

9.2. **Theorems.** A theorem is a special assumption constant. We maintain an association list `THEOREMS` assigning to every name of a theorem the assumption constant and its proof.

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

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

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

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

We also provide

```
(remove-theorem string1 ...),
(display-theorems string1 ...),
(pp theorem-name).
```

Initially we provide the following theorems

| | |
|---|---|
| $\mathrm{atom}(\hat{p}) \to \hat{p} = \mathtt{tt}$ | `AtomTrue` |
| $(\mathrm{atom}(p) \to \mathbf{F}) \to p = \mathtt{ff}$ | `AtomFalse` |

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

$$\begin{aligned} &n = n && \texttt{=-Refl-nat}\\ &\hat{n}_1 = \hat{n}_2 \to \hat{n}_2 = \hat{n}_1 && \texttt{=-Sym-nat}\\ &\hat{n}_1 = \hat{n}_2 \to \hat{n}_2 = \hat{n}_3 \to \hat{n}_1 = \hat{n}_3 && \texttt{=-Trans-nat}. \end{aligned}$$

Notice that the more general $(\mathrm{atom}(\hat{p}) \to \mathbf{F}) \to \hat{p} = \mathsf{ff}$ does *not* hold. A counterexample is the empty ideal of type $\mathbf{B}$.

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`

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

`(aconst Minpr-with-measure-l11`

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

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 $\mathbf{F}$). `l` 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

$$(P \to P_1) \to ((P \to \bot) \to P_1) \to P_1 \quad \texttt{CasesLog}.$$

The proof uses the global assumption `StabLog` (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*`)`.

9.3. **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).

| | |
|---|---|
| $\bot \to P$ | `EfqLog`, |
| $((P \to \bot) \to \bot) \to P$ | `StabLog`, |
| $\mathbf{F} \to P$ | `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`*`)`.

It is a practical problem to find existing theorems or global assumptions relevant for the situation at hand. To help searching for those we provide

`(search-about` *`symbol-or-string . opt-strings`*`)`.

It searches in `THEOREMS` and `GLOBAL-ASSUMPTIONS` for all items whose name contains each of the strings given, excluding the strings Total Partial CompRule RewRule Sound. It one wants to list all these as well, take the symbol 'all as first argument.

## 10. Proofs

Proofs are built from assumption variables and assumption constants (i.e., axioms, theorems and global assumptions) 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.

10.1. **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 ...*`)`.

The result of `(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. Further related functions are

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

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*`)`.

For the non-computational connectives $\to^{\mathrm{nc}}$ and $\forall^{\mathrm{nc}}$ (cf. 7.1) we need similar constructors, accessors and tests. For n.c. implication introduction

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

for n.c. implication elimination

| | |
|---|---|
| `(make-proof-in-impnc-elim-form` *proof1 proof2*`)` | constructor, |
| `(proof-in-impnc-elim-form-to-op` *proof*`)` | accessor, |

`(proof-in-impnc-elim-form-to-arg` *proof*`)` accessor,
`(proof-in-impnc-elim-form?` *proof*`)` test,

for n.c. all introduction

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

for n.c. all elimination

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

Again it is convenient to have

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

We also provide

`(mk-proof-in-cr-nc-intro-form` *x . rest*`)`.

Here `x` is obtained from a list of premises and variables where each element is followed by an indicator for `nc` or `cr` (true means `nc`). Moreover we need

`(proof?` *x*`)`,
`(proof=?` *proof1 proof2*`)`,
`(proofs=?` *proofs1 proofs2*`)`,
`(proof-to-formula` *proof*`)`,
`(proof-to-context` *proof*`)`,
`(proof-to-cvars` *proof*`)`,
`(proof-to-free` *proof*`)`,
`(proof-to-tvars` *proof*`)`,
`(proof-to-pvars` *proof*`)`,
`(proof-to-free-avars` *proof*`)`,
`(proof-to-bound-avars` *proof*`)`,
`(proof-to-free-and-bound-avars-wrt` *avar-eq proof*`)`,
`(proof-to-free-and-bound-avars` *proof*`)`,
`(proof-respects-avar-convention?` *proof*`)`,
`(proof-to-aconsts-without-rules` *proof*`)`,

```
(proof-to-aconsts proof),
(proof-to-global-assumptions proof).
```

`proof-to-cvars` computes the computational variables of the proof, which are the ones the extra variable condition for $\forall^{\mathrm{nc}}$ refers to.

To work with contexts we provide

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

We can also convert the name of a theorem or a global assumption into a proof consisting of just the corresponding assumption constant:

```
(thm-or-ga-name-to-proof name).
```

*Decorating proofs.* In this section we are interested in "fine-tuning" the computational content of proofs, by inserting decorations (cf. 7.2). 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* [28], assigning to every proof pattern $U$ and every extension of its sequent an "optimal" decoration $M_\infty$ 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 $M_\infty$ of $U$ such that*

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

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.

We illustrate the effects of decoration on a simple example (from [28]) involving implications. Consider $A \to B \to A$ with the trivial proof $M := \lambda^A_{u_1}\lambda^B_{u_2}u_1$. Clearly the second implication has no computational significance. We apply the decoration algorithm and specify as extension of $\mathrm{Seq}(\mathrm{P}(M))$ the formula $A \to^{\mathrm{nc}} B \to^{\mathrm{nc}} A$. The algorithm detects that the first implication needs to be decorated, since the abstracted assumption variable is computational. Since the second implication can be left undecorated, a proof of $A \to^{\mathrm{c}} B \to^{\mathrm{nc}} A$ is constructed from $M$.

A similar phenomenon occurs for $A \wedge^{\mathrm{d}} B \to B$. Let $M$ be its proof and $U := \mathrm{P}(M)$ its proof pattern. When given the extension $A \wedge^{\mathrm{u}} B \to^{\mathrm{nc}} B$ for $\mathrm{Seq}(U)$, the decoration algorithm constructs a correct proof of $A \wedge^{\mathrm{r}} B \to^{\mathrm{c}} B$.

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

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

Due to our distinction between general variables `x^0, x^1, x^2, ...` and variables `x0, x1, x2, ...` intended to range over total objects only, $\eta$-conversion of proofs cannot be done via reduction to $\eta$-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 $\eta$-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 $\eta$-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).
```

Normalization of proofs can be made more efficient if we are interested in extraction only. To this end we can provide an "extraction-flag", indicating whether this is the case. If so, we can disregard (maximal) parts of the proof without computational content. Accordingly we have

```
(nbe-normalize-proof proof),
(nbe-normalize-proof-for-extraction proof)
```

abbreviated `np` and `npe`, respectively.

Finally we consider some proof transformations (Prawitz' simplification conversions, removal of predecided assumption variables, removal of predecided if-theorems, generalized pruning).

Simplification conversions (Prawitz [27]) make use of the concept of a permutative assumption constant. It is checked whether one side-proof-kernel has no free occurrence of any assumption variable bound in this side-proof. The corresponding function is

```
(normalize-proof-simp proof).
```

The function

```
(proof-to-proof-without-predecided-avars proof)
```

removes dependencies on assumption variables, and in this way helps to make `normalize-proof-simp` useful.

It can be also useful to remove predecided If's, including those with True or False as boolean arguments. This is particularly important in the context of "pruning" (cf. Chiarabini [8]). We have

```
(prune proof),
(remove-predecided-if-theorems proof).
```

For tests it is useful to have a level-wise decomposition of proofs into subproofs: one level transforms a proof $\lambda_{\vec{u}} v \vec{M}$ into the list $(v, M_1, \ldots, M_n)$. We provide

```
(proof-to-parts proof . opt-level),
(proof-to-proof-parts proof),
(proof-to-depth proof).
```

It can also be useful to do normalization by hand, including $\beta$-conversion and idpredconst-elim-intro conversion. The latter uses for nested idpredconstants

```
(formula-and-psubsts-to-mon-proof proof).
```

An elim-intro redex occurs when an elim aconst is applied to terms and the result of applying an intro-aconst to terms and an idpc-proof.

```
(proof-to-one-step-idpredconst-elim-intro-reduct proof),
(proof-to-one-step-reduct proof),
(proof-to-normal-form proof),
(proof-to-length proof).
```

10.3. **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.

```
(aconst-substitute aconst topsubst),
(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. If opt-name-test is provided, it only expands (non-recursively) the theorems passing the test. expand-thm expands a single theorem given by its name and expand-theorems-with-positive-content does what its name says.

(expand-theorems *proof . opt-name-test*),
(expand-thm *proof thm-name*),
(expand-theorems-with-positive-content *proof*).

10.4. **Display.** There are many ways to display a proof. We normally use display-proof for a linear representation, showing the formulas and the rules used. We also provide a (hopefully) 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. In case the optional proof argument is not present, the current proof is taken instead.

| | |
|---|---|
| (display-proof . *opt-proof*) | abbreviated dp, |
| (display-normalized-proof . *opt-proof*) | abbreviated dnp, |
| (proof-to-expr . *opt-proof*), | |
| (proof-to-expr-with-formulas . *opt-proof*), | |
| (proof-to-expr-with-aconsts . *opt-proof*). | |

Here display-normalized-proof normalizes the proof first. There are the following (less useful) display functions:

| | |
|---|---|
| (display-pterm . *opt-proof*) | abbreviated dpt, |
| (display-proof-expr . *opt-proof*) | abbreviated dpe, |
| (display-normalized-pterm . *opt-proof*) | abbreviated dnpt, |
| (display-normalized-proof-expr . *opt-proof*) | abbreviated dnpe. |

rename-avars renames bound assumption variables in terms, formulas and comprehension terms.

10.5. **Check.** When in addition one wants to check the correctness of the proof, use `check-and-display-proof`, abbreviated `cdp`.

`(check-and-display-proof . opt-proof-and-ignore-deco-flag)`.

`ignore-deco-flag` is set to true as soon as the present proof argument proves a formula of nulltype. There is a global variable `CDP-COMMENT-FLAG` by which one can suppress some of the information `cdp` is providing. Initially `CDP-COMMENT-FLAG` is set to true.

10.6. **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`.

We now consider the Gödel-Gentzen translation, also known as negative translation. It allows embed classical logic into minimal logic; more precisely into its "negative fragment" involving only $\to$ and $\forall$. First we define the Gödel-Gentzen translation of formulas:

`(formula-to-goedel-gentzen-translation formula)`.

We do not consider $\tilde{\exists}$, because it can be unfolded (is not needed for program extraction).

Finally we will define the Gödel-Gentzen translation of proofs. To this end we introduce a further observation (due to Leivant; see Troelstra and van Dalen [41, Ch.2, Sec.3]) which will be particularly useful for program extraction from classical proofs. There it will be necessary to transform a given classical derivation $\vdash_c A$ into a minimal logic derivation $\vdash A^g$. In particular, for every assumption constant $C$ used in the given derivation we have to provide a derivation of $C^g$. Now for some formulas $S$ – the so-called

spreading formulas – this is immediate, for we can derive $S \to S^g$, and hence can use the original assumption constant.

First notice that our formulas may contain *predicate variables* denoted by $X$, which are place holders for comprehension terms, i.e., formulas with distinguished variables. We use the notation $A[X := \{\, \vec{x} \mid B \,\}]$ or shortly $A[\{\, \vec{x} \mid B \,\}]$ or even $A[B]$ for substitution of a comprehension term $\{\, \vec{x} \mid B \,\}$ for the predicate variable $X$. Recall that the Gödel-Gentzen translation of $X\vec{t}$ is $\neg\neg X\vec{t}$.

Recall also that we view an assumption constant as consisting of an uninstantiated formula (e.g., $X0 \to \forall_n(Xn \to X(n+1)) \to \forall_n Xn$ for induction) together with a substitution of comprehension terms for predicate variables (e.g., $X \mapsto \{\, n \mid n < n+1 \,\}$). Then in order to obtain a derivation of $C^g$ for an assumption constant $C$ it suffices to know that its *uninstantiated* formula $S$ is spreading, for then we have $\vdash S[\vec{A}^g] \to S[\vec{A}]^g$ (see the theorem below) and hence can use the same assumption constant with a different substitution.

We define *spreading* formulas $S$, *wiping* formulas $W$ and *isolating* formulas $I$ inductively.

$$\begin{aligned} S &::= \bot \mid R\vec{t} \mid X\vec{t} \mid S \wedge S \mid I \to S \mid \forall_x S, \\ W &::= \bot \mid X\vec{t} \mid W \wedge W \mid S \to W \mid \forall_x W, \\ I &::= R\vec{t} \mid W \mid I \wedge I. \end{aligned}$$

Let $\mathcal{S}$ ($\mathcal{W}$,$\mathcal{I}$) be the class of spreading (wiping, isolating) formulas.

**Theorem.**

$$\begin{aligned} &\vdash S[\vec{A}^g] \to S[\vec{A}]^g && \textit{for every spreading formula } S, \\ &\vdash W[\vec{A}]^g \to W[\vec{A}^g] && \textit{for every wiping formula } W, \\ &\vdash I[\vec{A}]^g \to \neg\neg I[\vec{A}^g] && \textit{for every isolating formula } I. \end{aligned}$$

*We assume here that all occurrences of predicate variables are substituted.*

*Proof.* By induction on the simultaneous generation of $\mathcal{S}$, $\mathcal{W}$ and $\mathcal{I}$. We write $S^g$ for $S[\vec{A}]^g$ and $S$ for $S[\vec{A}^g]$, and similarly for $W$ and $I$.

*Case* $\bot \in \mathcal{S}$. We must show $\vdash \bot \to \bot^g$. Take $\lambda u^{\bot} u$.

*Case* $R\vec{t} \in \mathcal{S}$. We must show $\vdash R\vec{t} \to \neg\neg R\vec{t}$. Take $\lambda u^{R\vec{t}} \lambda v^{\neg R\vec{t}}.vu$.

*Case* $X\vec{t} \in \mathcal{S}$, with $X$ substituted by $\{\, \vec{x} \mid A \,\}$. We must show $\vdash A^g[\vec{t}] \to A^g[\vec{t}]$, which is trivial.

*Case* $S_1 \wedge S_2 \in \mathcal{S}$. We must show $\vdash S_1 \wedge S_2 \to S_1^g \wedge S_2^g$. Take

$$\dfrac{\dfrac{\text{IH} \quad \dfrac{u\colon S_1 \wedge S_2}{S_1}}{S_1^g} \quad \dfrac{\text{IH} \quad \dfrac{u\colon S_1 \wedge S_2}{S_2}}{S_2^g}}{S_1^g \wedge S_2^g}$$

*Case* $I \to S \in \mathcal{S}$. We must show $\vdash (I \to S) \to I^g \to S^g$. Recall that $\vdash \neg\neg S^g \to S^g$ by the Stability Lemma, because $S^g$ is negative. Take

$$\dfrac{\dfrac{\text{Stab} \quad \dfrac{\dfrac{\text{IH} \quad \dfrac{u\colon I \to S \quad w_2\colon I}{S}}{S^g} \quad w_1\colon \neg S^g}{\bot}\to^+ w_2}{\neg I} \quad \dfrac{\dfrac{\text{IH} \quad v\colon I^g}{I^g \to \neg\neg I}}{\neg\neg I}}{\neg\neg S^g \to S^g} \quad \dfrac{\bot}{\neg\neg S^g}\to^+ w_1}{S^g}$$

*Case* $\forall_x S \in \mathcal{S}$. We must show $\vdash \forall_x S \to \forall_x S^g$. Take

$$\dfrac{\dfrac{\text{IH} \quad \dfrac{u\colon \forall_x S \quad x}{S}}{S^g \to S^g}}{S^g}$$

*Case* $\bot \in \mathcal{W}$. We must show $\vdash \bot^g \to \bot$. Take $\lambda u^{\bot} u$.

*Case* $X\vec{t} \in \mathcal{W}$, with $X$ substituted by $\{\, \vec{x} \mid A \,\}$. We must show $\vdash A^g[\vec{t}] \to A^g[\vec{t}]$, which is trivial.

*Case* $W_1 \wedge W_2 \in \mathcal{W}$. We must show $\vdash W_1^g \wedge W_2^g \to W_1 \wedge W_2$. Take

$$\dfrac{\dfrac{\text{IH} \quad \dfrac{u\colon W_1^g \wedge W_2^g}{W_1^g}}{W_1^g \to W_1} \quad \dfrac{\text{IH} \quad \dfrac{u\colon W_1^g \wedge W_2^g}{W_2^g}}{W_2^g \to W_2}}{W_1 \wedge W_2}$$

*Case* $S \to W \in \mathcal{W}$. We must show $\vdash (S^g \to W^g) \to S \to W$. Take

$$\dfrac{\dfrac{\text{IH} \quad \dfrac{u\colon S^g \to W^g \quad \dfrac{\text{IH} \quad \dfrac{S \to S^g \quad v\colon S}{S^g}}{W^g}}{W^g \to W}}{W}$$

*Case* $\forall_x W \in \mathcal{W}$. We must show $\vdash \forall_x W^g \to \forall_x W$. Take

$$\dfrac{\dfrac{\text{IH} \quad \dfrac{u\colon \forall_x W^g \quad x}{W^g}}{W^g \to W}}{W}$$

*Case* $R\vec{t} \in \mathcal{I}$. We must show $\vdash \neg\neg R\vec{t} \to \neg\neg R\vec{t}$, which is trivial.

*Case* $W \in \mathcal{I}$. We must show $\vdash W^g \to \neg\neg W$, which trivially follows from the IH $\vdash W^g \to W$. Take

$$\dfrac{v\colon \neg W \qquad \dfrac{\overset{\text{IH}}{W^g \to W} \qquad u\colon W^g}{W}}{\bot}$$

*Case* $I_1 \wedge I_2 \in \mathcal{I}$. We must show $\vdash I_1^g \wedge I_2^g \to \neg\neg(I_1 \wedge I_2)$. Take

$$\dfrac{\dfrac{\overset{\text{IH}}{I_2^g \to \neg\neg I_2} \qquad \dfrac{I_1^g \wedge I_2^g}{I_2^g}}{\neg\neg I_2} \qquad \dfrac{\dfrac{\overset{\text{IH}}{I_1^g \to \neg\neg I_1} \qquad \dfrac{I_1^g \wedge I_2^g}{I_1^g}}{\neg\neg I_1} \qquad \dfrac{\neg(I_1 \wedge I_2) \qquad \dfrac{I_1 \quad I_2}{I_1 \wedge I_2}}{\bot}}{\neg I_1}}{\dfrac{\bot}{\neg I_2}}}{\bot}$$

□

This completes the proof.

The theory above is implemented as follows. Simultaneously with spreading formulas we need to define wiping and isolating formulas:

```
(spreading-formula? formula),
(wiping-formula? formula),
(isolating-formula? formula),
(spreading-formula-to-proof formula . opt-psubst),
(wiping-formula-to-proof formula . opt-psubst),
(isolating-formula-to-proof formula . opt-psubst).
```

Using these we can define the Gödel-Gentzen translation:

```
(proof-to-goedel-gentzen-translation proof).
```

Notice that the Gödel-Gentzen translation double negates every atom, and hence may produce triple negations. However, we can systematically replace triple negations by single ones. The final result then is

```
(proof-to-reduced-goedel-gentzen-translation proof).
```

10.7. **Existence formulas.** In case of existence formulas $\exists_{\vec{x}_1} A_1 \dots \exists_{\vec{x}_n} A_n$ and conclusion $B$ we recursively construct a proof of

$$\exists_{\vec{x}_1} A_1 \to \dots \exists_{\vec{x}_n} A_n \to \forall_{\vec{x}_1,\dots,\vec{x}_n}(A_1 \to \dots \to A_n \to B) \to B$$

by means of

```
(ex-formulas-and-concl-to-ex-elim-proof x . rest).
```

10.8. **Basic proof constructions.** For every formula $A$, a proof of $\mathbf{F} \to A$ (i.e., ex-falso-quodlibet) is constructed, and also proofs that constructors are injective and have disjoint ranges. For ex-falso-quodlibet we use

```
(formula-to-efq-proof formula).
```

To make this work easily for (simultaneous) inductive definitions, we assume that taking the initial clause of each inductively defined predicate constant produces clauses without recursive calls which are terminating. This is checked in `add-ids`.

Given proofs of Leibniz equalities $r_1 \equiv s_1, \dots, r_n \equiv s_n$ and a predicate-proof of $Ir_1 \dots r_n$ we construct a proof of $Is_1 \dots s_n$ using `EqDCompat` by means of

```
(eqd-proofs-and-predicate-proof-to-proof eqd-proofs
                                         predicate-proof)
```

To generate proofs of the injectivity of constructors we have

```
(constructor-eqd-proof-to-args-eqd-proof eqd-proof
                                         . opt-index).
```

It expects an eqd-proof of $\mathrm{C}\vec{r} \equiv \mathrm{C}\vec{s}$ with the same constructor C and $\mathrm{C}\vec{r}$ of ground type, and an optional index (with default value 0). The result is a proof of $r_i \equiv s_i$.

```
(constructor-eqd-imp-args-eqd-proof eqd-formula . opt-index)
```

is similar, but expects an `eqd-formula` rather than an `eqd-proof` and proves the implication $\mathrm{C}\vec{r} \equiv \mathrm{C}\vec{s} \to r_i \equiv s_i$.

Finally we have

```
(constructors-overlap-imp-falsity-proof eqd-formula).
```

It generates a proof of an implication from an equality between two constructor terms with different constructors at their heads to falsity. In this sense we provide proofs that constructors have disjoint ranges.

## 11. 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.

To construct a goal and access its components we have

`(make-goal-in-avar-form` *avar*`)`,
`(make-goal-in-all-elim-form` *goal uservar*`)`,
`(make-goal-in-allnc-elim-form` *goal uservar*`)`,
`(make-goal-in-imp-elim-form` *avar*`)`,
`(make-goal-in-impnc-elim-form` *avar*`)`,
`(mk-goal-in-elim-form . ` *elim-items*`)`,
`(goal-to-goalvar` *goal*`)`,
`(goal-to-context` *goal*`)`,
`(goal-to-formula` *goal*`)`.

For interactively building proofs we need

`(goal=? ` *proof goal*`)`,
`(goal-subst` *proof goal proof1*`)`.

Initialization of the global variable `PPROOF-STATE` and access to its parts is possible via

`(make-pproof-state` *num-goals proof maxgoal*`)`,
`(pproof-state-to-num-goals)`,
`(pproof-state-to-proof)`,
`(pproof-state-to-formula)`.

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)`.

We initially supply our axioms (see 9.1) as theorems, and also

```
AtomTrue: all boole^(boole^ -> boole^ =True)
AtomFalse: all boole((boole -> F) -> boole=False)
```

There is a global constant `THEOREMS` containing all theorems known to the system, and also their proofs. Similarly we maintain a global constant

`GLOBAL-ASSUMPTIONS`, which initially contains the global assumptions listed in 9.3.

For display we have

`(display-current-goal)` abbreviated `dcg`,

`(display-current-goal-with-normalized-formulas)`,

abbreviated `dcg` and `dcgnf`, respectively. One can switch to a different display style for the current goal by setting `COQ-GOAL-DISPLAY` to true.

We list some commands for interactively building proofs.

11.1. **set-goal.** An interactive proof starts with setting the goal

`(set-goal` *`string-or-formula`*`)`,

i.e., with setting a goal. The goal-formula might be given by its display string. It should be closed; if not, universal quantifiers are inserted automatically.

11.2. **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.

11.3. **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.

Internally, assume extends the partial proof under construction by introduction rules. To every quantifier $\forall_x$ (resp. $\forall_y^{\mathrm{nc}}$) in the present goal corresponds an application of the $\forall^+$-rule (resp. $(\forall^{\mathrm{nc}})^+$-rule). To meet the variable condition for $(\forall^{\mathrm{nc}})^+$-rules, the $\forall^{\mathrm{nc}}$-variable `y` in the assumed context is not admitted as a computational variable in a future proof of the present goal. Therefore it is displayed in braces, as {`y`}.

11.4. **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 (implicitly 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`.

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

Internally `x-and-x-list-to-proof-and-new-num-goals-and-maxgoal` will be used by `use-with` (and also by `inst-with`) to construct the new data. It appears in error messages if the arguments of `use-with` are incorrect.

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 (for instance 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* cannot. 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 hypothesis and the term $Sl$ succeeds (since $k + Sl$ and $S(k + l)$ have the same normal form).

11.6. **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; 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.

11.7. **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.

11.8. **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.

11.9. **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.

11.10. **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)`.

11.11. **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.

11.12. **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 $i$th hypothesis.

11.13. **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).

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

11.15. **undo.** With `(undo . n)`, the last $n$ steps of an interactive proof can be made undone. `(undo)` has the same effect as `(undo 1)`. `(undoto n)` allows to go back to a previous pproof state whose (top) goal had number $n$.

11.16. **ind.** `(ind)` expects a goal $\forall_{x^\iota} A(x)$ with $x$ total and $\iota$ an algebra. Let $c_1, \dots, c_n$ be the constructors of the algebra. Then $n$ new goals $\forall_{\vec{x}_i}(A(x_{1i}) \to \cdots \to A(x_{ki}) \to A(c_i\vec{x}_i)$ are generated. `(ind t)` expects a goal $A(t)$. It computes the algebra $\iota$ as type of the term $t$. Then again the $n$ new goals above are generated.

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

11.18. **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 $\rho_i$ 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 $\forall_{\vec{x}} A(x)$.

11.19. **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`.

11.20. **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.

11.21. **inversion, simplified-inversion.** In

`(inversion` *x . imp-formulas*`)`

it is assumed 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.

11.22. **coind.** Recall that $J(\vec{r})$ with a coinductively defined predicate $J$ provides

(i) a type substitution,
(ii) a predicate instantiation, and
(iii) the list $\vec{r}$ of argument terms.

`(coind` *hyp*`)` expects a goal $J(\vec{r})$ with a coinductively defined predicate $J$, and *hyp* is expected to be

(i) a number or string identifying a hypothesis $A(\vec{r})$ form the context;
(ii) the name of a global assumption or theorem $A(\vec{r})$;
(iii) a closed proof of a formula $A(\vec{r})$;

(iv) a formula $A(\vec{r})$ with free variables from the context, generating a new goal.

`(coind)` expects an inst-imp-formula $A(\vec{r}) \to J\vec{r}$ as goal. Then the greatest-fixed-point axiom for $J$ is used: $P\vec{x} \to \forall_{\vec{x}}(P\vec{x} \to C(P)) \to J\vec{x}$ with $C(J)$ the defining clause for $J$. Substitute $\{\, \vec{x} \mid A(\vec{x}) \,\}$ for $P$, and use $A(\vec{x}) \to \forall_{\vec{x}}(A(\vec{x}) \to C(\{\, \vec{x} \mid A(\vec{x}) \,\})) \to J\vec{x}$ (i.e., its universal closure). After an appropriate application ($\vec{r}$ for $\vec{x}$) we are left with a new goal saying that $\{\, \vec{x} \mid A(\vec{x}) \,\}$ satisfies the defining clause for $J$.

In case of simultaneous coinductively defined predicates we can provide other imp-formulas to be proved simultaneously with the given one. Then their clauses are generated as new goals.

11.23. **ex-intro.** In `(ex-intro term)`, the user provides a term to be used for the present (existential) goal.

11.24. **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.

11.25. **by-assume.** Suppose we prove a goal from an existential formula $\exists_x A$, $\exists^{\mathrm{d}}_x A$, $\exists^{\mathrm{r}}_x A$, $\exists^{\mathrm{l}}_x A$, $\exists^{\mathrm{u}}_x A$ or $\tilde{\exists}_{x_1,\dots,x_n}(A_1 \tilde{\wedge} \dots \tilde{\wedge} A_m)$. The natural way to use this hypothesis is to say "by the existential hypothesis assume we have an $x$ satisfying $A$" or "by ... assume we have $x_1, \dots, x_n$ satisfying $A_1, \dots, A_m$". Correspondingly we provide `(by-assume 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 $A$.

Example (introducing abbreviations). Suppose that in a proof we want to abbreviate a (complex) term $t$ by a variable $x$. Then do

```
(assert "ex x x=t")
 (ex-intro (pt "t"))
 (use "Truth")
(assume "ExHyp")
(by-assume "ExHyp" "x" "x-Def")
```

Now we have $x$ and $x$-Def: $x = t$ in the context, and can work with $x$ rather than the term $t$.

11.26. **cases.** `(cases)` expects a goal formula $\forall_x A$ with $x$ of an algebra type and total. Let $c_1, \dots, c_n$ be the constructors of the algebra. Then $n$ new goals $\forall_{\vec{x}_i} A(c_i \vec{x}_i)$ are generated. `(cases t)` expects a goal $A(t)$ with $t$ a total term. If $t$ is a boolean term, the goal $A(t)$ is replaced by the two new goals $\mathrm{atom}(t) \to A(\mathtt{tt})$ and $(\mathrm{atom}(t) \to \mathbf{F}) \to A(\mathtt{ff})$. If $t$ is a total non-boolean term, `cases` is called with the all-formula $\forall_x(x = t \to A(x))$.

`(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.

11.27. **casedist.** `(casedist t)` replaces the goal $A$ containing a boolean term $t$ by two new goals $\mathrm{atom}(t) \to A(\mathtt{tt})$ and $(\mathrm{atom}(t) \to \mathbf{F}) \to A(\mathtt{ff})$.

11.28. **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 $s$. Example: for `f` of type `nat=>boole` consider the situation

```
  f  n  m  EqHyp:n=m
  fHyp:f n
----------------------------
?_2:[if (f m) False True]=f n
```

Then the command

```
(simp "EqHyp")
```

generates the a new goal with `n` replaced by `m`:

```
[if (f m) False True]=f m
```

We can also change the direction: the command

```
(simp "<-" "EqHyp")
```

generates the a new goal with `m` replaced by `n`:

```
[if (f n) False True]=f n
```

11.29. **simp-with.** This is a more verbose form of `simp`, where the terms are not inferred via matching, but have to be given explicitly. In fact, `simp` is defined via `simp-with`. 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` (except that the formula of $x$ must be an atom, a negated atom or $t \approx s$), 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.

11.30. **simphyp, simphyp-to.** `simphyp` does for forward chaining the same as `simp` 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` *hyp opt-dir x . elab-path-and-terms*`)`, *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.

`simphyp-to` expects a string as its last argument, which is used (via `name-hyp`) to name the newly introduced simplified hypothesis. Example: if in the situation in 11.28 above we type

```
(simphyp-to "fHyp" "EqHyp" "fHypSimp")
```

we obtain as new goal

```
  f  n  m  EqHyp:n=m
  fHyp:f n
  fHypSimp:f m
----------------------------
?_2:[if (f m) False True]=f n
```

11.31. **simphyp-with, simphyp-with-to.** `simphyp-with` is a more verbose form of `simphyp`, where the terms are not inferred via matching, but have to be given explicitly. `simphyp-with-to` again expects a string as

its last argument, to be used as name for the newly introduced simplified hypothesis.

11.32. **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.

11.33. **by-assume-minimal-wrt.** For convenience in classical arguments there is `(by-assume-minimal-wrt 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 $\alpha_1 \to \ldots \to \alpha_n \to \mathbf{N}$.

11.34. **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`

11.35. **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`.

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

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

11.38. **search.** We provide a proof search tool `search` based on Huet's [17] unification algorithm for the simply typed lambda calculus; its underlying theory is explained in 12. `(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. One can trace `search` by setting `VERBOSE-SEARCH` to true.

11.39. **auto.** It can be convenient to automate (the easy cases of an) interactive proof development by iterating `search` as long as it is successful in finding a proof. Then the first goal where it failed is presented as the new goal. `(auto` *`m (name1 m1) ...`*`)` takes the same arguments as `search`.

11.40. **prop.** `prop` searches for a proof of the current goal in minimal propositional logic. In particular it provides easy access to the axiom of truth for proving $T$ and to ex-falso-quodlibed and proof-by-contradiction. The search meachnism is based on work of Hudelmaier [15, 19, 16] and Dyckhoff [11]. If the search does not succeed, the same is done for intuitionistic propositional logic (by adding ex-falso-quodlibet assumptions to the context). If it does not succeed again, it does the same for classical propositional logic (by adding stability assumptions to the context).

11.41. **efproof.** `efproof` constructs a proof of the present goal from falsity `F`, which must be part of the context.

11.42. **def, defnc.** `def` is used to introduce an abbreviation of a term $t$ by a variable $x$ of the same type. The syntax is that def is called with two strings, the first for the variable and the second for the term. This adds a context item `x=t` with name `xDef`. `defnc` does the same, with $x$ a non-computational variable.

## 12. Unification and proof search

We describe a proof search method suitable for minimal logic with higher order functionals. It is based on Huet's [17] 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 [30].

12.1. **Huet's unification algorithm.** We work in the simply typed $\lambda$-calculus, with the usual conventions. For instance, whenever we write a term we assume that it is correctly typed. *Substitutions* are denoted by $\varphi, \psi, \rho$. The result of applying a substitution $\varphi$ 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 $\varphi$ 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 [17]) 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* $\xi$, 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 $\rho$ 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 $\rho$ 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 $\varphi'$ the substitution $(\rho \circ \varphi')\restriction Q_\exists$ is a $\mathcal{U}$-solution, and*
   (ii) *(completeness) for every $\mathcal{U}$-solution $\varphi$ there is a transition $\mathcal{U} \Longrightarrow_\rho \mathcal{U}'$ and $\mathcal{U}'$-solution $\varphi'$ such that $\varphi = (\rho \circ \varphi')\restriction Q_\exists$, and moreover $\mu(\varphi') \le \mu(\varphi)$ 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 $\varphi$.*

**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 $(Q', \rho)$ with a "transition" substitution $\rho$ and such that for any $Q'$-substitution $\varphi'$, $(\rho \circ \varphi')\restriction Q_\exists$ 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.*

12.2. **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.

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

Explicitly, a formula $A$ is a *$Q$-goal* iff all its free variables are in $Q$, and for every subterm $y\vec{r}$ of $A$ with $y$ either existentially bound in $A$ (with $y\vec{r}$ in the scope) or else free in $A$ and flexible in $Q$, the $\vec{r}$ are distinct variables either $\lambda$- 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 $\varphi$ 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 $\rho$ and another unification problem $\mathcal{U}' = Q'C'$. Note that $\rho$ will be neither a $Q$-substitution nor a $Q'$-substitution, but will have the property that

(a) $\rho$ 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 $\varphi'$ 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* $\xi$, 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 $\rho$ 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 ($\rho$ 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*

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

*is well-defined and we have* $\Phi\colon \mathcal{U}$*-solutions* $\to \mathcal{U}'$*-solutions such that* $\Phi'$ *is inverse to* $\Phi$*, 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* $(Q', \rho)$ *with a "transition" substitution* $\rho$ *and a prefix* $Q'$ *(i.e., a unification problem* $\mathcal{U}'$ *with no unification pairs) such that for any* $Q'$*-substitution* $\varphi'$*,* $(\rho \circ \varphi')\restriction Q_\exists$ *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.*

12.3. **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* $\varphi$*:*

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

*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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $(Q', \rho)$ of either Huet's or the pattern unification algorithm applied to $Q^*(\vec{r} = \vec{s}^*)$ and a $Q'$-substitution $\varphi'$ 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 $(Q', \rho)$ be such a result, and assume that $\varphi'$ 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 $\varphi$ 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 $\varphi$, 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., $\vartheta^*$ 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 $\varphi'$ 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 $[x_1, \ldots, x_n := X_1\vec{z}, \ldots, X_n\vec{z}]$, 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 $\bigwedge\!\!\!\bigwedge \mathcal{P} \to P\vec{r}$ 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 $\varphi$ we have: $\varphi$ is a $Q$-substitution satisfying $\vdash S\varphi$ iff there is a prefix $Q'$, a substitution $\rho$ and a $Q'$-substitution $\varphi'$ 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.

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

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

12.4. **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. [25]). 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 $(\texttt{left})$, $(\texttt{right}, \texttt{left})$ and $(\texttt{right}, \texttt{right})$. 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[2]. 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 12.3. 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.

[2]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.

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

12.5. **Implementation.** Following Miller [25], Berger and [30], 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 $\varphi$ and proofs of the $\varphi$-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 $n$ ($\geq 0$) new goals with their clauses (and also all remaining sequents) are substituted with $\texttt{star} \circ \rho$, where `star` is the "raising" substitution and $\rho$ is the most general unificator. For this constellation `intro-search` is called again. In case of success, one obtains a $Q'$-substitution $\varphi'$ and proofs of the $\texttt{star} \circ \rho \circ \varphi'$ -substituted new sequents. Let $\varphi := (\rho \circ \varphi')\restriction Q_\exists$, and take the first $n$ proofs of these to build a proof of the $\varphi$-substituted (first) sequent originally considered by `intro-search`.

12.6. **Notes.** The present treatment benefitted from a presentation of Miller's [25] 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 [25]; it has been called *patterns* by Nipkow [26]. 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 [25], 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 [26] already had presented a version of Miller's pattern unification algorithm for such prefixes, and Miller in [25, 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 [25, 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. 12.4). The latter can be avoided in principle, but of course is a useful thing to have.

## 13. Extracted terms

13.1. **The type of a formula.** We assign to every formula $A$ an object $\tau(A)$, a type or the nulltype symbol (written $\circ$ in text and displayed `eps` in Minlog). $\tau(A)$ 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`*`)`,

`(idpreconst-to-et-type` *`idpc`*`)`.

Both are defined simultaneously; this makes sense, since the clauses and also the comprehension terms of an idpredconst are prior to the idpredconst.

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`.

We also define separately

`(formula-of-nulltype?` *`formula`*`)`,

since this test can be done more efficiently.

13.2. **Extracted terms.** We can define, for a given derivation $M$ of a formula $A$ with $\tau(A) \neq \circ$, 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-or-thm-name`*`)`.

Hence `proof-to-extracted-term` gets either a proof or else a theorem name. In the former case it works its way through the proof, until it comes to an assumption variable, an axiom, a theorem or a global assumption. When it is a theorem, `theorem-to-extracted-term` is called. This also happens in when a theorem name is the input. `theorem-to-extracted-term` applies as its default operation `theorem-or-global-assumption-to-pconst`, where in case of a lemma `L` the pconst has name `cL`.

When we want to execute the program, we have to replace the constant `cL` corresponding to 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).
```

The constant `cL` will unfold under normalization if the lemma is animated. However, in some cases `theorem-to-extracted-term` directly gives short and meaningful terms:

| | |
|---|---|
| `InhabTotal` | $\mapsto$ (`Inhab rho`), |
| `AllAllPartial`, `AllPartialAll` | $\mapsto$ [`x`]`x`, |
| `ExExPartial`, `ExPartialEx` | $\mapsto$ [`x`]`x`, |
| `Pconst` + `Total` | $\mapsto$ `pconst`, |
| `Pconst` + `STotal` | $\mapsto$ the extract from the proof, |
| `AlgEqTotal` | $\mapsto$ [`n`, `m`]`n` = `m`, |
| `BooleIfTotal` | $\mapsto$ [`free`][`if test arg1 arg2`], |
| `EqDCompat`, `EqDCompatRev` | $\mapsto$ [`x`]`x`, |
| `Id` | $\mapsto$ [`x`]`x`, if `unfold-let-flag` is true. |

Here is an example. It is easy to prove `NatEqTotal`:

$$\forall^{\mathrm{nc}}_{\hat{n}}(T_{\mathbf{N}}\hat{n} \to \forall^{\mathrm{nc}}_{\hat{m}}(T_{\mathbf{N}}\hat{m} \to T_{\mathbf{B}}(\hat{n} = \hat{m})))$$

Since the proof is by induction (or rather elimination for `TotalNat`), the extracted term will involve recursion:

```
[n](Rec nat=>nat=>boole)n([n0][if n0 True ([n1]False)])
   ([n0,(nat=>boole),n1][if n1 False (nat=>boole)])
```

However, we can prove that $\lambda_{n,m}(n = m)$ realizes the formula as well:

```
;; NatEqTotalSound
(set-goal (real-and-formula-to-mr-formula
   (pt "[n,m]n=m")
   (proof-to-formula (theorem-name-to-proof "NatEqTotal"))))
(assume "n^" "n^0" "TMRn0n")
(elim "TMRn0n")
```

```
(assume "m^" "m^0" "TMRm0m")
(elim "TMRm0m")
(use "TotalBooleTrueMR")
(assume "m^1" "m^10" "Useless1" "Useless2")
(use "TotalBooleFalseMR")
(assume "m^" "m^0" "Useless1" "IH" "m^1" "m^10" "TMRm10m1")
(elim "TMRm10m1")
(use "TotalBooleFalseMR")
(assume "m^2" "m^20" "TMRm20m2" "Useless2")
(use "IH")
(use "TMRm20m2")
;; Proof finished.
(save "NatEqTotalSound")
```

Hence we are allowed to change the extracted term of `NatEqTotal` into $\lambda_{n,m}(n = m)$.

Generally, `proof-to-soundness-proof` at `FinAlgEqTotal` looks for a therem with name `FinAlgEqTotalSound` and uses it. An error is raised if `FinAlgEqTotalSound` does not exist.

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

| **Explanation** | **Symbol** | **Minlog's output** |
|---|---|---|
| $\lambda$-abstraction | $\lambda_x M$ | `[x]M` |
| pair | $\langle M, N\rangle$ | `M pair N` |
| left element of a pair | $(M\ 0)$ | `lft M` |
| right element of a pair | $(M\ 1)$ | `rht M` |
| left embedding into a sum type $\rho + \sigma$ | $\mathrm{Inl}_{\rho\sigma} M$ | `(InL rho sigma)M` |
| right embedding into a sum type $\rho + \sigma$ | $\mathrm{Inr}_{\sigma\rho} M$ | `(InR sigma rho)M` |
| recursion operator | $\mathcal{R}$ | `Rec` |
| corecursion operator | ${}^{\mathrm{co}}\mathcal{R}$ | `CoRec` |
| arrow for types | $\to$ | `=>` |
| product for types | $\times$ | `yprod` |
| sum for types | $+$ | `ysum` |
| primitive pair | $\langle M, N\rangle$ | `M@N` |
| left element of a primitive pair | $(M\ 0)$ | `left M` |
| right element of a primitive pair | $(M\ 1)$ | `right M` |
| primitive product for types | $\times$ | `@@` |

13.3. **Soundness.** One can prove that every theorem in TCF + $\mathrm{Ax}_{\mathrm{nci}}$ has a realizer: the extracted term of its proof. Here ($\mathrm{Ax}_{\mathrm{nci}}$) is an arbitrary set of non-computational invariant formulas viewed as axioms.

**Theorem** (Soundness)**.** *Let $M$ be a derivation of $A$ from assumptions $u_i\colon C_i$ $(i < n)$. Then we can derive* $\mathrm{et}(M)$ $\mathbf{r}$ $A$ *from assumptions $x_{u_i}$* $\mathbf{r}$ $C_i$ *(with $x_{u_i} := \varepsilon$ in case $C_i$ is n.c.).*

The proof is by induction on $M$, and can be traced back to early work of Kleene, Kreisel and Troelstra. References and a detailed exposition close to the present terminology can be found in [35, 7.2.8].

We clearly want that `proof-to-soundness-proof` does not unfold the auxiliary propositions used in the proof. Let a theorem `Thm` (thought of as coming with its proof $M$) prove a formula $A$. Then we should have `ThmSound` proving $\mathrm{et}(M)$ $\mathbf{r}$ $A$ in our proof library. When `proof-to-soundness-proof` arrives at `Thm`, it inserts `ThmSound`. There is no circularity here, since $t$ $\mathbf{r}$ $A$ is invariant, i.e., $\varepsilon$ $\mathbf{r}$ ($t$ $\mathbf{r}$ $A$) is the formula $t$ $\mathbf{r}$ $A$ itself (cf. [35, 7.2.4]).

In the special case of a theorem `PconstTotal` proving totality of the constant `Pconst` the extracted term is `Pconst` rather than `cPconstTotal` (which when animated would unfold into a term with recursion operators). Then `PconstTotalSound` proves `Pconst` **r** `PconstTotal`, with a simple standard proof.

An internal proof of soundness can be generated by calling

```
(proof-to-soundness-proof proof-or-thm-name).
```

This uses the auxiliary functions

```
(axiom-to-soundness-proof aconst),
(theorem-to-soundness-proof aconst),
(global-assumption-to-soundness-proof aconst).
```

## 14. Computational content of classical proofs

14.1. **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 [13] and Dragalin [10], 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

$$\vec{D} \to D_i[\bot := \exists_y G], \tag{10}$$

$$G[\bot := \exists_y G] \to \exists_y G. \tag{11}$$

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 (10) allows the substitution in $\vec{D}$ to be dropped, and by (11) 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 (10) and (11) hold.

This section is based on [4] and particularly [35, 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

$$
\begin{array}{ll}
D^{\mathbf{F}} \to D, & \\
G \to (G^{\mathbf{F}} \to \bot) \to \bot, & \\
((R^{\mathbf{F}} \to \mathbf{F}) \to \bot) \to R & \text{for } R \text{ relevant and definite,} \\
I \to I^{\mathbf{F}} & \text{for } I \text{ irrelevant and goal.}
\end{array}
$$

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).
```

By

```
(atr-min-excl-proof-to-structured-extracted-term
```

```
min-excl-proof . realizers-for-nondefinite-formulas)
```

we can then extract a term $r$ such that $\vec{A} \to \vec{D} \to \vec{G}^{\mathbf{F}}[y := r]$ (if $\vec{y} = y$).

One can test with `min-excl-formula?` whether a given formula indeed is 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 13 for an interpretation of the symbols of the extracted terms in Minlog's output.

14.2. **Gödel's Dialectica interpretation.** In his original functional interpretation [14], 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 [14] Gödel referred to a Hilbert-style proof calculus. However, since the realizers will be formed in a $\lambda$-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 $A$.

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

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.

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

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 $A$.

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

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* $\mathrm{HA}^\omega$ 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 EqD, 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 $\mathrm{HA}^\omega$ 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.

## 15. 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.

15.1. **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).`

15.2. **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 $\rho$ and $\sigma$ are types, then $\rho$`=>`$\sigma$ is a type (function type) and $\rho$`@@`$\sigma$ is a type (primitive pair 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[3]). 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 general (or partial)xs 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 white-space 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.

[3]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

(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; for instance, `(Rec <typelist>)` is the recursion over the types given in the type list.

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 `=` as described above, the binding operator `lambda`, and the primitive 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 `all`, `ex`, `allnc`, `excl`, `exca`, and `excu`.

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), `->` (implication) and `-->` (non-computational 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.

## 16. Natural numbers

For the algebra **N** of natural numbers there is a library file `nat.scm` collecting definitions of some standard functions and relations referring to the type **N**. These are

```
(add-program-constant "NatPlus" (py "nat=>nat=>nat"))
(add-program-constant "NatTimes" (py "nat=>nat=>nat"))
(add-program-constant "NatLt" (py "nat=>nat=>boole"))
(add-program-constant "NatLe" (py "nat=>nat=>boole"))
(add-program-constant "Pred" (py "nat=>nat"))
(add-program-constant "NatMinus" (py "nat=>nat=>nat"))
(add-program-constant "NatMax" (py "nat=>nat=>nat"))
(add-program-constant "NatMin" (py "nat=>nat=>nat"))
(add-program-constant "AllBNat" (py "nat=>(nat=>boole)=>boole"))
(add-program-constant "ExBNat" (py "nat=>(nat=>boole)=>boole"))
(add-program-constant "NatLeast" (py "nat=>(nat=>boole)=>nat"))
(add-program-constant "NatLeastUp" (py "nat=>nat=>(nat=>boole)=>nat"))
```

with there standard (mostly infix) notations. To see their defining equations (i.e., computation rules) and proved equations used as (left to right) rewrite rules one can type for instance

```
(display-pconst "NatPlus")
```

and obtains

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

```
nat1+(nat2+nat3) nat1+nat2+nat3
```

The file also contains often used theorems and their proofs. To review them the `search-about` command is recommended. To find out about what has been proved about a particular constant use

```
(search-about "NatMax")
```

The result contains

```
NatMaxLUB
all nat1,nat2,nat3(nat1<=nat3 -> nat2<=nat3 -> nat1 max nat2<=nat3)
NatMaxUB2
all nat1,nat2 nat2<=nat1 max nat2
NatMaxUB1
all nat1,nat2 nat1<=nat1 max nat2
NatMaxComm
all nat1,nat2 nat1 max nat2=nat2 max nat1
```

To see which theorems relating to case distinctions are available type

```
(search-about "Cases")
```

One obtains the following theorems with their names:

```
NatLeLtCases
all nat1,nat2((nat1<=nat2 -> Pvar) ->
              (nat2<nat1 -> Pvar) -> Pvar)
NatLeCases
all nat1,nat2(nat1<=nat2 ->
              (nat1<nat2 -> Pvar) ->
              (nat1=nat2 -> Pvar) -> Pvar)
NatLtSuccCases
all nat1,nat2(nat1<Succ nat2 ->
              (nat1<nat2 -> Pvar) ->
              (nat1=nat2 -> Pvar) -> Pvar)
```

As another example, to check the available transitivity theorems type

```
(search-about "Trans")
```

The result is

```
NatLtLtSuccTrans
all nat1,nat2,nat3(nat1<nat2 -> nat2<Succ nat3 -> nat1<nat3)
NatLeLtTrans
all nat1,nat2,nat3(nat1<=nat2 -> nat2<nat3 -> nat1<nat3)
NatLtLeTrans
all nat1,nat2,nat3(nat1<nat2 -> nat2<=nat3 -> nat1<nat3)
NatLeTrans
all nat1,nat2,nat3(nat1<=nat2 -> nat2<=nat3 -> nat1<=nat3)
```

```
NatLtTrans
all nat1,nat2,nat3(nat1<nat2 -> nat2<nat3 -> nat1<nat3)
NatEqTrans
all nat1,nat2,nat3(nat1=nat2 -> nat2=nat3 -> nat1=nat3)
```

Yet another example is

```
(search-about "NatLeast")
```

The bounded least number operator `NatLeast`, written $\mu_n g$, is defined recursively as follows. Here $g$ is a variable of type $\mathbb{N} \to \mathbb{B}$.

$$\mu_0 g := 0,$$
$$\mu_{\mathrm{S}n} g := \begin{cases} 0 & \text{if } g0 \\ \mathrm{S}\mu_n(g \circ \mathrm{S}) & \text{otherwise.} \end{cases}$$

Then we obtain

| | |
|---|---|
| `NatLeastBound:` | $\mu_n g \le n$, |
| `NatLeastLeIntro:` | $gm \to \mu_n g \le m$, |
| `NatLeastLtElim:` | $\mu_n g < n \to g(\mu_n g)$, |
| `PropNatLeast:` | $m \le n \to gm \to g(\mu_n g)$. |

From $\mu_n g$ we define

$$\mu^n_{n_0} g := \begin{cases} (\mu_{n-n_0} \lambda_m g(m + n_0)) + n_0 & \text{if } n_0 \le n \\ 0 & \text{otherwise.} \end{cases}$$

Clearly $\mu^n_0 g = \mu_n g$. Generally we have

| | |
|---|---|
| `NatLeastUpLBound:` | $n_0 \le n \to n_0 \le \mu^n_{n_0} g$, |
| `NatLeastUpBound:` | $\mu^n_{n_0} g \le n$, |
| `NatLeastUpLeIntro:` | $n_0 \le m \to gm \to \mu^n_{n_o} g \le m$, |
| `NatLeastUpLtElim:` | $n_0 \le \mu^n_{n_0} g < n \to g(\mu^n_{n_0} g)$. |

## References


1. P. Aczel, H. Simmons, and S.S. Wainer (eds.), *Proof theory. a selection of papers from the leeds proof theory programme 1990*, Cambridge University Press, 1992. 29
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. ———, *From coinductive proofs to exact real arithmetic*, Computer Science Logic (E. Grädel and R. Kahle, eds.), LNCS, Springer Verlag, Berlin, Heidelberg, New York, 2009, pp. 132–146. 4.3
4. Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg, *Refined program extraction from classical proofs*, Annals of Pure and Applied Logic **114** (2002), 3–25. iii, 14.1

5. Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg, *Term rewriting for normalization by evaluation*, Information and Computation **183** (2003), 19–42. 1.1, 2.3, 6.2
6. 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
7. Stefan Berghofer, *Proofs, programs and executable specifications in higher order logic*, Ph.D. thesis, Institut für Informatik, TU München, 2003. 1.6
8. Luca Chiarabini, *Program development by proof transformation*, Ph.D. thesis, Fakultät für Mathematik, Informatik und Statistik der LMU, München, 2009. 10.2
9. Coq Development Team, *The Coq Proof Assistant Reference Manual – Version 8.2*, Inria, 2009. 1.6
10. 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, 14.1
11. Roy Dyckhoff, *Contraction–free sequent calculi for intuitionistic logic*, The Journal of Symbolic Logic **57** (1992), 793–807. 11.40
12. 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
13. 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, 14.1
14. Kurt Gödel, *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts*, Dialectica **12** (1958), 280–287. 1, 4.1, 5.1, 14.2
15. Jörg Hudelmaier, *Bounds for cut elimination in intuitionistic propositional logic*, Ph.D. thesis, Mathematische Fakultät, Eberhard–Karls–Universität Tübingen, 1989. 11.40
16. Jörg Hudelmaier, *An $o(n \log n)$–space decision procedure for intuitionistic propositional logic*, J. Logic Computation **3** (1993), no. 1, 63–75. 11.40
17. Gérard Huet, *A unification algorithm for typed λ-calculus*, Theoretical Computer Science **1** (1975), 27–57. 6.4, 11.38, 12, 12.1
18. 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
19. Jörg Hudelmaier, *Bounds for cut elimination in intuitionistic propositional logic*, Archive for Mathematical Logic **31** (1992), 331–354, Lemma 4 is true (and needed) for atomic $u$ only. 11.40
20. Nils Köpp and Helmut Schwichtenberg, *Lookahead analysis in exact real arithmetic with logical methods*, Theoretical Computer Science **943** (2023), 171–186.
21. Alberto Martelli and Ugo Montanari, *An efficient unification algorithm*, ACM Transactions on Programming Languages and Systems **4** (1982), no. 2, 258–282. 2.2, 6.4
22. 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. 5.4
23. ———, *Intuitionistic type theory*, Bibliopolis, 1984. 1
24. 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

25. 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, 6.4, 12.4, 12.5, 12.6, i, ii
26. 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. 12.6, i
27. Dag Prawitz, *Natural deduction*, Acta Universitatis Stockholmiensis. Stockholm Studies in Philosophy, vol. 3, Almqvist & Wiksell, Stockholm, 1965. 10.2
28. 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. 10.1, 10.1
29. Helmut Schwichtenberg, *Proofs as programs*, in Aczel et al. [1], pp. 81–113. (document)
30. ———, *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. 12, 12.5
31. ———, *Program extraction from proofs: the fan theorem for uniformly coconvex bars*, Mathesis Universalis, Computability and Proof (S. Centrone, S. Negri, D. Sarikaya, and P. Schuster, eds.), Synthese Library, vol. 412, Springer Nature, 2019, pp. 333–341.
32. ———, *Computational aspects of Bishop's constructive mathematics*, Handbook of constructive mathematics (D. Bridges, H. Ishihara, H. Schwichtenberg, and M. Rathjen, eds.), Cambridge University Press, 2023.
33. ———, *Logic for exact real arithmetic: multiplication*, Mathematics for Computation (M4C) (M. Benini, O. Beyersdorff, M. Rathjen, and P. Schuster, eds.), World Scientific, Singapore, 2023, pp. 39–69.
34. Helmut Schwichtenberg, Monika Seisenberger, and Franziskus Wiesnet, *Higman's lemma and its computational content*, Advances in Proof Theory (R. Kahle, T. Strahm, and T. Studer, eds.), Birkhäuser, 2016, pp. 353–375.
35. Helmut Schwichtenberg and Stanley S. Wainer, *Proofs and computations*, Perspectives in Logic, Association for Symbolic Logic and Cambridge University Press, 2012. 1.5, 13.3, 14.1
36. ———, *Tiered arithmetics*, Feferman on Foundations (G. Jäger and W. Sieg, eds.), Outstanding Contributions to Logic, vol. 13, Springer, 2017, pp. 145–168.
37. Helmut Schwichtenberg and Franziskus Wiesnet, *Logic for exact real arithmetic*, Logical Methods in Computer Science **17** (2021), no. 2, arxiv.org/abs/1904.12763.
38. Dana Scott, *Outline of a mathematical theory of computation*, Technical Monograph PRG–2, Oxford University Computing Laboratory, 1970. 1
39. 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
40. Anne S. Troelstra and Helmut Schwichtenberg, *Basic proof theory*, 2nd ed., Cambridge University Press, 2000. 1.5
41. Anne S. Troelstra and Dirk van Dalen, *Constructivism in mathematics. an introduction*, Studies in Logic and the Foundations of Mathematics, vol. 121, 123, North–Holland, Amsterdam, 1988. 1.5, 10.6
42. Franziskus Wiesnet, *Introduction to Minlog*, Proof and Computation (K. Mainzer, P. Schuster, and H. Schwichtenberg, eds.), World Scientific, 2018, pp. 233–288.