Proofs with Feasible Computational Content

Helmut Schwichtenberg

Working material for the Marktoberdorf Summer School, 2007.
Mathematisches Institut der Ludwig-Maximilians-Universität,
Theresienstraße 39, D-80333 München, Germany.
July 9, 2007.

Contents

CHAPTER 1

Minimal Arithmetic in Finite Types

We develop an arithmetic HA^ω in finite types, which is based on decidable predicates and uses the "negative" logical operators $\to$ and $\forall$ only. Hence proofs in HA^ω have no "computational content".

1.1. Gödel's T

Gödel (1958) proposed to extend Hilbert's concept of "finitary methods" to include higher (but still finite) types. This makes it possible to consider definition schemes for functions (like primitive recursion or transfinite recursion) as higher type operators. Moreover, admittance of computable functionals of higher type is a must when we want to capture the computational content of proofs in arithmetic.

1.1.1. Types. A free algebra is given by its *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 infinitely branching, and hence we allow infinitary free algebras.

The freeness of the constructors is expressed by requiring that their ranges are disjoint and that they are injective. To allow for partiality – which is mandatory when we want to deal with computable objects –, we have to embed our algebras into domains. Both requirements together imply that we need "lazy domains".

Our type system is defined by two type forming operations: arrow types $\rho \to \sigma$ and the formation of *inductively generated types* or *base types* $\mu_{\vec{\alpha}}\vec{\kappa}$, where $\vec{\alpha} = (\alpha_j)_{j<N}$ is a list of distinct "type variables", and $\vec{\kappa} = (\kappa_i)_{i<k}$ is a list of "constructor types", whose argument types contain $\alpha_0, \dots, \alpha_{N-1}$ in strictly positive positions only.

For instance, $\mu_\alpha(\alpha, \alpha \to \alpha)$ is the type of natural numbers; here the list $(\alpha, \alpha \to \alpha)$ stands for two generation principles: α for "there is a natural number" (the 0), and $\alpha \to \alpha$ for "for every natural number there is a next one" (its successor).

DEFINITION. Let $\vec{\alpha} = (\alpha_j)_{j<N}$ be a list of distinct type variables. *Types* $\rho, \sigma, \tau, \mu \in \mathrm{Ty}$ and *constructor types* $\kappa \in \mathrm{KT}_{\vec{\alpha}}$ are defined inductively:

$$\frac{\vec{\rho}, \vec{\sigma}_0, \dots, \vec{\sigma}_{n-1} \in \mathrm{Ty}}{\vec{\rho} \to (\vec{\sigma}_\nu \to \alpha_{j_\nu})_{\nu<n} \to \alpha_j \in \mathrm{KT}_{\vec{\alpha}}} \quad (n \geq 0),$$

$$\frac{\kappa_0, \dots, \kappa_{k-1} \in \mathrm{KT}_{\vec{\alpha}}}{(\mu_{\vec{\alpha}}(\kappa_0, \dots, \kappa_{k-1}))_j \in \mathrm{Ty}} \quad (k \geq 1), \qquad \frac{\rho, \sigma \in \mathrm{Ty}}{\rho \to \sigma \in \mathrm{Ty}}.$$

Here $\vec{\rho} \to \sigma$ means $\rho_0 \to \ldots \to \rho_{n-1} \to \sigma$, associated to the right. For a constructor type $\vec{\rho} \to (\vec{\sigma}_\nu \to \alpha_{j_\nu})_{\nu<n} \to \alpha_j$ we call $\vec{\rho}$ the *parameter* argument types and the $\vec{\sigma}_\nu \to \alpha_{j_\nu}$ *recursive* argument types. We require that for every α_j $(j < N)$ there is a *nullary* constructor type κ_{i_j} with value type α_j, each of whose recursive argument types has a value type α_{j_ν} with $j_\nu < j$.

We reserve μ for base types, i.e., types of the form $(\mu_{\vec{\alpha}}(\kappa_0, \dots, \kappa_{k-1}))_j$. The *parameter types* of μ are all parameter argument types of its constructor types $\kappa_0, \dots, \kappa_{k-1}$.

EXAMPLES.

$$\begin{aligned}
\mathbf{U} &:= \mu_\alpha \alpha, \\
\mathbf{B} &:= \mu_\alpha(\alpha, \alpha), \\
\mathbf{N} &:= \mu_\alpha(\alpha, \alpha \to \alpha), \\
\mathbf{L}(\rho) &:= \mu_\alpha(\alpha, \rho \to \alpha \to \alpha), \\
\rho \otimes \sigma &:= \mu_\alpha(\rho \to \sigma \to \alpha), \\
\rho + \sigma &:= \mu_\alpha(\rho \to \alpha, \sigma \to \alpha), \\
(\mathrm{tree}, \mathrm{tlist}) &:= \mu_{\alpha,\beta}(\mathbf{N} \to \alpha, \beta \to \alpha, \beta, \alpha \to \beta \to \beta), \\
\mathrm{bin} &:= \mu_\alpha(\alpha, \alpha \to \alpha \to \alpha), \\
\mathcal{O} &:= \mu_\alpha(\alpha, \alpha \to \alpha, (\mathbf{N} \to \alpha) \to \alpha), \\
\mathcal{T}_0 &:= \mathbf{N}, \\
\mathcal{T}_{n+1} &:= \mu_\alpha(\alpha, (\mathcal{T}_n \to \alpha) \to \alpha).
\end{aligned}$$

Note that there are many equivalent ways to define these types. For instance, we could take $\mathbf{U} + \mathbf{U}$ to be the type of booleans, and $\mathbf{L}(\mathbf{U})$ to be the type of natural numbers.

A type is *finitary* if it is a μ-type with all its parameter types $\vec{\rho}$ finitary, and all its constructor types have recursive argument types of the form α_{j_m} only (so the $\vec{\sigma}_m$ in the general definition are all empty). In the examples above $\mathbf{U}$, $\mathbf{B}$, $\mathbf{N}$, tree, tlist and bin are all finitary, but $\mathcal{O}$ and $\mathcal{T}_{n+1}$ are not. $\mathbf{L}(\rho)$, $\rho \otimes \sigma$ and $\rho + \sigma$ are finitary provided their parameter types are. An argument position in a type is called *finitary* if it is occupied by a finitary type.

1.1.2. Recursion operators. The inductive structure of the types $\vec{\mu} = \mu_{\vec{\alpha}}\vec{\kappa}$ corresponds to two sorts of constants: with the *constructors* $\mathrm{C}_i^{\vec{\mu}} \colon \kappa_i(\vec{\mu})$ we can build elements of a type μ_j, and with the (structural) *recursion operators* $\mathcal{R}_{\mu_j}^{\vec{\mu},\vec{\tau}}$ we can construct mappings recursion on the structure of $\vec{\mu}$.

In order to define the type of the recursion operators w.r.t. $\vec{\mu} = \mu_{\vec{\alpha}}\vec{\kappa}$ and result types $\vec{\tau}$, we first define for

$$\kappa_i = \vec{\rho} \to (\vec{\sigma}_\nu \to \alpha_{j_\nu})_{\nu<n} \to \alpha_j \in \mathrm{KT}_{\vec{\alpha}}$$

the *step type*

$$\delta_i^{\vec{\mu},\vec{\tau}} := \vec{\rho} \to (\vec{\sigma}_\nu \to \mu_{j_\nu})_{\nu<n} \to (\vec{\sigma}_\nu \to \tau_{j_\nu})_{\nu<n} \to \tau_j.$$

Here $\vec{\rho}, (\vec{\sigma}_\nu \to \mu_{j_\nu})_{\nu<n}$ correspond to the *components* of the object of type μ_j under consideration, and $(\vec{\sigma}_\nu \to \tau_{j_\nu})_{\nu<N}$ to the previously defined values. The recursion operator $\mathcal{R}_{\mu_j}^{\vec{\mu},\vec{\tau}}$ has type

$$\mathcal{R}_{\mu_j}^{\vec{\mu},\vec{\tau}} \colon \mu_j \to \delta_0^{\vec{\mu},\vec{\tau}} \to \ldots \to \delta_{k-1}^{\vec{\mu},\vec{\tau}} \to \tau_j$$

(recall that k is the total number of constructors for all types μ_j, $j < N$).

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

For some common base types the constructors have standard names, as follows. We also spell out the type of the recursion operators:

$$\begin{aligned}
&\mathtt{tt}^{\mathbf{B}} := \mathrm{C}_1^{\mathbf{B}}, \quad \mathtt{ff}^{\mathbf{B}} := \mathrm{C}_2^{\mathbf{B}},\\
&\mathcal{R}_{\mathbf{B}}^\tau \colon \mathbf{B} \to \tau \to \tau \to \tau,\\
&0^{\mathbf{N}} := \mathrm{C}_1^{\mathbf{N}}, \quad \mathrm{S}^{\mathbf{N}\to\mathbf{N}} := \mathrm{C}_2^{\mathbf{N}},\\
&\mathcal{R}_{\mathbf{N}}^\tau \colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau,\\
&\mathrm{nil}^{\mathbf{L}(\rho)} := \mathrm{C}_1^{\mathbf{L}(\rho)}, \quad \mathrm{cons}^{\rho\to\mathbf{L}(\rho)\to\mathbf{L}(\rho)} := \mathrm{C}_2^{\mathbf{L}(\rho)},\\
&\mathcal{R}_{\mathbf{L}(\rho)}^\tau \colon \mathbf{L}(\rho) \to \tau \to (\rho \to \mathbf{L}(\rho) \to \tau \to \tau) \to \tau,\\
&\big(\mathrm{inl}_{\rho\sigma}\big)^{\rho\to\rho+\sigma} := \mathrm{C}_1^{\rho+\sigma}, \quad \big(\mathrm{inr}_{\rho\sigma}\big)^{\sigma\to\rho+\sigma} := \mathrm{C}_2^{\rho+\sigma},\\
&\mathcal{R}_{\rho+\sigma}^\tau \colon \rho+\sigma \to (\rho \to \tau) \to (\sigma \to \tau) \to \tau,\\
&\big(\otimes^+_{\rho\sigma}\big)^{\rho\to\sigma\to\rho\otimes\sigma} := \mathrm{C}_1^{\rho\otimes\sigma},\\
&\mathcal{R}_{\rho\otimes\sigma}^\tau \colon \rho\otimes\sigma \to (\rho \to \sigma \to \tau) \to \tau.
\end{aligned}$$

One often writes $x :: l$ as shorthand for $\mathrm{cons}\, x\, l$, and $\langle y, z\rangle$ for $\otimes^+ yz$.

Terms are inductively defined from typed variables x^ρ and the constants, that is, constructors $\mathrm{C}_i^{\vec{\mu}}$ and recursion operators $\mathcal{R}_{\mu_j}^{\vec{\mu},\vec{\tau}}$, by abstraction $\lambda_{x^\rho} M^\sigma$ and application $M^{\rho\to\sigma} N^\rho$.

Examples. We define the *canonical inhabitant* ε^ρ of a type $\rho \in \mathrm{Ty}$:

$$\varepsilon^{\mu_j} := \mathrm{C}_j^{\vec{\mu}} \varepsilon^{\vec{\rho}} (\lambda_{\vec{x}_1} \varepsilon^{\mu_{j_1}}) \dots (\lambda_{\vec{x}_n} \varepsilon^{\mu_{j_n}}), \quad \varepsilon^{\rho\to\sigma} := \lambda_x \varepsilon^\sigma.$$

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

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

The *append*-function :+: for lists is defined recursively by

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

It can be defined as the term

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

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

$$\mathrm{Rev}\ \mathrm{nil} := \mathrm{nil},$$
$$\mathrm{Rev}(x :: l) := (\mathrm{Rev}\ l) :+: (x :: \mathrm{nil});$$

the corresponding term is

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

1.1.3. Conversion. To define the conversion relation, it will be helpful to use the following notation. Let $\vec{\mu} = \mu_{\vec{\alpha}} \vec{\kappa}$ and $\kappa_i =$

$$\rho_0 \to \dots \to \rho_{m-1} \to (\vec{\sigma}_0 \to \alpha_{j_0}) \to \dots \to (\vec{\sigma}_{n-1} \to \alpha_{j_{n-1}}) \to \alpha_j \in \mathrm{KT}_{\vec{\alpha}},$$

and consider $\mathrm{C}_i^{\vec{\mu}} \vec{N}$. We write $\vec{N}^P = N_0^P, \dots, N_{m-1}^P$ for the *parameter arguments* $N_0^{\rho_0}, \dots, N_{m-1}^{\rho_{m-1}}$ and $\vec{N}^R = N_0^R, \dots, N_{n-1}^R$ for the *recursive arguments* $N_m^{\vec{\sigma}_0\to\mu_{j_0}}, \dots, N_{m+n-1}^{\vec{\sigma}_{n-1}\to\mu_{j_{n-1}}}$, and n^R for the number n of recursive arguments.

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

$$(\lambda_x M) N \mapsto M[x := N],$$
$$\lambda_x (M x) \mapsto M \quad \text{if } x \notin \mathrm{FV}(M) \ (M \text{ not an abstraction}),$$
$$\mathcal{R}_j (\mathrm{C}_i^{\vec{\mu}} \vec{N}) \vec{M} \mapsto M_i \vec{N} \big((\mathcal{R}_{j_0} \cdot \vec{M}) \circ N_0^R \big) \dots \big((\mathcal{R}_{j_{n-1}} \cdot \vec{M}) \circ N_{n-1}^R \big).$$

Here we have written $\mathcal{R}_j \cdot \vec{M}$ for $\lambda_{x^{\mu_j}} (\mathcal{R}_{\mu_j}^{\vec{\mu},\vec{\tau}} x^{\mu_j} \vec{M})$.

The *one step reduction relation* $\to$ can now be defined as follows. $M \to N$ if N is obtained from M by replacing a subterm M' in M by N', where $M' \mapsto N'$. The reduction relations $\to^+$ and $\to^*$ are the transitive and the reflexive transitive closure of $\to$, respectively. For $\vec{M} = M_1, \dots, M_n$ we

write $\vec{M} \to \vec{M}'$ if $M_i \to M_i'$ for some $i \in \{1, \dots, n\}$ and $M_j = M_j'$ for all $i \neq j \in \{1, \dots, n\}$. A term M is *normal* (or in *normal form*) if there is no term N such that $M \to N$.

Clearly normal closed terms are of the form $\mathrm{C}_i^{\vec{\mu}} \vec{N}$.

THEOREM. *Every term can be reduced to a normal form.*

1.2. Natural Deduction

We define Heyting Arithmetic HA and its extension HA^ω to a finitely typed language.

1.2.1. Derivations as terms. Recall that we have a decidable equality $=_\mu \colon \mu \to \mu \to \mathbf{B}$, for finitary base types μ. Every *atomic formula* has the form $\mathrm{atom}(r^{\mathbf{B}})$, i.e., is built from a boolean term $r^{\mathbf{B}}$. In particular, there is no need for (logical) falsity $\bot$, since we can take the atomic formula $F := \mathrm{atom}(\mathsf{ff})$ – called *arithmetical falsity* – built from the boolean constant ff instead.

The *formulas* of HA^ω are built from atomic ones by the connectives $\to$ and $\forall$. We define *negation* $\neg A$ by $A \to F$, and the *weak* (or "classical") existential quantifier by

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

We use natural deduction rules: $\to^+$, $\to^-$, $\forall^+$ and $\forall^-$.

It will be convenient to write derivations as terms, where the derived formula is viewed as the type of the term. This representation is known under the name *Curry-Howard correspondence*. From now on we use M, N etc. to range over derivation terms, and r, s etc. for object terms.

We give an inductive definition of derivation terms in Table 1, where for clarity we have written the corresponding derivations to the left. For the universal quantifier $\forall$ there is an introduction rule $\forall^+ x$ and an elimination rule $\forall^-$, whose right premise is the term r to be substituted. The rule $\forall^+ x$ is subject to the standard *(Eigen-) variable condition*: The derivation term M of the premise A should not contain any open assumption with x as a free variable.

1.2.2. Structural induction. The general form of (structural) *induction* over simultaneous free algebras $\vec{\mu} = \mu_{\vec{\alpha}} \vec{\kappa}$, with goal formulas $A_j(x_j^{\mu_j})$ is as follows (cf. 1.1.2). For the constructor type

$$\kappa_i = \vec{\rho} \to (\vec{\sigma}_\nu \to \alpha_{j_\nu})_{\nu<n} \to \alpha_j \in \mathrm{KT}_{\vec{\alpha}}$$

derivation	term
$u\colon A$	u^A
$\dfrac{\begin{array}{c}[u\colon A]\\ \mid M\\ B\end{array}}{A\to B}\to^+ u$	$(\lambda_{u^A} M^B)^{A\to B}$
$\dfrac{\begin{array}{cc}\mid M & \mid N\\ A\to B & A\end{array}}{B}\to^-$	$(M^{A\to B}N^A)^B$
$\dfrac{\begin{array}{c}\mid M\\ A\end{array}}{\forall_x A}\forall^+ x$ (with var.cond.)	$(\lambda_x M^A)^{\forall_x A}$ (with var.cond.)
$\dfrac{\begin{array}{cc}\mid M & \\ \forall_x A(x) & r\end{array}}{A(r)}\forall^-$	$(M^{\forall_x A(x)}r)^{A(r)}$

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

we have the *step formula*

(1.1)
$$D_i := \forall_{y_0^{\rho_0},\dots,y_{m-1}^{\rho_{m-1}},y_m^{\vec\sigma_0\to\mu_{j_0}},\dots,y_{m+n-1}^{\vec\sigma_{n-1}\to\mu_{j_{n-1}}}} \big(\forall_{\vec x^{\vec\sigma_0}} A_{j_0}(y_{m+1}\vec x\,)\to\cdots\to\forall_{\vec x^{\vec\sigma_{n-1}}} A_{j_{n-1}}(y_{m+n-1}\vec x\,)\to A_j(\mathrm{C}_i^{\vec\mu}\vec y)\big).$$

Here $\vec y = y_0^{\rho_0},\dots,y_{m-1}^{\rho_{m-1}},y_m^{\vec\sigma_0\to\mu_{j_0}},\dots,y_{m+n-1}^{\vec\sigma_{n-1}\to\mu_{j_{n-1}}}$ are the *components* of the object $\mathrm{C}_i^{\vec\mu}\vec y$ of type μ_j under consideration, and

$$\forall_{\vec x^{\vec\sigma_0}} A_{j_0}(y_m\vec x\,),\dots,\forall_{\vec x^{\vec\sigma_{n-1}}} A_{j_{n-1}}(y_{m+n-1}\vec x\,)$$

are the hypotheses available when proving the induction step. The induction axiom $\mathrm{Ind}^{\vec{x},\vec{A}}_{\mu_j}$ then proves the universal closure of the formula

$$\forall_{x_j}\big(D_0 \to \cdots \to D_{k-1} \to A_j(x_j^{\mu_j})\big).$$

We will often write $\mathrm{Ind}^{\vec{x},\vec{A}}_{j}$ for $\mathrm{Ind}^{\vec{x},\vec{A}}_{\mu_j}$, and omit the upper indices $\vec{x},\vec{A}$ when they are clear from the context. In case of a non-simultaneous free algebra, i.e., of type $\mu_\alpha\vec{\kappa}$, for $\mathrm{Ind}^{x,A}_{\mu}$ we normally write $\mathrm{Ind}_{x,A}$.

Examples.

$$\begin{aligned}
&\mathrm{Ind}_{p,A}\colon \forall_p\big(A(\mathtt{tt}) \to A(\mathtt{ff}) \to A(p^{\mathbf{B}})\big),\\
&\mathrm{Ind}_{n,A}\colon \forall_m\big(A(0) \to \forall_n(A(n) \to A(\mathrm{S}n)) \to A(m^{\mathbf{N}})\big),\\
&\mathrm{Ind}_{l,A}\colon \forall_l\big(A(\mathrm{nil}) \to \forall_{x,l'}(A(l') \to A(x :: l')) \to A(l^{\mathbf{L}(\rho)})\big),\\
&\mathrm{Ind}_{x,A}\colon \forall_x\big(\forall_{y_1}A(\mathrm{inl}\,y_1) \to \forall_{y_2}A(\mathrm{inr}\,y_2) \to A(x^{\rho_1+\rho_2})\big),\\
&\mathrm{Ind}_{x,A}\colon \forall_x\big(\forall_{y^\rho,z^\sigma}A(\langle y,z\rangle) \to A(x^{\rho\wedge\sigma})\big),
\end{aligned}$$

where $x :: l$ is shorthand for $\mathrm{cons}\,x\,l$ and $\langle y,z\rangle$ for $\otimes^+yz$.

Let HA^ω be the theory based on the axioms above including the induction axioms.

1.2.3. Indirect proofs. We show that our arithmetic is "classical" in the sense that the principle of indirect proof holds. Here we make essential use of the fact that our formulas are built from (decidable) atomic ones by the connectives $\to$ and $\forall$. Recall that negation $\neg A$ and the weak existential quantifier $\tilde{\exists}_x A$ are definable. In the next chapter we will (inductively) define the strong (or "constructive") existential quantifier, which will cause proofs to have computational content. In this richer language the principle of indirect proof does not hold any more.

Lemma (Ex falso quodlibet). $\mathrm{HA}^\omega \vdash F \to A$.

Proof. Induction on A, using $\mathrm{Ind}_{p,\mathrm{atom}(p)}$ in the prime formula case. The details are left as an exercise.. □

The following lemma expresses the *principle of indirect proof*.

Lemma (Stability). $\mathrm{HA}^\omega \vdash \neg\neg A \to A$.

Proof. Induction on A (exercise). □

Lemma (Compatibility). *For finitary μ,*

$$\mathrm{HA}^\omega \vdash x_1 =_\mu x_2 \to A(x_1) \to A(x_2).$$

Proof. Induction on x_1 with a side induction on x_2, using ex falso quodlibet. The details are left as an exercise. □

1.3. Example: List Reversal

As a running example we treat list reversal; the example is taken from Berger (2005). For its formulation in our language we view Rev as a function parameter of type $\mathbf{L}(\mathbf{N}) \to \mathbf{L}(\mathbf{N}) \to \mathbf{B}$, whose properties are axiomatized by

$$\mathrm{Rev}(\mathrm{nil}, \mathrm{nil}), \tag{1.2}$$

$$\mathrm{Rev}(v, w) \to \mathrm{Rev}(v :+: x:, x :: w). \tag{1.3}$$

We use $x :: v$ for the `cons`-operator, and $v :+: w$ for the append function. $x:$ denotes $x :: \mathrm{nil}$, i.e., the singleton list consisting of x. We prove

$$\forall_v \tilde{\exists}_w \mathrm{Rev}(v, w). \tag{1.4}$$

Fix v_0 and assume $\mathtt{Hyp}\colon \forall_w \neg\mathrm{Rev}(v_0, w)$; we need to derive a contradiction. To this end we prove that all initial segments of v are non-revertible, which contradicts (1.2). More precisely, we prove

$$\forall_v \big(v :+: u = v_0 \to \forall_w \neg\mathrm{Rev}(v, w)\big),$$

by induction on u. For $u = \mathrm{nil}$ this follows from our initial assumption. For the step case, assume $v :+: (x :: u) = v_0$, fix w and assume further $\mathrm{Rev}(v, w)$. We need to derive a contradiction. Properties of the append function (e.g., associativity) imply that $(v :+: x:) :+: u = v_0$. The IH for $v :+: x:$ gives $\forall_w \neg\mathrm{Rev}(v :+: x:, w)$. Now (1.3) yields the desired contradiction.

Does this proof of (1.4) have computational content? On the face of it, no, because in our "negative" arithmetic HA^ω there is no computational content at all. However, if we look closer, we *can* find computational content, in fact by two different methods. The first "direct" one is dicussed now, the second one in the next chapter, because it needs the (constructive) existential quantifier, which will be introduced only there.

The "direct" method makes use of the fact that proofs in natural deduction can be brought into "normal form", which does not make "detours": all assumptions are applied until an atomic formula is reached, and then built up again. Moreover, we can assume that our proof of the closed formula (1.4) from closed assumptions has no free variables.

Let M be this normal proof, for $v_0 := 2 :: 5:$. Consider all occurrences of the "false" assumption

$$\mathtt{Hyp}\colon \forall_w \neg\mathrm{Rev}(v_0, w).$$

Each of them must be applied to a closed list w_i, followed by a proof of $\mathrm{Rev}(v_0, w_i)$. If we take an uppermost occurrence of our false assumption `Hyp`, we obtain a proof of $\mathrm{Rev}(v_0, w_i)$ *not* containing this assumption. Hence $\mathrm{Rev}(v_0, w_i)$ is true, and we can read off the desired reversal w_i of our list v_0.

CHAPTER 2

Inductive Constructions

We now add computational content to our arithmetic, in the form of inductively defined predicates. It turns out that this is the only addition necessary: for example, the existential quantifier $\exists$ will be a special case of such an inductively defined predicate.

2.1. Inductively Defined Predicates

We study the concept of "computational content" of a proof. This only makes sense after we have introduced inductively defined predicates to our "negative" language of HA^ω involving $\forall$ and $\to$ only. The resulting system will be called *arithmetic with inductively defined predicates* ID^ω.

The intended meaning of an inductively defined predicate I is quite clear: the clauses correspond to constructors of an appropriate algebra μ (or better μ_I). We associate to I a new predicate $I^{\mathbf{r}}$, of arity $(\mu, \vec{\rho})$, where the first argument r of type μ represents a "generation tree", witnessing how the other arguments $\vec{r}$ were put into I. This object r of type μ is called a "realizer" of the prime formula $I(\vec{r})$.

Moreover, we want to be able to select relevant parts of the complete computational content of a proof. This will be possible if some "uniformities" hold; we express this fact by using a uniform variant $\forall^{\mathsf{U}}$ of the universal quantifier $\forall$ (as done by Berger (2005)) and in addition a uniform variant $\to^{\mathsf{U}}$ of implication $\to$. Both are governed by the same rules as the non-uniform ones. However, we will have to put some uniformity conditions on a proof to ensure that the extracted computational content will be correct.

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

More precisely, we allow the formation of inductively generated predicates $\mu_{\vec{X}}\vec{K}$, where $\vec{X} = (X_j)_{j<N}$ is a list of distinct predicate variables, and $\vec{K} = (K_i)_{i<k}$ is a list of constructor formulas (or "clauses") whoses premises contain $X_0, \dots, X_{N-1}$ in strictly positive positions only.

2.1.1. Introduction and elimination axioms.

DEFINITION (Inductively defined predicates). Let $\vec{X} = (X_j)_{j<N}$ be a list of distinct predicate variables. *Formulas* $A, B, C, D \in \mathrm{F}$, *predicates* $P, Q, I \in \mathrm{Preds}$ and *constructor formulas* (or *clauses*) $K \in \mathrm{KF}_{\vec{X}}$ are defined inductively as follows. Let $\breve{\forall}$ denote either $\forall$ or $\forall^{\mathsf{U}}$, and $\breve{\to}$ denote either $\to$ or $\to^{\mathsf{U}}$.

$$\frac{\vec{A}, \vec{B}_0, \dots, \vec{B}_{n-1} \in \mathrm{F}}{\breve{\forall}_{\vec{x}}\big(\vec{A} \mathrel{\breve{\to}} \big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} X_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \to X_j(\vec{t}\,)\big) \in \mathrm{KF}_{\vec{X}}} \quad (n \geq 0)$$

$$\frac{K_0, \dots, K_{k-1} \in \mathrm{KF}_{\vec{X}} \quad (k \geq 1)}{(\mu_{\vec{X}}(K_0, \dots, K_{k-1}))_j \in \mathrm{Preds}} \qquad \frac{P \in \mathrm{Preds}}{P(\vec{r}\,) \in \mathrm{F}} \qquad \frac{C \in \mathrm{F}}{\{\, \vec{x} \mid C \,\} \in \mathrm{Preds}}$$

$$\frac{A, B \in \mathrm{F}}{A \to B \in \mathrm{F}} \quad \frac{A \in \mathrm{F}}{\forall_{x^\rho} A \in \mathrm{F}} \quad \frac{A, B \in \mathrm{F}}{A \to^{\mathsf{U}} B \in \mathrm{F}} \quad \frac{A \in \mathrm{F}}{\forall^{\mathsf{U}}_{x^\rho} A \in \mathrm{F}} \quad \mathrm{atom}(r) \in \mathrm{F}.$$

Here $\vec{A} \mathrel{\breve{\to}} B$ means $A_0 \mathrel{\breve{\to}} \dots \mathrel{\breve{\to}} A_{n-1} \mathrel{\breve{\to}} B$, associated to the right. For a constructor formula $\breve{\forall}_{\vec{x}}(\vec{A} \mathrel{\breve{\to}} (\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} X_{j_\nu}(\vec{s}_\nu)))_{\nu<n} \mathrel{\breve{\to}} X_j(\vec{t}\,))$ we call $\vec{A}$ the *parameter* premises and the $\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} X_{j_\nu}(\vec{s}_\nu))$ *recursive* premises. We require that for every X_j $(j < N)$ there is a clause K_{i_j} with final conclusion $X_j(\vec{t}\,)$, amongst whose premises there is either a parameter premise or else a recursive premise with final conclusion $X_{j_\nu}(\vec{s}_\nu)$ with $j_\nu < j$. (The presence of such clauses guarantees that we can derive ex-falso-quodlibet for every inductively defined predicate I). A clause of the form $\forall^{\mathsf{U}}_{\vec{x}}(F \to X_j(\vec{x}\,))$ is called an *efq-clause*.

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 $C(\vec{r}\,)$. I will be used for predicates of the form $(\mu_{\vec{X}}\,(K_0, \dots, K_{k-1}))_j$.

Consider inductively defined predicates $\vec{I} := \mu_{\vec{X}}(K_0, \dots K_{k-1})$. For each of the k clauses we have an introduction axiom, as follows. Let the i-th clause for I_j be

$$K(\vec{x}\,) := \breve{\forall}_{\vec{x}}\big(\vec{A} \mathrel{\breve{\to}} \big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} X_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \to X_j(\vec{t}\,)\big).$$

The corresponding *introduction axiom* then is $K(\vec{I}\,)$, that is

$$(I_j)^+_i \colon \breve{\forall}_{\vec{x}}\big(\vec{A} \mathrel{\breve{\to}} \big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} I_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \to I_j(\vec{t}\,)\big). \tag{2.1}$$

Also for every I_j we have the *elimination axiom*

$$\forall^{\mathsf{U}}_{\vec{x}}\big(I_j(\vec{x}\,) \to K_0(\vec{P}) \to \dots \to K_{k-1}(\vec{P}) \to P_j(\vec{x}\,)\big).$$

However, in applications one often wants to use a strengthened form of the elimination axioms. For their formulation it is useful to introduce the notation

$$K(\vec{Q}, \vec{P}) := \breve{\forall}_{\vec{x}}\big(\vec{A} \mathrel{\breve{\to}} \big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} Q_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \to$$

$$\big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} P_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \to P_j(\vec{t}\,)\big).$$

Then the *strengthened elimination axioms* are

$$I_j^- : \forall_{\vec{x}}^{\mathsf{U}}\big(I_j(\vec{x}\,) \to K_0(\vec{I},\vec{P}) \to \cdots \to K_{k-1}(\vec{I},\vec{P}) \to P_j(\vec{x}\,)\big). \tag{2.2}$$

They are indeed stronger (and hence easier to use), since each premise $K_i(\vec{I},\vec{P})$ is weaker than $K_i(\vec{P})$ (because $K_i(\vec{I},\vec{P})$ has more premises than $K_i(\vec{P})$). However, there is no essential difference, because they are derivable from the (ordinary) elimination axioms.

2.1.2. Examples. The following inductive definitions of the existential quantifier, conjunction, falsity, equality and disjunction have been used by Martin-Löf (1971).

Existential quantifier. Let α be a type variable, y an object variable of type α, and $\hat{Q}$ a predicate variable of arity (α). We have four variants, depending on where we require uniformity.

$$\begin{aligned}
\mathrm{Ex}(\alpha,\hat{Q}) &:= \mu_X\big(\forall_y(\hat{Q}(y) \to X)\big),\\
\mathrm{ExL}(\alpha,\hat{Q}) &:= \mu_X\big(\forall_y(\hat{Q}(y) \to^{\mathsf{U}} X)\big),\\
\mathrm{ExR}(\alpha,\hat{Q}) &:= \mu_X\big(\forall_y^{\mathsf{U}}(\hat{Q}(y) \to X)\big),\\
\mathrm{ExU}(\alpha,\hat{Q}) &:= \mu_X\big(\forall_y^{\mathsf{U}}(\hat{Q}(y) \to^{\mathsf{U}} X)\big).
\end{aligned}$$

The introduction axioms are

$$\begin{aligned}
\exists^+ &: \quad \forall_x(A \to \exists_x A),\\
(\exists^{\mathsf{L}})^+ &: \forall_x(A \to^{\mathsf{U}} \exists_x^{\mathsf{L}} A),\\
(\exists^{\mathsf{R}})^+ &: \forall_x^{\mathsf{U}}(A \to \exists_x^{\mathsf{R}} A),\\
(\exists^{\mathsf{U}})^+ &: \forall_x^{\mathsf{U}}(A \to^{\mathsf{U}} \exists_x^{\mathsf{U}} A),
\end{aligned}$$

where $\exists_x A$ abbreviates $\mathrm{Ex}(\rho,\{\, x^\rho \mid A\,\})$ (and similarly for the other ones), and the elimination axioms are (with $x \notin \mathrm{FV}(C)$)

$$\begin{aligned}
\exists^- &: \quad \exists_x A \to \forall_x(A \to C) \to C,\\
(\exists^{\mathsf{L}})^- &: \exists_x^{\mathsf{L}} A \to \forall_x(A \to^{\mathsf{U}} C) \to C,\\
(\exists^{\mathsf{R}})^- &: \exists_x^{\mathsf{R}} A \to \forall_x^{\mathsf{U}}(A \to C) \to C,\\
(\exists^{\mathsf{U}})^- &: \exists_x^{\mathsf{U}} A \to \forall_x^{\mathsf{U}}(A \to^{\mathsf{U}} C) \to C.
\end{aligned}$$

Conversion:

$$\exists^-\vec{p}\vec{q}\big(\exists^+\vec{p}\vec{r}^{\,\rho}N^{A(\vec{p},r)}\big)M^{\forall_n(A(\vec{p})\to Q(\vec{p},\vec{q}))} \mapsto MrN.$$

Conjunction can be treated similarly.

Falsity. $\bot := \mu_X(F \to X)$. This example is somewhat extreme, since the list $\vec{K}$ in the general form $\mu_{\vec{X}}\vec{K}$ is almost empty here: it only consists of an efq-clause. The only introduction axiom is

$$\bot^+ \colon F \to \bot$$

and the elimination axiom

$$\bot^- \colon \bot \to C.$$

Conversion (assuming that there are no free variables in C): Let $N_0 \colon F \to C$.

$$(\bot^-)^{\bot\to C}\big((\bot^+)^{F\to\bot}M^F\big) \mapsto N_0 M.$$

Equality. Let α be a type variable, x, y object variables of type α, and X a predicate variable of arity (α, α). We define *Leibniz equality* by

$$\mathrm{Eq}(\alpha) := \mu_X\big(\forall^{\mathsf{U}}_{x,y}(F \to X(x,y)), \forall^{\mathsf{U}}_x X(x,x)\big).$$

The introduction axioms are

$$\mathrm{Eq}^+_0 \colon \forall^{\mathsf{U}}_{n,m}(F \to \mathrm{Eq}(n,m)), \quad \mathrm{Eq}^+_1 \colon \forall^{\mathsf{U}}_n \mathrm{Eq}(n,n)$$

where $\mathrm{Eq}(n,m)$ abbreviates $\mathrm{Eq}(\rho)(n^\rho, m^\rho)$, and the elimination axiom is

$$\mathrm{Eq}^- \colon \forall^{\mathsf{U}}_{n,m}\big(\mathrm{Eq}(n,m) \to \forall^{\mathsf{U}}_n Q(n,n) \to Q(n,m)\big).$$

One easily proves symmetry, transitivity and also *compatibility* of Eq:

LEMMA (CompatEq). $\forall^{\mathsf{U}}_{n_1,n_2}\big(\mathrm{Eq}(n_1,n_2) \to Q(n_1) \to Q(n_2)\big)$.

PROOF. Use Eq^-; the details are left as an exercise. □

The even numbers. The introduction axioms are

$$\begin{aligned}
&\mathrm{Even}^+_0 \colon \forall^{\mathsf{U}}_n(F \to \mathrm{Even}(n)),\\
&\mathrm{Even}^+_1 \colon \mathrm{Even}(0),\\
&\mathrm{Even}^+_2 \colon \forall^{\mathsf{U}}_n(\mathrm{Even}(n) \to \mathrm{Even}(\mathrm{S}(\mathrm{S}n)))
\end{aligned}$$

and the (strengthened) elimination axiom is Even^-:

$$\forall^{\mathsf{U}}_m\big(\mathrm{Even}(m) \to P(0) \to \forall^{\mathsf{U}}_n(\mathrm{Even}(n) \to P(n) \to P(\mathrm{S}(\mathrm{S}n))) \to P(m)\big).$$

The accessible part of an ordering. Let $\prec$ be a binary relation. Assume that we have decidable sets M of its minimal elements and I of its interior elements. Then the *accessible part* of $\prec$ is inductively defined as follows. The introduction axioms are

$$\begin{aligned}
&\mathrm{Acc}^+_0 \colon \forall_x(M(x) \to^{\mathsf{U}} \mathrm{Acc}(x)),\\
&\mathrm{Acc}^+_1 \colon \forall_x(I(x) \to^{\mathsf{U}} \forall_{y\prec x}\mathrm{Acc}(y) \to \mathrm{Acc}(x)),
\end{aligned}$$

and the (strengthened) elimination axiom is as expected.

The transitive closure of a relation $\prec$. The introduction axioms are

$$\forall_{x,y}(x \prec y \to^{\mathsf{U}} \mathrm{TrCl}(x,y)),$$
$$\forall_x \forall^{\mathsf{U}}_{y,z}(x \prec y \to^{\mathsf{U}} \mathrm{TrCl}(y,z) \to \mathrm{TrCl}(x,z))$$

and the (strengthened) elimination axiom is

$$\begin{aligned}\forall^{\mathsf{U}}_{x_1,y_1}\big(\mathrm{TrCl}(x_1,y_1) \to \forall_{x,y}(x \prec y \to^{\mathsf{U}} P(x,y)) \to \\ \forall_x \forall^{\mathsf{U}}_{y,z}(x \prec y \to^{\mathsf{U}} \mathrm{TrCl}(y,z) \to P(y,z) \to P(x,z)) \to \\ P(x_1,y_1)\big).\end{aligned}$$

Pointwise equality. For a type ρ let $\mathrm{fvt}(\rho)$ be the set of its *final value types*, consisting of base types. We define $\mathrm{fvt}(\rho)$ by induction on ρ: $\mathrm{fvt}(\rho \to \sigma) := \mathrm{fvt}(\sigma)$, and for a base type $\mu = (\mu_{\vec{\alpha}}(\kappa_0,\dots,\kappa_{k-1}))_j$, $\mathrm{fvt}(\mu)$ consists of all the (base) types $\mu_{\vec{\alpha}}(\kappa_0,\dots,\kappa_{k-1})$, plus the final value types of all parameter types of μ.

For every type ρ we inductively define *pointwise equality* $=_\rho$. The introduction axioms are, for every base type μ without parameter types,

$$\forall^{\mathsf{U}}_{x_1,x_2}(F \to x_1 =_\mu x_2).$$

For every constructor C_i of a base type μ_j we have an introduction axiom

$$\forall^{\mathsf{U}}_{\vec{y},\vec{z}}\big(\vec{y}^P =_{\vec{\rho}} \vec{z}^P \to \big(\forall_{\vec{x}_\nu}(y^R_{m+\nu}\vec{x}_\nu =_{\mu_{j_\nu}} z^R_{m+\nu}\vec{x}_\nu)\big)_{\nu<n} \to \mathrm{C}_i\vec{y} =_{\mu_j} \mathrm{C}_i\vec{z}\big).$$

For every arrow type $\rho \to \sigma$ we have the introduction axiom

$$\forall^{\mathsf{U}}_{x_1,x_2}\big(\forall_y(x_1 y =_\sigma x_2 y) \to x_1 =_{\rho\to\sigma} x_2\big).$$

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

$$\forall^{\mathsf{U}}_{n_1,n_2}(F \to n_1 =_{\mathbf{N}} n_2),$$
$$0 =_{\mathbf{N}} 0,$$
$$\forall^{\mathsf{U}}_{n_1,n_2}(n_1 =_{\mathbf{N}} n_2 \to \mathrm{S}n_1 =_{\mathbf{N}} \mathrm{S}n_2),$$

and the elimination axiom is

$$\begin{aligned}\forall^{\mathsf{U}}_{m_1,m_2}\big(m_1 =_{\mathbf{N}} m_2 \to P(0,0) \to \\ \forall^{\mathsf{U}}_{n_1,n_2}\big(n_1 =_{\mathbf{N}} n_2 \to P(n_1,n_2) \to P(\mathrm{S}n_1,\mathrm{S}n_2)\big) \to \\ P(m_1,m_2)\big).\end{aligned}$$

An example with a non-finitary base type is $=_{\mathbf{T}}$ with $\mathbf{T} := \mathcal{T}_1$ (cf. 1.1.1):

$$\forall^{\mathsf{U}}_{x_1,x_2}(F \to x_1 =_{\mathbf{T}} x_2),$$
$$0 =_{\mathbf{T}} 0,$$
$$\forall^{\mathsf{U}}_{f_1,f_2}(\forall_n(f_1 n =_{\mathbf{T}} f_2 n) \to \mathrm{Sup}f_1 =_{\mathbf{T}} \mathrm{Sup}f_2),$$

and the elimination axiom is

$$=^{-}_{\mathbf{T}}\colon \forall^{\mathsf{U}}_{x_1,x_2}\big(x_1 =_{\mathbf{T}} x_2 \to P(0,0) \to \forall^{\mathsf{U}}_{f_1,f_2}\big(\forall_n(f_1 n =_{\mathbf{T}} f_2 n) \to \forall_n P(f_1 n, f_2 n) \to P(\mathrm{Sup} f_1, \mathrm{Sup} f_2)\big) \to P(x_1,x_2)\big).$$

One can prove *reflexivity* of $=_\rho$, using meta-induction on ρ, and induction on the types in $\mathrm{fvt}(\rho)$.

Lemma (ReflPtEq). $\forall_n(n =_\rho n)$.

A consequence is that Leibniz equality implies pointwise equality:

Lemma (EqToPtEq). $\forall_{n_1,n_2}\big(\mathrm{Eq}(n_1,n_2) \to n_1 =_\rho n_2\big)$.

Proof. Use CompatEq and ReflPtEq. □

2.1.3. Further axioms and their consequences. We express *extensionality* of our intended model by stipulating that pointwise equality implies Leibniz equality:

Axiom (PtEqToEq). $\forall_{n_1,n_2}\big(n_1 =_\rho n_2 \to \mathrm{Eq}(n_1,n_2)\big)$.

Notice that this implies the following proposition, which is sometimes called extensionality as well:

Lemma (CompatPtEqFct). $\forall_f \forall^{\mathsf{U}}_{n_1,n_2}(n_1 =_\rho n_2 \to f n_1 =_\sigma f n_2)$.

Proof. We obtain $\mathrm{Eq}(n_1,n_2)$ by PtEqToEq. By ReflPtEq we have $f n_1 =_\sigma f n_1$, hence $f n_1 =_\sigma f n_2$ by CompatEq. □

A consequence of the extensionality axioms is that compatibility holds for pointwise equality as well:

Lemma (CompatPtEq). $\forall^{\mathsf{U}}_{n_1,n_2}\big(n_1 =_\rho n_2 \to P(n_1) \to P(n_2)\big)$.

Proof. Use PtEqToEq and CompatEq. □

We write E-ID$^\omega$ when the extensionality axioms PtEqToEq are present. In E-ID$^\omega$ we can prove properties of the constructors of our free algebras: that they are *injective*, and have *disjoint ranges*. For finitary algebras this can be seen easily, using boolean-valued equality. However, for non-finitary algebras we need extensionality. Since extensionality implies that pointwise and Leibniz equality are equivalent, it suffices to consider pointwise equality. Rather than dealing with the general case, we confine ourselves with the algebra $\mathbf{T}$.

The proof uses some recursive functions: $\mathrm{TreeSup}\colon \mathbf{T} \to \mathbf{B}$ defined by

$$\mathrm{TreeSup}(0) := \mathsf{ff}, \quad \mathrm{TreeSup}(\mathrm{Sup} f) := \mathsf{tt}$$

and a predecessor function TreePred: $\mathbf{T} \to \mathbf{N} \to \mathbf{B}$ defined by

$$\mathrm{TreePred}(0, n) := 0, \quad \mathrm{TreePred}(\mathrm{Sup} f, n) := fn.$$

LEMMA. *$0 =_{\mathbf{T}} \mathrm{Sup} f \to F$, and $\forall^{\mathsf{U}}_{f_1, f_2}(\mathrm{Sup} f_1 =_{\mathbf{T}} \mathrm{Sup} f_2 \to f_1 =_{\mathbf{N}\to\mathbf{T}} f_2)$.*

PROOF. The proof uses the various compatibilities, and the fact that conversions give rise to Leibniz equalities. □

We now list some further axioms, which will be mentioned when we use them. All of them involve (inductively defined) existentially quantified formulas, wich come in four versions, with $\exists, \exists^{\mathsf{R}}, \exists^{\mathsf{L}}, \exists^{\mathsf{U}}$. Let $\breve{\exists}$ denote any of these. When $\breve{\exists}$ appears more than once in an axiom below, it is understood that it denotes the same quantifier each time.

The *axiom of choice* (AC) is the scheme

AXIOM (AC). $\forall_{x^\rho} \breve{\exists}_{y^\sigma} A(x, y) \to \breve{\exists}_{f^{\rho\to\sigma}} \forall_{x^\rho} A(x, f(x))$.

The *independence* axioms express the intended meaning of uniformities. The *independence of premise* axiom (IP) is

AXIOM (IP). $(A \to^{\mathsf{U}} \breve{\exists}_x B) \to \breve{\exists}_x (A \to^{\mathsf{U}} B)$ *($x \notin \mathrm{FV}(A)$).*

Similarly we have an *independence of quantifier* axiom (IQ) axiom

AXIOM (IQ). $\forall^{\mathsf{U}}_x \breve{\exists}_y A \to \breve{\exists}_y \forall^{\mathsf{U}}_x A$ *($x \notin \mathrm{FV}(A)$).*

2.2. Computational Content

Along the inductive definition of formulas, predicates and constructor formulas (or clauses) in 2.1.1, we define simultaneously

- the *type* $\tau(A)$ of a formula A;
- when a formula is *computationally relevant*;
- the formula z *realizes* A, written z **r** A, for a variable z of type $\tau(A)$;
- when a formula is *negative*;
- when an inductively defined predicate requires *witnesses*;
- for an inductively defined I requiring witnesses, its base type μ_I, and – if I has an efq-clause – when an object of type μ_I is *efq-free*;
- for an inductively defined predicate I of arity $\vec{\rho}$ requiring witnesses, a *witnessing* predicate $I^{\mathbf{r}}$ of arity $(\mu_I, \vec{\rho})$, and a predicate $I^{\mathbf{ef}}$ of arity (μ_I) expressing *efq-free*ness.

2.2.1. The type of a formula. Every formula A possibly containing inductively defined predicates can be seen as a "computational problem". We define $\tau(A)$ as the type of a potential realizer of A, i.e., the type of the term (or "program") to be extracted from a proof of A.

More precisely, we assign to every formula A an object $\tau(A)$ (a type or the "nulltype" symbol ε). In case $\tau(A) = \varepsilon$ proofs of A have no computational content; such formulas A are called *Harrop formulas*, or computationally *irrelevant* (c.i.). Non-Harrop formulas are also called computationally *relevant* (c.r.).

The definition can be conveniently written if we extend the use of $\rho \to \sigma$ to the nulltype symbol ε:

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

With this understanding of $\rho \to \sigma$ we can simply write

$$\tau(\mathrm{atom}(r)) := \varepsilon, \quad \tau(I(\vec{r})) := \begin{cases} \varepsilon & \text{if } I \text{ does not require witnesses} \\ \mu_I & \text{otherwise,} \end{cases}$$
$$\tau(A \to B) := (\tau(A) \to \tau(B)), \quad \tau(\forall_{x^\rho} A) := (\rho \to \tau(A)),$$
$$\tau(A \to^{\mathsf{U}} B) := \tau(B), \quad \tau(\forall^{\mathsf{U}}_{x^\rho} A) := \tau(A).$$

2.2.2. Realizability. Let A be a formula and z either a variable of type $\tau(A)$ if the latter is a type, or the nullterm symbol ε if $\tau(A) = \varepsilon$. For a convenient definition we extend the use of term application to the nullterm symbol ε:

$$\varepsilon x := \varepsilon, \quad z\varepsilon := z, \quad \varepsilon\varepsilon := \varepsilon.$$

We define the formula $z\ \mathbf{r}\ A$, to be read z *realizes* A. The definition uses the predicates $I^{\mathbf{r}}$ and $I^{\mathbf{ef}}$ introduced below.

$$z\ \mathbf{r}\ \mathrm{atom}(s) := \mathrm{atom}(s),$$
$$z\ \mathbf{r}\ I(\vec{s}) := \begin{cases} I(\vec{s}) & \text{if } I \text{ does not require witnesses} \\ I^{\mathbf{r}}(z, \vec{s}) & \text{if not, and } I \text{ has no efq-clause} \\ I^{\mathbf{ef}}(z) \wedge I^{\mathbf{r}}(z, \vec{s}) & \text{otherwise,} \end{cases}$$
$$z\ \mathbf{r}\ (A \to B) := \forall_x (x\ \mathbf{r}\ A\ \to\ zx\ \mathbf{r}\ B),$$
$$z\ \mathbf{r}\ (\forall_x A) := \forall_x\, zx\ \mathbf{r}\ A,$$
$$z\ \mathbf{r}\ (A \to^{\mathsf{U}} B) := (A \to z\ \mathbf{r}\ B),$$
$$z\ \mathbf{r}\ (\forall^{\mathsf{U}}_x A) := \forall_x\, z\ \mathbf{r}\ A.$$

Formulas which do not contain inductively defined predicates requiring witnesses play a special role in this context; we call them *negative*. Their crucial property is $(\varepsilon\ \mathbf{r}\ A) = A$. Notice also that every formula $z\ \mathbf{r}\ A$ is negative.

2.2.3. Witnesses. Consider a particularly simple inductively defined predicate, where

- there is at most one clause apart from an efq-clause, and
- this clause is uniform, i.e., contains no $\forall$ but $\forall^{\mathsf{U}}$ only, and its premises are either negative or followed by $\to^{\mathsf{U}}$.

Examples are $\exists^{\mathsf{U}}$, $\wedge^{\mathsf{U}}$, $\bot$, Eq. We call those predicates "uniform one-clause" defined. An inductively defined predicate *requires witnesses* if it is not one of those, and not one of the predicates $I^{\mathbf{r}}$ and $I^{\mathbf{ef}}$ introduced below.

For an inductively defined predicate I requiring witnesses, we define μ_I to be the corresponding component of the types $\vec{\mu} = \mu_{\vec{\alpha}}\vec{\kappa}$ generated from constructor types $\kappa_i := \tau(K_i)$ for all constructor formulas $K_0, \dots K_{k-1}$ from $\vec{I} = \mu_{\vec{X}}(K_0, \dots K_{k-1})$. An object of type μ_I is called *efq-free* it it does not contain a constructor of μ_I corresponding to an efq-clause.

The witnessing predicate $I^{\mathbf{r}}$ of arity $(\mu_I, \vec{\rho})$ can now be defined as follows. For every constructor formula

$$K = \breve{\forall}_{\vec{x}}\big(\vec{A} \mathrel{\breve{\to}} \big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} X_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \to X_j(\vec{t}\,)\big)$$

of the original inductive definition of $\vec{I}$ we build the new constructor formula

$$K^{\mathbf{r}} := \breve{\forall}_{\vec{x}}\forall_{\vec{u},\vec{f}}\big(\vec{u}\ \mathbf{r}\ \vec{A} \to \big(\breve{\forall}_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to Y_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu, \vec{s}_\nu))\big)_{\nu<n} \to Y_j(\mathrm{C}\vec{x}\vec{u}\vec{f}, \vec{t}\,)\big),$$

with the understanding that

- only those x_i with a non-uniform $\forall_{x_i}$ occur as arguments in $\mathrm{C}\vec{x}\vec{u}\vec{f}$,
- only those u_i with A_i a non-uniform premise and $\tau(A_i) \neq \varepsilon$ actually appear (for the other A_i we take either A_i or $\varepsilon\ \mathbf{r}\ A_i$),

and similarly for $y_{\nu,i}$, $v_{\nu,i}$ and $f_\nu\vec{y}_\nu\vec{v}_\nu$. Here C is the constructor of the algebra $\vec{\mu} = \mu_{\vec{\alpha}}\vec{\kappa}$ generated from our constructor types $\kappa_i := \tau(K_i)$ (i.e., for K_i we have $\mathrm{C} := \mathrm{C}_i$). Then $\vec{I}^{\mathbf{r}} := \mu_{\vec{Y}}(\vec{K}^{\mathbf{r}})$. The corresponding introduction axiom then is $K^{\mathbf{r}}(\vec{I}^{\mathbf{r}})$, that is

$$(I_j^{\mathbf{r}})_i^+ \colon \forall_{\vec{x},\vec{u},\vec{f}}\big(\vec{u}\ \mathbf{r}\ \vec{A} \to \big(\forall_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to I^{\mathbf{r}}_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu, \vec{s}_\nu))\big)_{\nu<n} \to I^{\mathbf{r}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f}, \vec{t}\,)\big) \tag{2.3}$$

and the (strengthened) elimination axiom is

$$(I_j^{\mathbf{r}})^- \colon \forall_w\forall^{\mathsf{U}}_{\vec{x}}\big(I^{\mathbf{r}}_j(w, \vec{x}\,) \to \big(K^{\mathbf{r}}_i(\vec{I}^{\mathbf{r}}, \vec{P})\big)_{i<k} \to P_j(w, \vec{x}\,)\big) \tag{2.4}$$

with

$$K^{\mathbf{r}}(\vec{I}^{\mathbf{r}}, \vec{P}) := \breve{\forall}_{\vec{x}}\forall_{\vec{u},\vec{f}}\big(\vec{u}\ \mathbf{r}\ \vec{A} \to \big(\forall_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to I^{\mathbf{r}}_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu, \vec{s}_\nu))\big)_{\nu<n} \to \big(\breve{\forall}_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to P_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu, \vec{s}_\nu))\big)_{\nu<n} \to$$

$$P_j(\mathrm{C}\vec{x}\vec{u}\vec{f},\vec{t}\,)\big).$$

Notice that each of the clauses $(I^{\mathbf{r}}_j)^+_i$ has a conclusion $I^{\mathbf{r}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f},\vec{t}\,)$ with its own constructor C. Therefore it is to be expected that the following *inversion* properties for $I^{\mathbf{r}}$ hold:

Lemma (Inversion).

$$(I^{\mathbf{r}}_j)^{\mathrm{invEq}}_i\colon I^{\mathbf{r}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f},\vec{z}) \to \exists_{\vec{y}}\mathrm{Eq}(\vec{z},\vec{t}\,) \quad (\vec{y} \text{ the uniform variables}),$$
$$(I^{\mathbf{r}}_j)^{\mathrm{invP}}_i\colon I^{\mathbf{r}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f},\vec{t}\,) \to \vec{u}\ \mathbf{r}\ \vec{A},$$
$$(I^{\mathbf{r}}_j)^{\mathrm{invR},\nu}_i\colon I^{\mathbf{r}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f},\vec{t}\,) \to \forall_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to I^{\mathbf{r}}_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu,\vec{s}_\nu)).$$

For an inductively defined predicate I requiring witnesses and with an efq-clause we define the predicate $I^{\mathbf{ef}}$ of arity (μ_I) expressing efq-freeness as follows. For every constructor formula

$$K = \breve{\forall}_{\vec{x}}\big(\vec{A} \mathrel{\breve{\to}} \big(\breve{\forall}_{\vec{y}_\nu}(\vec{B}_\nu \mathrel{\breve{\to}} X_{j_\nu}(\vec{s}_\nu))\big)_{\nu<n} \mathrel{\breve{\to}} X_j(\vec{t}\,)\big)$$

of the original inductive definition of $\vec{I}$ *except* the efq-clause the corresponding introduction axiom is $(I^{\mathbf{ef}}_j)^+_i$:

$$\breve{\forall}_{\vec{x},\vec{u},\vec{f}}\big(\vec{u}\ \mathbf{r}\ \vec{A} \to \big(\breve{\forall}_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to I^{\mathbf{ef}}_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu))\big)_{\nu<n} \to I^{\mathbf{ef}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f}\,)\big) \tag{2.5}$$

and the elimination axiom is

$$(I^{\mathbf{ef}}_j)^-\colon \forall_w\big(I^{\mathbf{ef}}_j(w) \to \big(K^{\mathbf{ef}}_i(\vec{I}^{\mathbf{ef}},\vec{P})\big)_{i<k} \to P_j(w)\big). \tag{2.6}$$

As before we can prove

Lemma (Inversion).

$$(I^{\mathbf{ef}}_j)^{\mathrm{invP}}_i\colon I^{\mathbf{ef}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f}\,) \to \vec{u}\ \mathbf{r}\ \vec{A},$$
$$(I^{\mathbf{ef}}_j)^{\mathrm{invR},\nu}_i\colon I^{\mathbf{ef}}_j(\mathrm{C}\vec{x}\vec{u}\vec{f}\,) \to \forall_{\vec{y}_\nu,\vec{v}_\nu}(\vec{v}_\nu\ \mathbf{r}\ \vec{B}_\nu \to I^{\mathbf{ef}}_{j_\nu}(f_\nu\vec{y}_\nu\vec{v}_\nu)).$$

2.3. Extracted Terms and Uniform Proofs

We define the extracted term of a proof, and (using this concept) the notion of a uniform proof, which gives a special treatment to uniform implication $\to^{\mathsf{U}}$ and the uniform universal quantifier $\forall^{\mathsf{U}}$.

2.3.1. Extracted terms. For a derivation M in $\mathrm{ID}^\omega + \mathrm{AC} + \mathrm{IP}_\varepsilon + \mathrm{Ax}_\varepsilon$, we simultaneously define

- its *extracted term* $[\![M]\!]$, of type $\tau(A)$, and
- when M is *uniform*.

For derivations M^A where $\tau(A) = \varepsilon$ (i.e., A is a Harrop formula) let $[\![M]\!] := \varepsilon$ (the *nullterm* symbol); every such derivation is uniform. Now assume that M derives a formula A with $\tau(A) \neq \varepsilon$. Recall our extended use of term application to the nullterm symbol ε: $\varepsilon x := \varepsilon$, $z\varepsilon := z$, $\varepsilon\varepsilon := \varepsilon$. We also understand that in case $\tau(A) = \varepsilon$, $\lambda_{x_u^{\tau(A)}}[\![M]\!]$ means just $[\![M]\!]$. Then

$$\begin{aligned}
[\![u^A]\!] &:= x_u^{\tau(A)} \quad (x_u^{\tau(A)} \text{ uniquely associated with } u^A),\\
[\![\lambda_{u^A} M]\!] &:= \lambda_{x_u^{\tau(A)}}[\![M]\!],\\
[\![M^{A\to B}N]\!] &:= [\![M]\!][\![N]\!],\\
[\![(\lambda_{x^\rho} M)^{\forall_x A}]\!] &:= \lambda_{x^\rho}[\![M]\!],\\
[\![M^{\forall_x A} r]\!] &:= [\![M]\!] r.\\
[\![\lambda^{\mathsf{U}}_{u^A} M]\!] &:= [\![M^{A\to^{\mathsf{U}} B}N]\!] := [\![(\lambda^{\mathsf{U}}_{x^\rho} M)^{\forall^{\mathsf{U}}_x A}]\!] := [\![M^{\forall^{\mathsf{U}}_x A} r]\!] := [\![M]\!].
\end{aligned}$$

In all these cases uniformity is preserved, except possibly in those involving λ^{U}: $\lambda^{\mathsf{U}}_{u^A} M$ is uniform if M is and $x_u \notin \mathrm{FV}([\![M]\!])$, and $\lambda^{\mathsf{U}}_{x^\rho} M$ is uniform if M is and – in addition to the usual variable condition – $x \notin \mathrm{FV}([\![M]\!])$.

It remains to define extracted terms for the axioms: structural induction, introduction and elimation axioms for inductively defined predicates, (AC) and (IP_ε).

The extracted term $[\![\mathrm{Ind}_j]\!]$ of an induction axiom is defined to be the recursion operator $\mathcal{R}^{\vec{\mu},\vec{\tau}}_{\mu_j}$. For example, in case of an induction scheme

$$\mathrm{Ind}_{n,A}\colon \forall_m\big(A(0) \to \forall_n(A(n) \to A(\mathrm{S}n)) \to A(m^{\mathbf{N}})\big)$$

we have

$$[\![\mathrm{Ind}_{n,A}]\!] := \mathcal{R}^{\tau}_{\mathbf{N}}\colon \mathbf{N} \to \tau \to (\mathbf{N} \to \tau \to \tau) \to \tau \quad (\tau := \tau(A) \neq \varepsilon).$$

Generally, the $\vec{\mu}, \vec{\tau}$ in $\mathcal{R}^{\vec{\mu},\vec{\tau}}_{\mu_j}$ list only the types μ_j, τ_j with $\tau_j := \tau(A_j) \neq \varepsilon$, i.e., the recursion operator is simplified accordingly.

For the introduction axiom (2.1) and the (strengthened) elimination axiom (2.2) of an inductively defined predicate I we define

$$[\![(I_j)^+_i]\!] := \mathrm{C}, \quad [\![I^-_j]\!] := \mathcal{R}_j,$$

and similary for the introduction and elimination axioms for $I^{\mathbf{r}}$ and $I^{\mathbf{ef}}$: (2.3), (2.4) and (2.5), (2.6), respectively.

As extracted terms of (AC), (IP) and (IQ) we take identities of the appropriate types.

2.3.2. Uniform derivations. Here we collect some general remarks on uniform derivations.

Lemma. *There are purely logical uniform derivations of*

(a) $A \to B$ *from* $\mathrm{A} \to^{\mathsf{U}} B$;
(b) $A \to^{\mathsf{U}} B$ *from* $\mathrm{A} \to B$, *provided* $\tau(A) = \varepsilon$ *or* $\tau(B) = \varepsilon$;
(c) $\forall_x A$ *from* $\forall_x^{\mathsf{U}} A$;
(d) $\forall_x^{\mathsf{U}} A$ *from* $\forall_x A$, *provided* $\tau(A) = \varepsilon$.

Proof. (a). $\lambda_v(u^{A \to^{\mathsf{U}} B} v^A)$ is uniform (there are no conditions on λ_v). (b). If $\tau(B) = \varepsilon$, then $\lambda_v^{\mathsf{U}}(u^{A \to B} v^A)$ is unform because its conclusion is a Harrop formula. Now assume $\tau(A) = \varepsilon$. Then for $\lambda_v^{\mathsf{U}}(u^{A \to B} v^A)$ to be uniform we need to know that $x_v \notin \mathrm{FV}([\![uv]\!])$. But $[\![uv]\!] = [\![u]\!]$ because of $\tau(A) = \varepsilon$. (c). Exercise. □

We certainly want to know that in formulas involving $\to^{\mathsf{U}}$ and $\forall^{\mathsf{U}}$ we can replace a subformula by an equivalent one.

Lemma. *There are purely logical uniform derivations of*
(a) $(A \to^{\mathsf{U}} B) \to (B \to B') \to A \to^{\mathsf{U}} B'$;
(b) $(A' \to A) \to^{\mathsf{U}} (A \to^{\mathsf{U}} B) \to A' \to^{\mathsf{U}} B$;
(c) $\forall_x^{\mathsf{U}} A \to (A \to A') \to \forall_x^{\mathsf{U}} A'$.

Proof. (a). Exercise. (b). Exercise. (c). $\lambda_{u,v}\lambda_x^{\mathsf{U}}(v^{A \to A'}(u^{\forall_x^{\mathsf{U}} A} x))$ is uniform because $[\![v(ux)]\!] = x_v x_u$ does not contain x free. □

For the (inductively defined) existential quantifiers $\exists, \exists^{\mathsf{R}}, \exists^{\mathsf{L}}, \exists^{\mathsf{U}}$ we observe the following. Let $\breve{\exists}$ denote any of these.

Lemma. *There are uniform derivations using* $\exists$*-axioms only of*
(a) $\exists_x A \to \breve{\exists}_x A$;
(b) $\breve{\exists}_x A \to \exists_x^{\mathsf{U}} A$;
(c) $\exists_x^{\mathsf{L}} A \to \exists_x A$, *provided* $\tau(A) = \varepsilon$.

Proof. (a) Use $\exists^-\colon \exists_x A \to \forall_x(A \to \breve{\exists}_x A) \to \breve{\exists}_x A$. We derive the second premise using an introduction axiom. An example is

$$\dfrac{\dfrac{(\exists^{\mathsf{L}})^+\colon \forall_x(A \to^{\mathsf{U}} \exists_x^{\mathsf{L}} A) \qquad x \qquad u\colon A}{\exists_x^{\mathsf{L}} A}}{\dfrac{A \to \exists_x^{\mathsf{L}} A}{\forall_x(A \to \exists_x^{\mathsf{L}} A)}\,\forall^+ x}\,{\to^+} u$$

(b). Exercise. (c). Exercise. □

2.3.3. Characterization. We consider the question when a formula A and its modified realizability interpretation $\exists_x\, x\ \mathbf{r}\ A$ are equivalent.

Theorem (Characterization; cf. Troelstra (1973, 3.4.8)).

$$\mathrm{ID}^\omega + \mathrm{AC} + \mathrm{IP} + \mathrm{IQ} \vdash A \leftrightarrow \exists_x\, x\ \mathbf{r}\ A.$$

PROOF. Induction on A, along the inductive definition of formulas, predicates and constructor formulas (or clauses) in 2.1.1. The case of an inductively defined predicate is similar to the examples above. *Case* $A \to B$.

$$\begin{aligned}(A \to B) &\leftrightarrow (\exists_x\, x\ \mathbf{r}\ A \to \exists_z\, z\ \mathbf{r}\ B) && \text{by IH}\\ &\leftrightarrow \forall_x(x\ \mathbf{r}\ A \to \exists_z\, z\ \mathbf{r}\ B)\\ &\leftrightarrow \forall_x\exists_z(x\ \mathbf{r}\ A \to z\ \mathbf{r}\ B) && \text{by (IP)}\\ &\leftrightarrow \exists_f\forall_x(x\ \mathbf{r}\ A \to f(x)\ \mathbf{r}\ B) && \text{by (AC)}\\ &\leftrightarrow \exists_f f\ \mathbf{r}\ (A \to B).\end{aligned}$$

Case $\forall_x A$.

$$\begin{aligned}\forall_x A &\leftrightarrow \forall_x\exists_z\, z\ \mathbf{r}\ A && \text{by IH}\\ &\leftrightarrow \exists_f\forall_x\, fx\ \mathbf{r}\ A && \text{by (AC)}\\ &\leftrightarrow \exists_f f\ \mathbf{r}\ \forall_x A.\end{aligned}$$

Case $A \to^{\mathsf{U}} B$.

$$\begin{aligned}(A \to^{\mathsf{U}} B) &\leftrightarrow (A \to^{\mathsf{U}} \exists_z\, z\ \mathbf{r}\ B) && \text{by IH}\\ &\leftrightarrow \exists_z(A \to^{\mathsf{U}} z\ \mathbf{r}\ B) && \text{by (IP)}\\ &\leftrightarrow \exists_z\, z\ \mathbf{r}\ (A \to^{\mathsf{U}} B).\end{aligned}$$

Case $\forall_x^{\mathsf{U}} A$.

$$\begin{aligned}\forall_x^{\mathsf{U}} A &\leftrightarrow \forall_x^{\mathsf{U}}\exists_z\, z\ \mathbf{r}\ A && \text{by IH}\\ &\leftrightarrow \exists_z\forall_x^{\mathsf{U}}\, z\ \mathbf{r}\ A && \text{by (IQ)}\\ &\leftrightarrow \exists_z\, z\ \mathbf{r}\ \forall_x^{\mathsf{U}} A.\end{aligned}$$

This concludes the proof. □

2.3.4. Soundness. We address the question how we know that the term extracted from a proof of A indeed "realizes" (Kolmogorov (1925): solves) the formula (Kolmogorov (1925): problem) A. Making use of the fact that what we extract is a term in our arithmetical language T, we can indeed prove such a "soundness" theorem. It implies that every theorem in E-ID$^\omega$ + AC + IP + IQ + Ax$_\varepsilon$ has a realizer. Here (Ax$_\varepsilon$) is an arbitrary set of Harrop formulas viewed as axioms.

THEOREM (Soundness). *We work in* ID$^\omega$ + AC + IP + IQ. *Let* M *be a derivation of* A *from assumptions* $u_i\colon C_i$ $(i < n)$. *Then we can find a derivation* $\sigma(M)$ *of* $[\![M]\!]\ \mathbf{r}\ A$ *from assumptions* $\bar{u}_i\colon x_{u_i}\ \mathbf{r}\ C_i$ *for a non-uniform* u_i *(i.e.,* $x_{u_i} \in \mathrm{FV}([\![M]\!])$*), and* $\bar{u}_i\colon C_i$ *for the other ones.*

PROOF. Induction on M. □

2.4. Examples: List Reversal Again

We again consider list reversal, but now extract terms from proofs with computational content. In fact, we will exemplify four different methods to do this, with different results.

2.4.1. A constructive proof of the existence of reverted lists. We first prove that every non-empty list can be written in the form $v :+: x:$. Using this, $\forall_v \exists_w \mathrm{Rev}(v, w)$ can be proved by induction on the length of v. In the step case, our list is non-empty, and hence can be written in the form $v :+: x:$. Since v has a smaller length, the IH yields its reversal w. Then we can take $x :: w$. Here is the term extracted (with the Minlog proof assistant (`www.minlog-system.de`)) from a formalization of this proof:

```
(Rec nat=>list nat=>list nat)([v2](Nil nat))
([n2,f3,v4]
  [if v4
    (Nil nat)
    ([n5,v6][let p7 (cListInitLastNat v6 n5)
              (right p7::f3 left p7)])])
```

It contains the term `cListInitLastNat` denoting the content of the auxiliary proposition, and in the step recursively calls itself via `f3`. The represented algorithm takes quadratic time.

2.4.2. Proving the correctness of the reversal function. Our arithmetical language based on Gödel's T allows to define functions recursively. In particular, we may directly define the list reversal function R by

$$R(\mathrm{nil}) = \mathrm{nil},$$
$$R(x :: v) = R(v) :+: x:.$$

Then we can prove $\forall_v \exists_w R(v, w)$, and extract a term from this proof. The result is

```
[Rev0]
 (Rec nat=>list nat=>list nat)([v3](Nil nat))
 ([n3,f4,v5]
   [if v5
     (Nil nat)
     ([n6,v7]right(cListInitLastNat v7 n6)::f4
             left(cListInitLastNat v7 n6))])
```

By a slight variation of the proof (insertion of an "identity" lemma $P \to P$) we can avoid the double calculation of `cListInitLastNat v7 n6` and introduce a `let` instead:

```
(Rec nat=>list nat=>list nat)([v2](Nil nat))
([n2,f3,v4]
  [if v4
    (Nil nat)
    ([n5,v6][let p7 (cListInitLastNat v6 n5)
              (right p7::f3 left p7)])])
```

Again the represented algorithm takes quadratic time.

2.4.3. Defining list reversal inductively, and using symmetry. We may avoid the existential quantifier altogether and consider the graph of the list reversal function as an inductively defined predicate. The clauses are

$$\begin{aligned}&\mathrm{Rev}_0^+ \colon \forall^{\mathsf{U}}_{v,w}(F \to \mathrm{Rev}(v,w)),\\ &\mathrm{Rev}_1^+ \colon \mathrm{Rev}(\mathrm{nil},\mathrm{nil}),\\ &\mathrm{Rev}_2^+ \colon \forall^{\mathsf{U}}_{v,w}\forall_x(\mathrm{Rev}(v,w) \to \mathrm{Rev}(v :+: x:, x :: w))\end{aligned}$$

and the (strengthened) elimination axiom is Rev^-:

$$\begin{aligned}&\forall^{\mathsf{U}}_{v,w}\big(\forall^{\mathsf{U}}_{v,w}(F \to P(v,w)) \to\\ &\quad P(\mathrm{nil},\mathrm{nil}) \to\\ &\quad \forall^{\mathsf{U}}_{v,w}\forall_x\big(\mathrm{Rev}(v,w) \to P(v,w) \to P(v :+: x:, x :: w)\big) \to\\ &\quad \mathrm{Rev}(v,w) \to P(v,w)\big).\end{aligned}$$

Then we can prove, using Rev^-

LEMMA (`RevCons`).

$$\forall^{\mathsf{U}}_{v,w}\big(\mathrm{Rev}(v,w) \to \forall_x\mathrm{Rev}(x :: v, v :+: x:)\big).$$

The extracted term is

```
(Rec algRev=>nat=>algRev)
([n2]cEfqRev)([n2]cGenRev n2 cInitRev)
([n2,algRev3,(nat=>algRev)_4,n5]
  cGenRev n2((nat=>algRev)_4 n5))
```

Using `RevCons` and the property $R(v :+: x:) = x :: R(v)$ of the reversal function we then can prove symmetry in the form $\mathrm{Rev}(v,w) \to \mathrm{Rev}(R(v),R(w))$. The term extracted from a formalization of this proof is

```
(Rec algRev=>algRev)cEfqRev cInitRev
([n1,algRev2,algRev3]cRevCons algRev3 n1)
```

This again is the usual quadratic algorithm.

2.4.4. The content of weak existence proofs. The proof given in 1.3 contains the propositional symbol $\bot$ for falsity; however, being a proof in minimal logic, it does not assume anything about $\bot$. Apart from the clauses for Rev (which do not involve $\bot$), it only made use of the "false" assumption

$$u\colon \forall_w(\mathrm{Rev}(v,w) \to \bot).$$

The end formula is $\bot$. Therefore we may replace $\bot$ throughout by an arbitrary formula, for instance $\exists_w \mathrm{Rev}(v,w)$. After this substitution the "false" assumption becomes trivially provable, and the end formula is $\bot$ becomes $\exists_w \mathrm{Rev}(v,w)$. So now we have converted the weak existence proof into a strong one. Here is its extracted term:

```
[Rev0,u1]
 (Rec list nat=>list nat=>list nat=>list nat)([u2,u3]u3)
 ([n2,u3,(list nat=>list nat=>list nat)_4,u5,u6]
   (list nat=>list nat=>list nat)_4(u5:+:n2:)(n2::u6))
 u1
 (Nil nat)
 (Nil nat)
```

Recall that in the proof we have made use of an auxiliary proposition

$$\forall_v\big(v :+: u = v_0 \to \forall_w \neg \mathrm{Rev}(v,w)\big).$$

It turns out that this proof does not use v computationally, that is, we can replace the leading universal quantifier $\forall_v$ by the uniform quantifier $\forall_v^{\mathsf{U}}$. The extracted term then is

```
[Rev0,u1]
 (Rec list nat=>list nat=>list nat)
 ([u2]u2)([n2,u3,f4,u5]f4(n2::u5))u1
 (Nil nat)
```

In fact, the underlying algorithm defines an auxiliary function h by

$$h(\mathrm{nil}, u) := u,$$
$$h(x :: v, u) := h(v, x :: u)$$

and gives the result by applying h to the original list and nil. So we have obtained (by automated extraction from a weak existence proof) the standard linear algorithm for list reversal, with its use of an accumulator.

The method described here is a simple special case of a certain refinement of the so-called "A-translation" of Dragalin (1979) and Friedman (1978); it is developed more generally in Berger et al. (2002); Berger (2005).

CHAPTER 3

Complexity

A natural question coming up when one extracts terms from proofs is what can be said about the complexity of the evaluation of these terms, when we view them as functional programs.

To demonstrate the usefulness and strength provided by finite types we show that all F_α ($\alpha < \varepsilon_0$) of the "fast growing" (or extended Grzegorczyk) hierarchy can be defined explicitely from higher type iteration operators alone. The F_α's are defined by recursion on α thus:

$$F_\alpha(n) = \begin{cases} n+1 & \text{if } \alpha = 0 \\ F_{\alpha-1}^{n+1}(n) & \text{if Succ}(\alpha) \\ F_{\alpha(n)}(n) & \text{if Lim}(\alpha) \end{cases}$$

where $F_{\alpha-1}^{n+1}(n)$ is the $n+1$-times iterate of $F_{\alpha-1}$ on n.

For instance, F_2 already has exponential growth, F_ω is essentially the Ackermann function (which grows faster than all primitive recursive functions), and every function definable in arithmetic is majorizable by one of the F_α's.

We define finite type extensions of the functions F_α, such that for $\alpha = 0$ we obtain iteration operators. Define the *pure types* ρ_n, by $\rho_0 := \mathbf{N}$ and $\rho_{n+1} := \rho_n \to \rho_n$. Let x_n be a variable of pure type ρ_n.

$$F_\alpha x_n \dots x_0 := \begin{cases} x_0 + 1 & \text{if } \alpha = 0 \text{ and } n = 0, \\ x_n^{x_0} x_{n-1} \dots x_0 & \text{if } \alpha = 0 \text{ and } n > 0, \\ F_{\alpha-1}^{x_0} x_n \dots x_0 & \text{if Succ}(\alpha), \\ F_{\alpha(x_0)} x_n \dots x_0 & \text{if Lim}(\alpha). \end{cases}$$

The lemma below shows that all F_α can be obtained by substitution alone from finite type iteration functionals F_0.

LEMMA. *$F_\alpha F_\beta = F_{\beta+\omega^\alpha}$, where it is assumed that α and β have Cantor Normal Forms which can simply be concatenated to form the normal form of $\alpha + \beta$.*

PROOF. By induction on α. If $\alpha = 0$,

$$F_0 F_\beta x_{n-1} \dots x_0 = F_\beta^{x_0} x_{n-1} \dots x_0 = F_{\beta+1} x_{n-1} \dots x_0.$$

If $\mathrm{Succ}(\alpha)$,

$$\begin{aligned} F_\alpha F_\beta x_{n-1} \dots x_0 &= F^{x_0}_{\alpha-1} F_\beta x_{n-1} \dots x_0 \\ &= F_{\beta+\omega^{\alpha-1}\cdot x_0} x_{n-1} \dots x_0 \quad \text{by IH} \\ &= F_{(\beta+\omega^\alpha)(x_0)} x_{n-1} \dots x_0 \\ &= F_{\beta+\omega^\alpha} x_{n-1} \dots x_0. \end{aligned}$$

If $\mathrm{Lim}(\alpha)$,

$$\begin{aligned} F_\alpha F_\beta x_{n-1} \dots x_0 &= F^{x_0}_{\alpha(x_0)} F_\beta x_{n-1} \dots x_0 \\ &= F_{\beta+\omega^{\alpha(x_0)}\cdot x_0} x_{n-1} \dots x_0 \quad \text{by IH} \\ &= F_{(\beta+\omega^\alpha)(x_0)} x_{n-1} \dots x_0 \\ &= F_{\beta+\omega^\alpha} x_{n-1} \dots x_0. \end{aligned}$$

This completes the proof. □

3.1. A Two-Sorted Variant T(;) of Gödel's T

We define a two-sorted variant T(;) of Gödel's T, by lifting the approach of Simmons (1988) and Bellantoni and Cook (1992) to higher types. It is shown that the functions definable in T(;) are exactly the elementary functions. The proof is based on the observation that β-normalization of terms of rank $\leq k$ has elementary complexity, and that the two-sortedness restriction allows to unfold $\mathcal{R}$ in a controlled way.

The elementary variant of Gödel's T developed in 3.1 has many relatives in the literature.

Beckmann and Weiermann (1996) characterize the elementary functions by means of a restriction of the combinatory logic version of Gödel's T. The restriction consists in allowing occurrences of the iteration operator only when immediately applied to a type $\mathbf{N}$ argument. For the proof they use an ordinal assignment due to Howard (1970) and Schütte (1977). The authors remark (on p. 477) that the methods of their paper can also be applied to a λ-formulation of T: the restriction on terms then consists in allowing only iterators of the form $\mathcal{I}_\rho t^{\mathbf{N}}$ and in disallowing λ-abstraction of the form $\lambda_x \dots \mathcal{I}_\rho t^{\mathbf{N}} \dots$ where x occurs in $t^{\mathbf{N}}$; however, no details are given. Moreover, our restrictions are slightly more liberal (input variables in t *can* be abstracted), and also the proof method is very different.

Aehlig and Johannsen (2005) characterize the elementary functions by means of a fragment of Girard's system F. They make essential use of the Church style representation of numbers in F. A somewhat different approach for characterizing the elementary functions based on a "predicative" setting has been developed by Leivant (1994).

3.1.1. Higher order terms with input/output restrictions. We shall work with two forms of arrow types and abstraction terms:

$$\begin{cases} \mathbf{N} \to \sigma \\ \lambda_n r \end{cases} \quad \text{as well as} \quad \begin{cases} \rho \multimap \sigma \\ \lambda_z r \end{cases}$$

and a corresponding syntactic distinction between input and output (typed) variables. Formally we proceed as follows. The *types* are

$$\rho, \sigma, \tau ::= \mathbf{N} \mid \mathbf{N} \to \rho \mid \rho \multimap \sigma,$$

and the *level* of a type is defined by

$$l(\mathbf{N}) := 0,$$
$$l(\rho \to \sigma) := l(\rho \multimap \sigma) := \max\{l(\sigma), 1 + l(\rho)\}.$$

Ground types are the types of level 0, and a *higher* type is any type of level at least 1. The $\to$-free types are called *safe*. In particular, every ground type is safe.

The *constants* are $0\colon \mathbf{N}$ and

$$\mathrm{S}\colon \mathbf{N} \multimap \mathbf{N},$$
$$\mathcal{R}_\tau\colon \mathbf{N} \to \tau \multimap (\mathbf{N} \to \tau \multimap \tau) \multimap \tau \quad (\tau \text{ safe}).$$

The restriction to safe types τ is needed in the proof of the Normalization Theorem below. Generally, the typing of $\mathcal{R}_\tau$ with its peculiar choices of $\to$ and $\multimap$ deserves some comments. The first argument is the one that is recursed on and hence must be an input argument, so the type starts with $\mathbf{N} \to \cdots$. The third argument is the step argument; here we have used the type $\mathbf{N} \to \tau \multimap \tau$ rather than $\mathbf{N} \multimap \tau \multimap \tau$, because then we can construct a step term in the form $\lambda_{n,p} t$ rather than $\lambda_{a,p} t$, which is more flexible.

We shall work with typed variables. A variable of type $\mathbf{N}$ is either an *input* or an *output* variable; variables of a type different from $\mathbf{N}$ are always output variables. We use the following conventions:

x	(input or output) variable;
z	output variable;
n, m	input variable of type $\mathbf{N}$;
a	output variable of type $\mathbf{N}$.

T(;)-*terms* (terms for short) are

$$r, s, t ::= x \mid C \mid (\lambda_n r)^{\mathbf{N}\to\sigma} \mid r^{\mathbf{N}\to\sigma} s^{\mathbf{N}} \ (s \text{ input term}) \mid$$
$$(\lambda_z r)^{\rho\multimap\sigma} \mid r^{\rho\multimap\sigma} s^{\rho}.$$

We call s an *input term* if all its free variables are input variables. C is a constant.

The *size* (or *length*) $|r|$ of a term r is the number of occurrences of constructors, variables and constants in r: $|x| = |C| = 1$, $|\lambda_n r| = |\lambda_z r| = |r| + 1$, and $|rs| = |r| + |s| + 1$.

3.1.2. β-normalization. In this section, the distinction between input and output variables and our two type formers $\to$ and $\multimap$ plays no role.

We are interested in the following process of simplification of terms:

$$\big(\lambda_{\vec{x},x} r(\vec{x}, x)\big)\vec{s}s \mapsto \big(\lambda_{\vec{x}} r(\vec{x}, s)\big)\vec{s}. \tag{3.1}$$

Terms of the form $(\lambda_{\vec{x},x} r(\vec{x}, x))\vec{s}s$ are called *β-convertible*. (3.1) is called the (generalized) *β-conversion* rule. Later we will also consider conversion rules for the recursion operator:

$$\begin{aligned} \mathcal{R}_\tau 0ts &\mapsto t, \\ \mathcal{R}_\tau(\mathrm{S}n)ts &\mapsto sn(\mathcal{R}_\tau nts). \end{aligned}$$

Note that converting $(\lambda_{\vec{x},x} r(\vec{x}, x))\vec{s}s$ into $(\lambda_{\vec{x}} r(\vec{x}, s))\vec{s}$ may be viewed as first converting $(\lambda_{\vec{x},x} r)\vec{s}s$ "permutatively" into $(\lambda_{\vec{x}}((\lambda_x r)s))\vec{s}$ and then performing the inner conversion to obtain $(\lambda_{\vec{x}}(r[x := s]))\vec{s}$. One may ask why we take this conversion relation as our basis and not the more common $(\lambda_x r(x))s \mapsto r(s)$. The reason is that our notion of level is defined with the clause $l(\rho \to \sigma) := l(\rho \multimap \sigma) := \max\{l(\sigma), 1 + l(\rho)\}$ and not $:= \max\{l(\sigma), l(\rho)\} + 1$; this in turn seems reasonable since then the level of $\rho_1, \dots, \rho_m \to \sigma$ (i.e., of $(\rho_1 \to (\rho_2 \to \dots (\rho_m \to \sigma) \dots))$) is 1 and hence independent of m. But given this definition of level, and given the need in some arguments (e.g., in the proof of the β-normalization theorem below) to perform conversions of highest level first, we must be able to convert $(\lambda_{\vec{x},x} r(\vec{x}, x))\vec{s}s$ with $\vec{x}$ of a low and x of a high level into $(\lambda_{\vec{x}} r(\vec{x}, s))\vec{s}$.

β-redexes are instances of the left side of (3.1). We write $r \to r'$ ($r \to^* r'$) if r can be reduced into r' by one (an arbitrary number of) β-conversion of a subterm. A term is said to be in *β normal form* if it does not contain a β-redex.

We want to show that every term reduces to a β normal form. This can be seen easily if we follow a certain order in our conversions. To define this order we have to make use of the fact that all our terms have types.

A β-convertible term

$$\big(\lambda_{\vec{x},x} r(\vec{x}^{\vec{\rho}}, x^\rho)\big)\vec{s}s$$

is also called a *cut* with *cut-type* ρ. By the level of a cut we mean the level of its cut-type. The *cut-rank* of a term r is the least number bigger than the levels of all cuts in r. Now let t be a term of cut-rank $k + 1$. Pick a cut of the maximal level k in t, such that s does not contain another cut of level k. (E.g., pick the rightmost cut of level k.) Then it is easy to see

that replacing the picked occurrence of $(\lambda_{\vec{x},x} r(\vec{x}^{\vec{\rho}}, x^{\rho}))\vec{s}s$ in t by $(\lambda_{\vec{x}} r(\vec{x}, s))\vec{s}$ reduces the number of cuts of the maximal level k in t by 1. Hence

THEOREM (β-Normalization). *We have an algorithm which reduces any given term into a β normal form.*

We now want to give an estimate of the number of conversion steps our algorithm takes until it reaches the normal form. The key observation for this estimate is the obvious fact that replacing one occurrence of

$$\big(\lambda_{\vec{x},x} r(\vec{x}, x)\big)\vec{s}s \quad \text{by} \quad \big(\lambda_{\vec{x}} r(\vec{x}, s)\big)\vec{s}.$$

in a given term t at most squares the size of t.

A bound $E_k(l)$ for the number of steps our algorithm takes to reduce the rank of a given term of size l by k can be derived inductively, as follows. Let $E_0(l) := 0$. To obtain $E_{k+1}(l)$, first note that by induction hypothesis it takes $\leq E_k(l)$ steps to reduce the rank by k. The size of the resulting term is $\leq l^{2^s}$ where $s := E_k(l)$ since any step (i.e., β-conversion) at most squares the size. Now to reduce the rank by one more, we convert – as described above – one by one all cuts of the present rank, where each such conversion does not produce new cuts of this rank. Therefore the number of additional steps is bounded by the size s. Hence the total number of steps to reduce the rank by $k+1$ is bounded by

$$E_k(l) + l^{2^{E_k(l)}} =: E_{k+1}(l).$$

THEOREM (Upper bound for the complexity of β-normalization). *The β-normalization algorithm given in the proof above takes at most $E_k(l)$ steps to reduce a given term of cut-rank k and size l to normal form, where*

$$E_0(l) := 0 \quad \textit{and} \quad E_{k+1}(l) := E_k(l) + l^{2^{E_k(l)}}.$$

3.1.3. Examples. A function f is called *definable in* T(;) if there is a closed T(;)-term $r\colon \mathbf{N} \twoheadrightarrow \ldots \mathbf{N} \twoheadrightarrow \mathbf{N}$ ($\twoheadrightarrow \in \{\to, \multimap\}$) in T(;) denoting this function.

We show that in spite of our restrictions on the formation of types and terms we can define functions of exponential growth. By other examples we explain how our restrictions prevent obtaining superelementary growth.

Probably the easiest function of exponential growth is $B(n, a) = a + 2^n$ of type $B\colon \mathbf{N} \to \mathbf{N} \multimap \mathbf{N}$, with the defining equations

$$B(0, a) = a + 1,$$
$$B(n+1, a) = B(n, B(n, a)).$$

We formally define B as a term in T(;) by

$$B := \lambda_n\big(\mathcal{R}_{\mathbf{N}\multimap\mathbf{N}} n \mathrm{S}\big(\lambda_{m,p,a}(p^{\mathbf{N}\multimap\mathbf{N}}(pa))\big)\big).$$

From B we can define the exponential function $E := \lambda_n(Bn0)$ of type $E\colon \mathbf{N} \to \mathbf{N}$, and also iterated exponential functions like $\lambda_n(E(En))$.

Now consider iteration $I(n,f) = f^n$, with f a variable of type $\mathbf{N} \multimap \mathbf{N}$.

$$\begin{array}{rcl} I(0,f,a) := a, & & I(0,f) := \mathrm{id}, \\ I(n+1,f,a) := I(n,f,f(a)), & \text{or} & I(n+1,f) := I(n,f) \circ f. \end{array}$$

Formally, for every variable f of type $\mathbf{N} \multimap \mathbf{N}$ we have the term

$$I_f := \lambda_n\big(\mathcal{R}_{\mathbf{N}\multimap\mathbf{N}} n(\lambda_a a)\big(\lambda_{m,p,a}(p^{\mathbf{N}\multimap\mathbf{N}}(fa))\big)\big).$$

For the general definition we need the *pure safe types* ρ_k, defined by $\rho_0 := \mathbf{N}$ and $\rho_{k+1} := \rho_k \multimap \rho_k$. Then within $\mathrm{T}(;)$ we can define

$$Ina_k \dots a_0 := a_k^n a_{k-1} \dots a_0,$$

with a_k of type ρ_k. However, a definition $F_0 a_k \dots a_0 := Ia_0 a_k \dots a_0$ is *not* possible: Ia_0 is not allowed.

We now discuss the necessity of the restrictions on the type of $\mathcal{R}$. We must require that the value type is a safe type, for otherwise we could define

$$I_E := \lambda_n\big(\mathcal{R}_{\mathbf{N}\to\mathbf{N}} n(\lambda_m m)\big(\lambda_{n,p,m}(p^{\mathbf{N}\to\mathbf{N}}(Em))\big)\big),$$

and $I_E(n,m) = E^n(m)$, a function of superelementary growth.

We also need to require that the "previous"-variable is an output variable, because otherwise we could define

$$S := \lambda_n\big(\mathcal{R}_{\mathbf{N}} n0\big(\lambda_{n,m}(Em)\big)\big) \quad \text{(superelementary)}.$$

Then $S(n) = E^n(0)$.

3.1.4. Normalization. We show that we can also eliminate the recursion operator, and still have an elementary estimate on the time needed.

Lemma ($\mathcal{R}$ Elimination). *Let $t(\vec{x})$ be a β-normal term of safe type. There is an elementary function E_t such that: if $\vec{s}$ are safe type $\mathcal{R}$-free terms and the free variables of $t(\vec{s})$ are output variables of safe type, then in time $E_t(|\vec{s}|)$ (with $|\vec{s}| := \sum_i |s_i|$) one can compute an $\mathcal{R}$-free term $\mathrm{rf}(t;\vec{x};\vec{s})$ such that $t(\vec{s}) \to^* \mathrm{rf}(t;\vec{x};\vec{s})$.*

Proof. Induction on t.

If $t(\vec{x})$ has the form $\lambda_x u_1$, then x is an output variable and x, u_1 have safe type because t has safe type. If $t(\vec{x})$ is of the form $D\vec{u}$ with D a variable or a constant different from $\mathcal{R}$, then each u_i is a safe type term. Here (in case D is a variable) we need that $\vec{x}$ and the free variables of $t(\vec{s})$ are of safe type.

In all of the preceding cases, the free variables of each $u_i(\vec{s})$ are output variables of safe type. Apply the IH to obtain $u_i^* := \mathrm{rf}(u_i;\vec{x};\vec{s})$. Let t^* be obtained from t by replacing each u_i by u_i^*. Then t^* is $\mathcal{R}$-free. The result

is obtained in linear time from $\vec{u}^*$. This finishes the lemma in all of these cases.

The only remaining case is if t is an $\mathcal{R}$ clause. Then it is of the form $\mathcal{R}rust\vec{}$, because the term has safe type. One obtains $\mathrm{rf}(r;\vec{x};\vec{s})$ in time $E_r(|\vec{s}|)$ by the IH. By assumption $t(\vec{s})$ has free output variables only. Hence $r(\vec{s})$ is closed, because the type of $\mathcal{R}$ requires $r(\vec{s})$ to be an input term. By β-normalization one obtains the number $N := \mathrm{nf}(\mathrm{rf}(r;\vec{x};\vec{s}))$ in a further elementary time, $E'_r(|\vec{s}|)$.

Now consider sn with a new variable n, and let s' be its β normal form. Since s is β-normal, $|s'| \leq |s| + 1 < |t|$. Applying the IH to s' one obtains a monotone elementary bounding function E_{sn}. One computes all $s_i := \mathrm{rf}(s';\vec{x},n;\vec{s},i)$ $(i < N)$ in a total time of at most

$$\sum_{i<N} E_{sn}(|\vec{s}| + i) \leq E'_r(|\vec{s}|) \cdot E_{sn}(|\vec{s}| + E'_r(|\vec{s}|)).$$

Consider u, $\vec{t}$. The IH gives $\hat{u} := \mathrm{rf}(u;\vec{x};\vec{s})$ in time $E_u(|\vec{s}|)$, and all $\hat{t}_i := \mathrm{rf}(t_i;\vec{x};\vec{s})$ in time $\sum_i E_{t_i}(|\vec{s}|)$. These terms are also $\mathcal{R}$-free by IH.

Using additional time bounded by a polynomial P in the lengths of these computed values, one constructs the $\mathcal{R}$-free term

$$\mathrm{rf}(\mathcal{R}rust\vec{};\vec{x};\vec{s}) := \big(s_{N-1}\ldots(s_1(s_0\hat{u}))\ldots\big)\vec{\hat{t}}.$$

Defining $E_t(l) := P(E_u(l) + \sum_i E_{t_i}(l) + E'_r(l) \cdot E_{sn}(l + E'_r(l)))$, the total time used in this case is at most $E_t(|\vec{s}|)$. □

We can now combine our results and state the final theorem. Let the *$\mathcal{R}$-rank* of a term t be the least number bigger than the level of all value types τ of recursion operators $\mathcal{R}_\tau$ in t. By the *rank* of a term we mean the maximum of its cut-rank and its $\mathcal{R}$-rank.

THEOREM. *For every k there is an elementary function N_k such that every term t of* T(;) *of rank $\leq k$ can be reduced in time $N_k(|t|)$ to normal form.*

3.2. A Linear Two-Sorted Variant LT(;) of Gödel's T

We now add some linearity restrictions, which will allow us to characterize the polynomial-time computable functions as those definable in a certain fragment of Gödel's T. The exposition in based on Bellantoni et al. (2000) and Schwichtenberg and Bellantoni (2002).

When discussing polynomial time, it is appropriate to work with a *binary* (rather than unary) representation of the natural numbers, with two successors $S_0(a) = 2a$ and $S_1(a) = 2a + 1$.

Recall that in the first example above of a recursion producing exponential growth, the definition of $B(n,a) = a + 2^n$, we had the defining term

$$B := \lambda_n\big(\mathcal{R}_{\mathbf{N}\multimap\mathbf{N}} n \mathrm{S}\big(\lambda_{m,p,a}(p^{\mathbf{N}\multimap\mathbf{N}}(pa))\big)\big)$$

with the higher type variable p for the "previous" value appearing twice in the step term. The linearity restriction will forbid this.

We essentially keep the definitions of types, safe types, input/output variables from 3.1.1. However, the term definition will be different: it now involves a linearity constraint. Moreover, the typing of the recursion operator $\mathcal{R}$ needs to be changed: its (higher type) step argument will be used many times, and hence we need a $\to$ after it. As a consequence, we now allow higher types as argument types for $\to$. Therefore we change the names of input/output variables into normal/safe variables.

3.2.1. Feasible computation with higher types. We shall work with two forms of arrow types and abstraction terms:

$$\begin{cases} \rho \to \sigma \\ \lambda_{\bar{x}^\rho} r \end{cases} \quad \text{as well as} \quad \begin{cases} \rho \multimap \sigma \\ \lambda_{x^\rho} r \end{cases}$$

and a corresponding syntactic distinction between normal and safe (typed) variables, $\bar{x}$ and x. The intuition is that a function of type $\rho \to \sigma$ may recurse on its argument (if it is of ground type) or use it many times (if it is of higher type), whereas a function of type $\rho \multimap \sigma$ is not allowed to recurse on its argument (if it is of ground type) or can use it only once (if it is of higher type). As is well known, we then need a corresponding distinction for product types: the ordinary product $\wedge$ for $\to$, and the tensor product $\otimes$ for the linear arrow $\multimap$. Formally we proceed as follows. The *types* are

$$\rho, \sigma, \tau ::= \mathbf{U} \mid \mathbf{B} \mid \mathbf{L}(\rho) \mid \rho \to \sigma \mid \rho \multimap \sigma \mid \rho \wedge \sigma \mid \rho \otimes \sigma,$$

and the *level* of a type is defined by

$$\begin{aligned} l(\mathbf{U}) &:= 0, \\ l(\mathbf{B}) &:= 0, \\ l(\mathbf{L}(\rho)) &:= l(\rho), \end{aligned} \qquad \begin{aligned} l(\rho \to \sigma) := l(\rho \multimap \sigma) &:= \max\{l(\sigma), 1 + l(\rho)\}, \\ l(\rho \wedge \sigma) := l(\rho \otimes \sigma) &:= \max\{l(\rho), l(\sigma)\}. \end{aligned}$$

Ground types are the types of level 0, and a *higher* type is any type of level at least 1. The $\to$-free types are also called *safe*. In particular, every ground type is safe.

The *constants* are $\mathbf{u}\colon \mathbf{U}$, $\mathtt{tt}, \mathtt{ff}\colon \mathbf{B}$, $\mathrm{nil}_\rho\colon \mathbf{L}(\rho)$ and

$$
\begin{aligned}
&::_\rho\colon \rho \multimap \mathbf{L}(\rho) \multimap \mathbf{L}(\rho),\\
&\mathrm{if}_\tau\colon \mathbf{B} \multimap \tau \wedge \tau \multimap \tau \quad (\tau \text{ safe}),\\
&\boldsymbol{c}^\rho_\tau\colon \mathbf{L}(\rho) \multimap \tau \wedge (\rho \multimap \mathbf{L}(\rho) \multimap \tau) \multimap \tau \quad (\tau \text{ safe}),\\
&\mathcal{R}^\rho_\tau\colon \mathbf{L}(\rho) \to \tau \multimap (\rho \multimap \mathbf{L}(\rho) \to \tau \multimap \tau) \to \tau \quad (\rho \text{ ground}, \tau \text{ safe}).
\end{aligned}
$$

The restriction to safe types τ is needed in the proof of the Normalization Theorem below (in 3.2.4). $\boldsymbol{c}^\rho_\tau$ is used for definition by cases (on the constructor form of a list), and $\mathcal{R}^\rho_\tau$ as a recursion operator. Note that a single recursion operator (over lists) is used here to cover both, numeric and word recursion.

The typing of $\mathcal{R}^\rho_\tau$ with its peculiar choices of $\to$ and $\multimap$ deserves some comments. The first argument is the one that is recursed on and hence must be normal, so the type starts with $\mathbf{L}(\rho) \to \cdots$. The third argument is for the step term, which is of a higher type and will be used many times (when the recursion operator is unfolded), so it must be normal as well. Hence we need a $\to$ after the step type. We will crucially need this typing when we prove (in the Sufficiency Lemma below) that the functions definable in LT(;) are closed under "safe recursion".

Further constants are, for safe ρ, σ, τ,

$$
\begin{aligned}
&\otimes^+_{\rho\sigma} &&: \rho \multimap \sigma \multimap \rho \otimes \sigma,\\
&\otimes^-_{\rho\sigma\tau} &&: \rho \otimes \sigma \multimap (\rho \multimap \sigma \multimap \tau) \multimap \tau,\\
&\wedge^+_{\rho\sigma} &&: \rho \multimap \sigma \multimap \rho \wedge \sigma \qquad \text{if } \rho, \sigma \text{ ground},\\
&\wedge^+_{\rho\sigma\tau} &&: (\tau \multimap \rho) \multimap (\tau \multimap \sigma) \multimap \tau \multimap \rho \wedge \sigma \qquad \text{if } l(\rho \wedge \sigma) > 0,\\
&\mathrm{fst}_{\rho\sigma} &&: \rho \wedge \sigma \multimap \rho,\\
&\mathrm{snd}_{\rho\sigma} &&: \rho \wedge \sigma \multimap \sigma.
\end{aligned}
$$

The restriction to safe types ρ, σ, τ again will be needed in the proof of the Normalization Theorem. The type of $\wedge^+_{\rho\sigma\tau}$ can be explained as follows. In our linear setting, using a term of type $\rho \wedge \sigma$ might be allowed only once. So if one picks one component, the other one is lost. Therefore it is perfectly legal to have an occurrence of a higher type safe variable in both components. Now the type of $\wedge^+_{\rho\sigma\tau}$ allows such duplications, via the argument of type τ. For ρ, σ both ground duplication is no problem, so we can use the simpler $\wedge^+_{\rho\sigma}$ in this case.

Definition. LT(;)-*terms* (terms for short) are built from these constants and typed variables $\bar{x}^\sigma$ (normal variables) and x^σ (safe variables) by introduction and elimination rules for the two type forms $\rho \to \sigma$ and $\rho \multimap \sigma$,

i.e.,

$$
\begin{array}{l}
\bar{x}^\rho \quad \text{(normal variable)} \mid \\
x^\rho \quad \text{(safe variable)} \mid \\
C^\rho \quad \text{(constant)} \mid \\
(\lambda_{\bar{x}^\rho} r^\sigma)^{\rho\to\sigma} \mid \\
(r^{\rho\to\sigma} s^\rho)^\sigma \quad (s \text{ ``normal''}) \mid \\
(\lambda_{x^\rho} r^\sigma)^{\rho\multimap\sigma} \mid \\
(r^{\rho\multimap\sigma} s^\rho)^\sigma \quad \text{(higher type safe variables in } r, s \text{ distinct)},
\end{array}
$$

where a term s is called *normal* if all its free variables are normal.

By the restriction on safe variables in the formation of an application $r^{\rho\multimap\sigma} s$, every higher type safe variable can occur at most once in a given term.

The *conversion* rules are as expected: β-conversion (for normal and safe variables) plus

$$
\begin{array}{lll}
\mathrm{if}_\tau \mathtt{tt} s & \mapsto \mathrm{fst}_{\tau\tau} s, & \\
\mathrm{if}_\tau \mathtt{ff} s & \mapsto \mathrm{snd}_{\tau\tau} s, & \\
\boldsymbol{c}^\rho_\tau \mathrm{nil}_\rho s & \mapsto \mathrm{fst}_{\tau\sigma} s & \text{for } \sigma := \rho \multimap \mathbf{L}(\rho) \multimap \tau, \\
\boldsymbol{c}^\rho_\tau (r ::_\rho l) s & \mapsto \mathrm{snd}_{\tau\sigma} s r l & \text{for } \sigma := \rho \multimap \mathbf{L}(\rho) \multimap \tau, \\
\mathcal{R}^\rho_\tau \mathrm{nil}_\rho t s & \mapsto t, & \\
\mathcal{R}^\rho_\tau (r ::_\rho l) t s & \mapsto s r l (\mathcal{R}^\rho_\tau l t s), & \\
\otimes^-_{\rho\sigma\tau} (\otimes^+_{\rho\sigma} r s) t & \mapsto t r s, & \\
\mathrm{fst}_{\rho\sigma} (\wedge^+_{\rho\sigma} r s) & \mapsto r, & \\
\mathrm{snd}_{\rho\sigma} (\wedge^+_{\rho\sigma} r s) & \mapsto s, & \\
\mathrm{fst}_{\rho\sigma} (\wedge^+_{\rho\sigma\tau} r s t) & \mapsto r t, & \\
\mathrm{snd}_{\rho\sigma} (\wedge^+_{\rho\sigma\tau} r s t) & \mapsto s t. &
\end{array}
$$

Redexes are subterms shown on the left side of the conversion rules above. We write $r \to r'$ ($r \to^* r'$) if r can be reduced into r' by one (an arbitrary number of) conversion of a subterm.

Note that projections w.r.t. $\rho \otimes \sigma$ can be defined easily: For a term t of type $\rho \otimes \sigma$ let

$$t0 := \otimes^-_{\rho\sigma\rho} t (\lambda_{x^\rho, y^\sigma} x) \quad \text{and} \quad t1 := \otimes^-_{\rho\sigma\sigma} t (\lambda_{x^\rho, y^\sigma} y).$$

Then clearly

$$(\otimes^+_{\rho\sigma} rs)0 = \otimes^-_{\rho\sigma\rho}(\otimes^+_{\rho\sigma} rs)(\lambda_{x^\rho,y^\sigma} x) \mapsto (\lambda_{x^\rho,y^\sigma} x)rs \to^* r,$$
$$(\otimes^+_{\rho\sigma} rs)1 = \otimes^-_{\rho\sigma\sigma}(\otimes^+_{\rho\sigma} rs)(\lambda_{x^\rho,y^\sigma} y) \mapsto (\lambda_{x^\rho,y^\sigma} y)rs \to^* s.$$

A function f is called *definable in* LT(;) if there is a closed LT(;)-term $r\colon \mathbf{W} \twoheadrightarrow \dots \mathbf{W} \twoheadrightarrow \mathbf{W}$ ($\twoheadrightarrow \in \{\to, \multimap\}$) in LT(;) denoting this function.

3.2.2. Examples. We now look at some examples intended to explain how our restrictions on the formation of types and terms make it impossible obtain exponential growth. However, for definiteness we first have to say precisely what we mean by a *numeral*.

Terms of the form $r^\rho_1 ::_\rho (r^\rho_2 ::_\rho \dots (r^\rho_n ::_\rho \mathrm{nil}_\rho)\dots)$ are called *lists*. We abbreviate $\mathbf{N} := \mathbf{L}(\mathbf{U})$ and $\mathbf{W} := \mathbf{L}(\mathbf{B})$.

$$0 := \mathrm{nil}_{\mathbf{U}}, \qquad 1 := \mathrm{nil}_{\mathbf{B}},$$
$$\mathrm{S} := \lambda_l.\mathbf{u} :: l^{\mathbf{N}}, \qquad S_0 := \lambda_l.\mathtt{ff} :: l^{\mathbf{W}},$$
$$S_1 := \lambda_l.\mathtt{tt} :: l^{\mathbf{W}}.$$

Particular lists are $\mathrm{S}(\dots(\mathrm{S}0)\dots)$ and $S_{i_1}(\dots(S_{i_n}1)\dots)$. The former are called *unary numerals*, and the latter *binary numerals* (or *numerals of type* $\mathbf{W}$). We denote binary numerals by ν.

Two recursions. Consider

$$D(1) := S_0(1), \qquad E(1) := 1,$$
$$D(S_i(x)) := S_0(S_0(D(x))), \qquad E(S_i(x)) := D(E(x)).$$

The corresponding terms are

$$D := \lambda_{\bar{x}}.\mathcal{R}_{\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p.S_0(S_0p))(S_01),$$
$$E := \lambda_{\bar{x}}.\mathcal{R}_{\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p.Dp)1.$$

Here D is legal, but E is not: the application Dp is not allowed.

Recursion with parameter substitution. Consider

$$E(1,y) := S_0(y), \qquad E(1) := S_0,$$
$$E(S_i(x),y) := E(x,E(x,y)), \quad \text{or} \quad E(S_i(x)) := E(x) \circ E(x).$$

The corresponding term

$$\lambda_{\bar{x}}.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p{}^{\mathbf{W}\multimap\mathbf{W}}\lambda_y.p(py))S_0$$

does not satisfy the linearity condition: the higher type variable p occurs twice, and the typing of $\mathcal{R}$ requires p to be safe.

A different form of recursion with parameter substitution is

$$E(1,y) := y, \qquad E(1) := \mathrm{id},$$
$$E(S_i(x),y) := E(x,D(y)), \quad \text{or} \quad E(S_i(x)) := E(x) \circ D.$$

The corresponding term would be

$$\lambda_{\bar{x}}.\mathcal{R}_{\mathbf{W}\to\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p{}^{\mathbf{w}\to\mathbf{w}}\lambda_{\bar{x}}.p(D\bar{x}))(\lambda_y y),$$

but it is not legal, since the result type is not safe.

Higher argument types: iteration. Consider

$$\begin{aligned} I(1,f,y) &:= y, \\ I(S_i(x),f,y) &:= f(I(x,f,y)), \end{aligned} \quad \text{or} \quad \begin{aligned} I(1,f) &:= \mathrm{id}, \\ I(S_i(x),f) &:= f \circ I(x,f) \end{aligned}$$

with the corresponding term

$$\begin{aligned} I_f &:= \lambda_{\bar{x}}.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p{}^{\mathbf{w}\multimap\mathbf{w}}\lambda_y.f(py))(\lambda_y y), \\ E &:= \lambda_x.I_D x1. \end{aligned}$$

Here I_f is legal, but E is not: the type of D prohibits iteration. – Note that in PV^ω (see Cook and Kapron (1990), Cook (1992)) I is *not* definable, for otherwise we could define $\lambda_z.IDz$.

A related phenomenon occurrs in

$$\begin{aligned} E(1) &:= \mathrm{id}, \\ E(S_i(x)) &:= E(x) \circ D. \end{aligned}$$

Now the terms are

$$\begin{aligned} I_f &:= \lambda_x.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p{}^{\mathbf{w}\multimap\mathbf{w}}\lambda_y.f(py))(\lambda_y y), \\ E &:= \lambda_{\bar{x}}.\mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_q{}^{\mathbf{w}\multimap\mathbf{w}}.I_q(S_0(S_0 1)))S_0. \end{aligned}$$

Again E is not legal, this time because the free parameter f in the step term of I_f is substituted with the *safe* variable q. This variable needs to be normal because of the typing of the recursion operator.

3.2.3. Polynomials. It is high time that we give some examples of what *can* de done in our term system. It is easy to define $\oplus\colon \mathbf{W} \to \mathbf{W} \multimap \mathbf{W}$ such that $x \oplus y$ concatenates $|x|$ bits onto y:

$$\begin{aligned} 1 \oplus y &= S_0 y, \\ (S_i x) \oplus y &= S_0(x \oplus y). \end{aligned}$$

The representing term is

$$\bar{x} \oplus y := \mathcal{R}_{\mathbf{W}\multimap\mathbf{W}}\bar{x}(\lambda_z\lambda_{\bar{l}}\lambda_p{}^{\mathbf{w}\multimap\mathbf{w}}\lambda_y.S_0(py))S_0 y.$$

Similarly we define $\odot\colon \mathbf{W} \to \mathbf{W} \to \mathbf{W}$ such that $x \odot y$ has output length $|x| \cdot |y|$:

$$\begin{aligned} x \odot 1 &= x, \\ x \odot (S_i y) &= x \oplus (x \odot y). \end{aligned}$$

The representing term is $\bar{x} \odot \bar{y} := \mathcal{R}_{\mathbf{W}}\bar{y}(\lambda_z\lambda_{\bar{l}}\lambda_p.\bar{x} \oplus p)\bar{x}$.

Note that the typing $\oplus\colon \mathbf{W} \to \mathbf{W} \multimap \mathbf{W}$ is crucial: it allows using the safe variable p in the definition of $\odot$. If we try to go on and define exponentiation from multiplication $\odot$ just as $\odot$ was defined from $\oplus$, we find out that we cannot go ahead, because of the different typing $\odot\colon \mathbf{W} \to \mathbf{W} \to \mathbf{W}$.

3.2.4. Normalization. A *dag* is a directed acyclic graph. A *parse dag* is a structure like a parse tree but admitting in-degree greater than one. For example, a parse dag for $\lambda_x r$ has a node containing λ_x and a pointer to a parse dag for r. A parse dag for (rs) has a node containing a pair of pointers, one to a parse dag for r and the other to a parse dag for s. Terminal nodes are labeled by constants and variables.

The *size* $|d|$ of a parse dag d is the number of nodes in it, but counting 3 for $\boldsymbol{c}_\tau$ nodes. Starting at any given node in the parse dag, one obtains a term by a depth-first traversal; it is the term *represented* by that node. We may refer to a node as if it were the term it represents.

A parse dag is *conformal* if (i) every node having in-degree greater than 1 is of ground type, and (ii) every maximal path to a bound variable x passes through the same binding λ_x node.

A parse dag is *h-affine* if every higher-type variable occurs at most once in the dag.

We adopt a model of computation over parse dags in which operations such as the following can be performed in unit time: creation of a node given its label and pointers to the sub-dags; deletion of a node; obtaining a pointer to one of the subsidiary nodes given a pointer to an interior node; conditional test on the type of node or on the constant or variable in the node. Concerning computation over terms (including numerals), we use the same model and identify each term with its parse tree. Although not all parse dags are conformal, every term is conformal (assuming a relabeling of bound variables).

A term is called *simple* if contains no higher type normal variables. Obviously simple terms are closed under reductions, taking of subterms, and applications. Every simple term is h-affine, due to the linearity of safe higher-type variables.

Lemma (Simplicity). *Let t be a ground type term whose free variables are of ground type. Then* $\mathrm{nf}(t)$ *contains no higher type normal variables.*

Proof. Suppose a variable $\bar{x}^\sigma$ with $l(\sigma) > 0$ occurs in $\mathrm{nf}(t)$. It must be bound in a subterm $(\lambda_{\bar{x}^\sigma} r)^{\sigma\to\tau}$ of $\mathrm{nf}(t)$. By the well known subtype property of normal terms, the type $\sigma \to \tau$ either occurs positively in the type of $\mathrm{nf}(t)$, or else negatively in the type of one of the constants or free

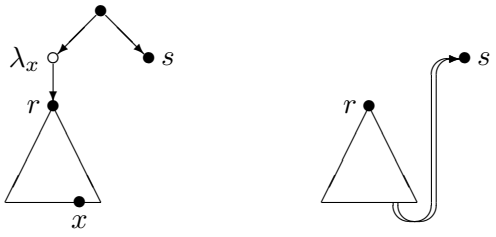

FIGURE 1. Redex $(\lambda_x r)s$ with r of ground type

variables of $\mathrm{nf}(t)$. The former is impossible since t is of ground type, and the latter by inspection of the types of the constants. □

LEMMA (Sharing Normalization). *Let t be an $\mathcal{R}$-free simple term. Then a parse dag for $\mathrm{nf}(t)$, of size at most $|t|$, can be computed from t in time $O(|t|^2)$.*

PROOF. Under our model of computation, the input t is a parse tree. Since t is simple, it is an h-affine conformal parse dag of size at most $|t|$. If there are no nodes which represent a redex, then we are done. Otherwise, locate a node representing a redex; this takes time at most $O(|t|)$. We show how to update the dag in time $O(|t|)$ so that the size of the dag has strictly decreased and the redex has been eliminated, while preserving conformality. Thus, after at most $|t|$ iterations the resulting dag represents the normal-form term $\mathrm{nf}(t)$. The total time therefore is $O(|t|^2)$.

Assume first that the redex in t is $(\lambda_x r)s$ with x of ground type (see Figure 1); the argument is similar for a normal variable $\bar{x}$. Replace pointers to x in r by pointers to s. Since s does not contain x, no cycles are created. Delete the λ_x node and the root node for $(\lambda_x r)s$ which points to it. By conformality (i) no other node points to the λ_x node. Update any node which pointed to the deleted node for $(\lambda_x r)s$, so that it now points to the revised r subdag. This completes the β reduction on the dag (one may also delete the x nodes). Conformality (ii) gives that the updated dag represents a term t' such that $t \to t'$.

One can verify that the resulting parse dag is conformal and h-affine, with conformality (i) following from the fact that s has ground type.

If the redex in t is $(\lambda_x r)s$ with x of higher type (see Figure 2), then x occurs at most once in r because the parse dag is h-affine. By conformality (i) there is at most one pointer to that occurrence of x. Update it to point to s instead, deleting the x node. As in the preceding case, delete the λ_x and the $(\lambda_x r)s$ node pointing to it, and update other nodes to point to the revised r. Again by conformality (ii) the updated dag represents t' such that

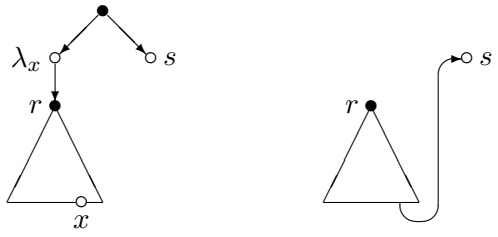

FIGURE 2. Redex $(\lambda_x r)s$ with r of higher type

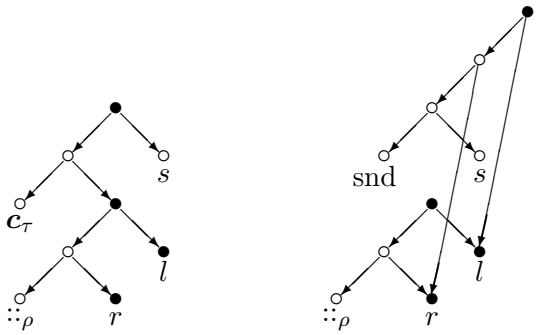

FIGURE 3. $\boldsymbol{c}_\tau(r ::_\rho l)s \mapsto \mathrm{snd}\, srl$ with ρ a ground type

$t \to t'$. Conformality and acyclicity are preserved, observing this time that conformality (i) follows because there is at most one pointer to s.

The remaining reductions are for the constant symbols.

Case $\mathrm{if}_\tau \mathtt{tt} s \mapsto \mathrm{fst}_{\tau\tau} s$. Easy; similar for $\mathtt{ff}$.

Case $\boldsymbol{c}_\tau \mathrm{nil}_\rho s \mapsto \mathrm{fst}\, s$. Easy.

Case $\boldsymbol{c}_\tau(r ::_\rho l)s \mapsto \mathrm{snd}\, srl$ with ρ a ground type (see Figure 3). Note that the new dag has one node more than the original one, but one $\boldsymbol{c}_\tau$-node less. Since we count the $\boldsymbol{c}_\tau$-nodes 3-fold, the total number of nodes decreases.

The remaining cases are treated in the Figures 4 – 7 below. Note that in the final one, where $\mathrm{fst}_{\rho\sigma}(\wedge^+_{\rho\sigma\tau} rst) \mapsto rt$, we need that $\rho \wedge \sigma$ in this case is not a ground type. The case $\mathrm{snd}_{\rho\sigma}(\wedge^+_{\rho\sigma\tau} rst) \mapsto st$ is of course similar. □

COROLLARY (Base Normalization). *Let t be a closed $\mathcal{R}$-free simple term of type $\mathbf{W}$. Then the binary numeral $\mathrm{nf}(t)$ can be computed from t in time $O(|t|^2)$, and $|\mathrm{nf}(t)| \leq |t|$.*

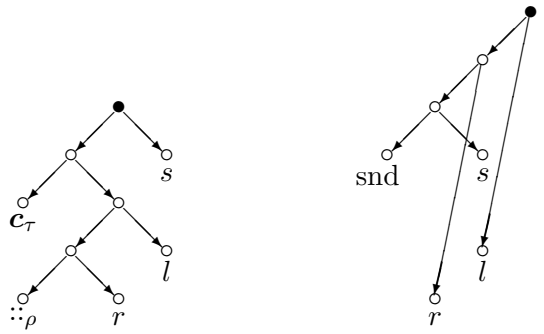

FIGURE 4. $\boldsymbol{c}_\tau(r ::_\rho l)s \mapsto \mathrm{snd}\, srl$ with ρ not a ground type.

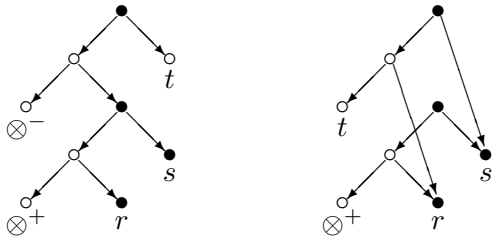

FIGURE 5. $\otimes^-_{\rho\sigma\tau}(\otimes^+_{\rho\sigma} rs)t \mapsto trs$ with $\rho \otimes \sigma$ a ground type.

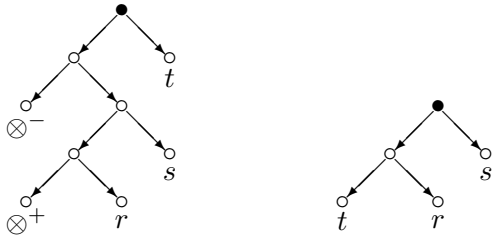

FIGURE 6. $\otimes^-_{\rho\sigma\tau}(\otimes^+_{\rho\sigma} rs)t \mapsto trs$ with $\rho \otimes \sigma$ not a ground type.

PROOF. By the Sharing Normalization Lemma we obtain a parse dag for $\mathrm{nf}(t)$ of size at most $|t|$, in time $O(|t|^2)$. Since $\mathrm{nf}(t)$ is a binary numeral, there is only one possible parse dag for it – namely, the parse tree of the numeral. This is identified with the numeral itself in our model of computation. □

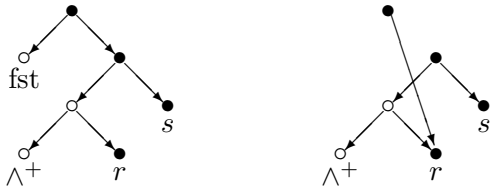

FIGURE 7. $\mathrm{fst}_{\rho\sigma}(\wedge^+_{\rho\sigma} rs) \mapsto r$ with $\rho \wedge \sigma$ a ground type.

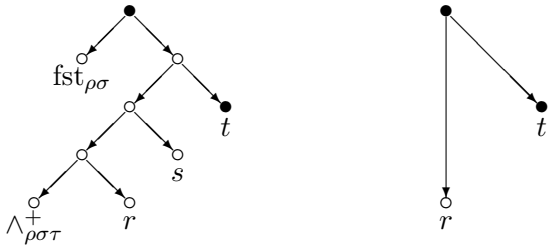

FIGURE 8. $\mathrm{fst}_{\rho\sigma}(\wedge^+_{\rho\sigma\tau} rst) \mapsto rt$

LEMMA ($\mathcal{R}$ Elimination). *Let $t(\vec{x}\,)$ be a simple term of safe type. There is a polynomial p_t such that: if $\vec{m}$ are safe type $\mathcal{R}$-free closed simple terms and the free variables of $t(\vec{m})$ are safe and of safe type, then in time $p_t(|\vec{m}|)$ one can compute an $\mathcal{R}$-free simple term $\mathrm{rf}(t;\vec{x};\vec{m})$ such that $t(\vec{m}) \to^* \mathrm{rf}(t;\vec{x};\vec{m})$.*

PROOF. By induction on $|t|$.

If $t(\vec{x}\,)$ has the form $\lambda_z u_1$, then z is safe and z, u_1 have safe type because t has safe type. If $t(\vec{x}\,)$ is of the form $D\vec{u}$ with D a variable or one of the symbols $\mathbf{u}$, $\mathtt{tt}$, $\mathtt{ff}$, nil_ρ, $::_\rho$, if_τ, $\boldsymbol{c}_\tau$, $\otimes^+_{\rho\sigma}$, $\otimes^-_{\rho\sigma\tau}$, $\wedge^+_{\rho\sigma\tau}$, $\mathrm{fst}_{\rho\sigma}$ or $\mathrm{snd}_{\rho\sigma}$, then each u_i is a safe type term. Here (in case D is a variable x_i) we need that x_i is safe.

In all of the preceding cases, each $u_i(\vec{m})$ has only safe free variables of safe type. Apply the IH as required to simple terms u_i to obtain $u_i^* := \mathrm{rf}(u_i;\vec{x};\vec{m})$; so each u_i^* is $\mathcal{R}$-free. Let t^* be obtained from t by replacing each u_i by u_i^*. Then t^* is an $\mathcal{R}$-free simple term; here we need that $\vec{m}$ are closed, to avoid duplication of variables. The result is obtained in linear time from $\vec{u}^*$. This finishes the lemma in all of these cases.

If t is $(\lambda_y r)s\vec{u}$ with a safe variable y of ground type, apply the IH to yield $(r\vec{u})^* := \mathrm{rf}(r\vec{u};\vec{x};\vec{m})$ and $s^* := \mathrm{rf}(s;\vec{x};\vec{m})$. Redirect the pointers to y in $(r\vec{u})^*$ to point to s^* instead. If t is $(\lambda\bar{y}r)s\vec{u}$ with a normal variable $\bar{y}$ of ground type, apply the IH to yield $s^* := \mathrm{rf}(s;\vec{x};\vec{m})$. Note that s^*

is closed, since it is normal and the free variables of $s(\vec{m})$ are safe. Then apply the IH again to obtain $\mathrm{rf}(r\vec{u}; \vec{x}, \bar{y}; \vec{m}, s^*)$. The total time is at most $q(|t|)+p_s(|\vec{m}|)+p_r(|\vec{m}|+p_s(|\vec{m}|))$, as it takes at most linear time to construct $r\vec{u}$ from $(\lambda_y r)s\vec{u}$.

If t is $(\lambda_y r(y))s\vec{u}$ with y of higher type, then y can occur at most once in r, because t is simple. Thus $|r(s)\vec{u}| < |(\lambda_y r)s\vec{u}|$. Apply the IH to obtain $\mathrm{rf}(r(s)\vec{u}; \vec{x}; \vec{m})$. Note that the time is bounded by $q(|t|) + p_{r(s)\vec{u}}(|\vec{m}|)$ for a degree one polynomial q, since it takes at most linear time to make the at-most-one substitution in the parse tree.

The only remaining case is if the term is an $\mathcal{R}$ clause. Then it is of the form $\mathcal{R}lst\vec{}$, because the term has safe type.

Since l is normal, all free variables of l are normal – they must be in $\vec{x}$ since free variables of $(\mathcal{R}lst\vec{})[\vec{x} := \vec{m}]$ are safe. Then $l(\vec{m})$ is closed, implying $\mathrm{nf}(l(\vec{m}))$ is a list. One obtains $\mathrm{rf}(l; \vec{x}; \vec{m})$ in time $p_l(|\vec{m}|)$ by the IH. Then by Base Normalization one obtains the list $\hat{l} := \mathrm{nf}(\mathrm{rf}(l; \vec{x}; \vec{m}))$ in a further polynomial time. Let $\hat{l} = r_0 ::_\rho (r_1 ::_\rho \ldots (r_N ::_\rho \mathrm{nil}_\rho)\ldots)$ and let l_i, $0 \le i \le N$ be obtained from $\hat{l}$ by omitting the initial elements $r_0, \ldots, r_i$. Thus all $\{\, r_i, l_i \mid i \le N \,\}$ are obtained in a total time bounded by $p'_l(|\vec{m}|)$ for a polynomial p'_l.

Now consider $sz\bar{y}$ with new variables z^ρ and $\bar{y}^{\mathbf{L}(\rho)}$. Applying the IH to $sz\bar{y}$ one obtains a monotone bounding polynomial $p_{sz\bar{y}}$. One computes all $s_i := \mathrm{rf}(sz\bar{y}; \vec{x}, z, \bar{y}; \vec{m}, r_i, l_i)$ in a total time of at most

$$\sum_{i=1}^{N} p_{sz\bar{y}}(|r_i| + |l_i| + |\vec{m}|) \le p'_l(|\vec{m}|) \cdot p_{sz\bar{y}}(2p'_l(|\vec{m}|) + |\vec{m}|).$$

Each s_i is $\mathcal{R}$-free by the IH. Furthermore, no s_i has a free safe variable: any such variable would also be free in s contradicting that s is normal.

Consider $\vec{t}$. The IH gives all $\hat{t}_i := \mathrm{rf}(t_i; \vec{x}; \vec{m})$ in time $\sum_i p_{t_i}(|\vec{m}|)$. These terms are also $\mathcal{R}$-free by IH. Clearly the t_i do not have any free (or bound) higher type safe variables in common. The same is true of all $\hat{t}_i$.

Using additional time bounded by a polynomial p in the lengths of these computed values, one constructs the $\mathcal{R}$-free term

$$\big(\lambda_x.s_0(s_1 \ldots (s_N x)\ldots)\big)\vec{\hat{t}}.$$

Defining $p_t(n) := p(\sum_i p_{t_i}(n) + p'_l(n) \cdot p_{sz\bar{y}}(2p'_l(n) + n))$, the total time used in this case is at most $p_t(|\vec{m}|)$. The result is a term because the $\hat{t}_i$ are terms which do not have any free higher-type safe variable in common and because s_i does not have any free higher-type safe variables at all. □

Theorem (Normalization). *Let r be a closed* $\mathrm{LT}(;)$*-term of type* $\mathbf{W} \twoheadrightarrow \ldots \mathbf{W} \twoheadrightarrow \mathbf{W}$ *($\twoheadrightarrow \in \{\to, \multimap\}$). Then r denotes a polytime function.*

PROOF. One must find a polynomial q_t such that for all $\mathcal{R}$-free simple closed terms $\vec{n}$ of types $\vec{\rho}$ one can compute $\mathrm{nf}(t\vec{n})$ in time $q_t(|\vec{n}|)$. Let $\vec{x}$ be new variables of types $\vec{\rho}$. The normal form of $t\vec{x}$ is computed in an amount of time that may be large, but it is still only a constant with respect to $\vec{n}$.

$\mathrm{nf}(t\vec{x})$ is simple by the Simplicity Lemma. By $\mathcal{R}$ Elimination one reduces to an $\mathcal{R}$-free simple term $\mathrm{rf}(\mathrm{nf}(t\vec{x}); \vec{x}; \vec{n})$ in time $p_t(|\vec{n}|)$. Since the running time bounds the size of the produced term, $|\mathrm{rf}(\mathrm{nf}(t\vec{x}); \vec{x}; \vec{n})| \leq p_t(|\vec{n}|)$.

By Sharing Normalization one can compute

$$\mathrm{nf}(t\vec{n}) = \mathrm{nf}(\mathrm{rf}(\mathrm{nf}(t\vec{x}); \vec{x}; \vec{n}))$$

in time $O(p_t(|\vec{n}|)^2)$. Let q_t be the polynomial referred to by the big-O notation. □

3.2.5. Sufficiency. The converse holds as well. The proof uses a characterization of the polynomial-time computable functions given by Bellantoni and Cook (1992). There the polynomial time computable functions are characterized by a function algebra B based on *safe recursion* and *safe composition*. There every function is written in the form $f(\vec{x}; \vec{y})$ where $\vec{x}; \vec{y}$ denotes a bookkeeping of those variables $\vec{x}$ that are used in a recursion defining f, and those variables $\vec{y}$ that are not recursed on. We proceed by induction on the definition of $f(x_1, \ldots, x_k; y_1, \ldots, y_l)$ in B, associating to f a closed term t_f of type $\mathbf{W}^{(k)} \to \mathbf{W}^{(l)} \multimap \mathbf{W}$, such that t denotes f.

The functions in B were defined over the non-negative integers rather than the positive ones, but this clearly is a minor point. We use the bijection $x \in \mathbb{N} \Leftrightarrow (2^{|x|} + x) \in \mathbb{Z}^+$.

LEMMA (Sufficiency). *Let f be a polynomial-time computable function. Then f is denoted by a closed* LT(;)*-term r.*

PROOF. If f in B is an initial function 0, S_0, S_1, P, conditional C or projection $\pi_i^{m,n}$, then t_f is easily defined.

If f is defined by safe composition in system B, then

$$f(\vec{x}; \vec{y}) := g(r_1(\vec{x};), \ldots, r_m(\vec{x};); s_1(\vec{x}; \vec{y}), \ldots, s_n(\vec{x}; \vec{y})).$$

Using the IH to obtain t_g, $t_{\vec{r}}$ and $t_{\vec{s}}$, define

$$t_f := \lambda_{\vec{x}} \lambda_{\vec{y}}. t_g(t_{r_1}\vec{x}) \ldots (t_{r_m}\vec{x})\, (t_{s_1}\vec{x}\vec{y}) \ldots (t_{r_m}\vec{x}\vec{y}).$$

Finally consider f defined by safe recursion in system B.

$$\begin{aligned} f(0, \vec{x}; \vec{y}) &:= g(\vec{x}; \vec{y}), \\ f(S_i n, \vec{x}; \vec{y}) &:= h_i(n, \vec{x}; \vec{y}, f(n, \vec{x}; \vec{y})) \quad \text{for } S_i n \neq 0. \end{aligned}$$

One has t_g, t_{h_0} and t_{h_1} by IH. Let p be a variable of type $\tau := \mathbf{W}^{(\#(\vec{y}))} \multimap \mathbf{W}$; this is the safe type used in the recursion. Then define a step term by

$$s := \lambda_x \lambda_{\bar{l}} \lambda_p \lambda_{\vec{y}} . \mathrm{if}_{\mathbf{W} \multimap \mathbf{W}} x \big(\wedge^+ (\lambda_z . t_{h_0} \bar{l} \vec{\bar{x}} \vec{y} z)(\lambda_z . t_{h_1} \bar{l} \vec{\bar{x}} \vec{y} z) \big) (p \vec{y}\,).$$

Note p is used only once. Let $t_f := \lambda_{\bar{n}} \lambda_{\vec{\bar{x}}} . \mathcal{R}_\tau \bar{n} s (t_g \vec{\bar{x}}\,)$. □

3.3. Towards Curry-Howard Extensions to Arithmetic

Curry and Howard observed that types correspond to formulas, and terms to proofs, when the logic is formulated in Gentzen's natural deduction calculus. Therefore it is tempting the transfer our restricted term systems to arithmetical theories, which then by construction have limited computational power: elementary arithmetic A(;) for T(;), and polynomial-time arithmetic LA(;) for LT(;). Initial attempts in this direction have already been carried out: Ostrin and Wainer (2005) for the elementary case, and Schwichtenberg (2006) for the polynomial-time case. There is also related work by Bellantoni and Hofmann (2002), which however uses a different approach based on the Hilbert calculus.

It remains to be seen whether such attempts to obtain feasible programs become feasible in practice. In any case, since such programs are automatically generated by extraction from checkable proofs, by their very construction the meet the highest possible security demands.

Bibliography

K. Aehlig and J. Johannsen. An elementary fragment of second-order lambda calculus. *ACM Transactions on Computational Logic*, 6(2):468–480, Apr. 2005.

A. Beckmann and A. Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Archive for Mathematical Logic*, 36:11–30, 1996.

S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

S. Bellantoni and M. Hofmann. A new "feasible" arithmetic. *The Journal of Symbolic Logic*, 67(1):104–116, 2002.

S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104: 17–30, 2000.

U. Berger. Uniform Heyting Arithmetic. *Annals Pure Applied Logic*, 133: 125–148, 2005.

U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

S. A. Cook. Computability and complexity of higher type functions. In Y. Moschovakis, editor, *Logic from Computer Science, Proceedings of a Workshop held November 13–17, 1989*, number 21 in MSRI Publications, pages 51–72. Springer Verlag, Berlin, Heidelberg, New York, 1992.

S. A. Cook and B. M. Kapron. Characterizations of the basic feasible functionals of finite type. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 71–96. Birkhäuser, 1990.

A. Dragalin. New kinds of realizability. In *Abstracts of the 6th International Congress of Logic, Methodology and Philosophy of Sciences*, pages 20–24, Hannover, Germany, 1979.

H. Friedman. Classically and intuitionistically provably recursive functions. In D. Scott and G. Müller, editors, *Higher Set Theory*, volume 669 of *Lecture Notes in Mathematics*, pages 21–28. Springer Verlag, Berlin, Heidelberg, New York, 1978.

K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkts. *Dialectica*, 12:280–287, 1958.

W. A. Howard. Assignment of ordinals to terms for primitive recursive functionals of finite type. In J. M. A. Kino and R. Vesley, editors, *Intuitionism and Proof Theory, Proceedings of the summer conference at Buffalo N.Y. 1968*, Studies in logic and the foundations of mathematics, pages 443–458. North–Holland, Amsterdam, 1970.

A. N. Kolmogorov. On the principle of the excluded middle (Russian). *Matematicheskij Sbornik. Akademiya Nauk SSSRi Moskovskoe Matematicheskoe Obshchestvo*, 32:646–667, 1925. Translated in J. van Heijenoort, From Frege to Gödel. A Source Book in Mathematical Logic 1879–1931, Harvard University Press, Cambridge, MA., 1967, pp. 414–437.

D. Leivant. Predicative recurrence in finite type. In A. Nerode and Y. Matiyasevich, editors, *Logical Foundations of Computer Science*, volume 813 of *LNCS*, pages 227–239, 1994.

P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North–Holland, Amsterdam, 1971.

G. Ostrin and S. S. Wainer. Elementary arithmetic. *Annals of Pure and Applied Logic*, 133:275–292, 2005.

K. Schütte. *Proof Theory*. Springer Verlag, Berlin, Heidelberg, New York, 1977.

H. Schwichtenberg. An arithmetic for polynomial-time computation. *Theoretical Computer Science*, 357:202–214, 2006.

H. Schwichtenberg and S. Bellantoni. Feasible computation with higher types. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, Proceedings NATO Advanced Study Institute, Marktoberdorf, 2001, pages 399–415. Kluwer Academic Publisher, 2002.

H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.

A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer Verlag, Berlin, Heidelberg, New York, 1973.

Index