Proof Theory

A selection of papers
from the Leeds Proof Theory Programme 1990

edited by

Peter Aczel
University of Manchester

Harold Simmons
University of Manchester

and

Stanley S. Wainer
University of Leeds

CAMBRIDGE
UNIVERSITY PRESS

CONTENTS

Proofs as programs

H. SCHWICHTENBERG

Proofs as Programs

HELMUT SCHWICHTENBERG

Mathematisches Institut, Universität München

Suppose a formal proof of $\forall x \exists y \operatorname{Spec}(x,y)$ is given, where $\operatorname{Spec}(x,y)$ is an atomic formula expressing some specification for natural numbers x, y. For any particular number n we then obtain a formal proof of $\exists y \operatorname{Spec}(n,y)$. Now the proof–theoretic normalization procedure yields another proof of $\exists y \operatorname{Spec}(n,y)$ which is in normal form. In particular, it does not use induction axioms any more, and it also does not contain non–evaluated terms. Hence we can read off, linearly in the size of the normal proof, an instance m for y such that $\operatorname{Spec}(n,m)$ holds. In this way a formal proof can be seen as a program, and the central part in implementing this programming language consists in an implementation of the proof–theoretic normalization procedure.

There are many ways to implement normalization. As usual, a crucial point is a good choice of the data structures. One possibility is to represent a term as a function (i.e. a SCHEME–procedure) of its free variables, and similarly to represent a derivation (in a Gentzen–style system of natural deduction) as a function of its free assumption and object variables. Then substitution is realized as application, and normalization is realized as the built–in evaluation process of SCHEME (or any other language of the LISP–family). We presently experiment with an implementation along these lines, and the results up to now are rather promising. Some details are given in an appendix.

It is not the prime purpose of the present paper to discuss this implementation. Rather, we want to explore the theoretical possibilities and limitations of a programming language based on formal proofs. The notion of proof is taken here in a quite basic sense: the formal language is supposed to talk about algebraic data structures (i.e. free algebras), and structural recursion as well as structural induction is allowed. Hence we discuss systems of the strength of ordinary arithmetic. We will measure the strength of our proofs/programs in terms of the so–called slow growing hierarchy G_α introduced by Wainer and studied by Girard. We will give a new proof of the following result of Kreisel and (Girard 1981): Any function defined by a proof of $\forall x \exists y \mathsf{Spec}(x,y)$ is bounded by a function G_α of the hierarchy with α below the Bachmann–Howard ordinal, and conversely that for any such G_α there is an atomic formula $\mathsf{Spec}_\alpha(x,y)$ such that $G_\alpha(n) \leq$ the least m with $\mathsf{Spec}_\alpha(n,m)$, and $\forall x \exists y \mathsf{Spec}_\alpha(x,y)$ is provable, and hence for any proof of this fact the function computed by that proof (considered as a program) grows at least as fast as G_α.

On the more technical side, our work builds heavily on earlier work of (Buchholz 1987) and (Arai 1989). In particular, the material in Sections 1–3 on trees, tree notations and the slow growing hierarchy is taken from Buchholz. Also, the ω^+–Rule below is derived from (a special case of) the Ω–Rule in (Buchholz 1987) (or more precisely of its "slow-growing" variant in (Arai 1989)), which in turn is based on earlier work of (Howard 1972). The new twist here is that we make use of a technique of (Howard 1980) to measure the complexity of a (finite) term/proof by (transfinite) trees; for this to go through it is essential to use a natural deduction system and not a Tait calculus as in (Buchholz 1987) or (Buchholz and Wainer 1987).

More precisely, we inductively define what it means for a (finite) term/proof involving recursion/induction constants to be SDH–generated (for Sanchis–Diller–Howard) with measure α (a transfinite tree) and rank m. One clause of this inductive definition is called ω^+–rule and introduces uncountable trees. In this setup it is easy to provide relatively perspicious and complete proofs of the relations mentioned above between the slow growing hierarchy and the functions computed by proofs/programs in arithmetical systems.

1. TREES

We give an informal treatment of the *tree classes* $\mathcal{T}_\sigma$ with $\sigma \leq \nu$, for some fixed $\nu < \omega$. For our later applications it will suffice to take $\nu = 2$. The material developed here will later (in Section 2) give rise to a system of (finitary, or algebraic) notations for such trees.

Let $\sigma < \omega$, and assume that $\mathcal{T}_\varrho$ for all $\varrho < \sigma$ is defined already. We then define the *tree class* $\mathcal{T}_\sigma$ inductively by the clause

$\mathcal{T}_\sigma$. If $\alpha : I \to \mathcal{T}_\sigma$ is a function with $I = \emptyset, \{0\}$ or $\mathcal{T}_\varrho$ for some $\varrho < \sigma$, then $\alpha \in \mathcal{T}_\sigma$.

If $I = \emptyset$, then $\alpha : I \to \mathcal{T}_\sigma$ is denoted by 0. If $I = \{0\}$, then $\alpha : I \to \mathcal{T}_\sigma$ is determined by $\alpha(0) =: \beta$ and denoted by β^+. If $I = \mathcal{T}_\varrho$, then $\alpha : I \to \mathcal{T}_\sigma$ is denoted by $(\alpha_\zeta)_{\zeta \in \mathcal{T}_\varrho}$ with $\alpha_\zeta := \alpha(\zeta)$.
$\mathcal{T}_0$ consists of $0, 0^+, 0^{++}, \ldots$ and hence is identified with the set $\mathbf{N}$ of natural

numbers. $\mathcal{T}_1$ is the set of countable trees. For example,

$$\omega := (n)_{n\in\mathbf{N}} \in \mathcal{T}_1$$

and more generally

$$\Omega_\sigma := (\zeta)_{\zeta\in\mathcal{T}_\sigma} \in \mathcal{T}_{\sigma+1},$$

hence $\omega = \Omega_0$.

Note that, since $\mathcal{T}_\sigma$ is defined inductively, the following principle of transfinite (or Noetherian) induction on $\mathcal{T}_\sigma$ holds.

$$(\forall\alpha \in \mathcal{T}_\sigma.\forall\zeta \in \mathrm{dom}(\alpha) : \varphi(\alpha_\zeta) \rightarrow \varphi(\alpha)) \rightarrow \forall\alpha \in \mathcal{T}_\sigma : \varphi(\alpha).$$

Here φ is an arbitrary property of elements of $\mathcal{T}_\sigma$.

Addition, multiplication and exponentiation of trees are defined as follows.

i. $\alpha + 0 = \alpha$

ii. $\alpha + \beta^+ = (\alpha + \beta)^+$

iii. $\alpha + (\beta_\zeta)_{\zeta\in\mathcal{T}_\sigma} = (\alpha + \beta_\zeta)_{\zeta\in\mathcal{T}_\sigma}$.

Then $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$; this can be proved easily by induction on γ.

i. $\alpha \cdot 0 = 0$.

ii. $\alpha \cdot (\beta + 1) = (\alpha \cdot \beta) + \alpha$.

iii. $\alpha \cdot (\beta_\zeta)_{\zeta\in\mathcal{T}_\sigma} = (\alpha \cdot \beta_\zeta)_{\zeta\in\mathcal{T}_\sigma}$.

Then $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$, and also $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$.

i. $\alpha^0 = 1$.

ii. $\alpha^{\beta+1} = \alpha^\beta \cdot \alpha$.

iii. $\alpha^{(\beta_\zeta)_{\zeta\in\mathcal{T}_\sigma}} = (\alpha^{\beta_\zeta})_{\zeta\in\mathcal{T}_\sigma}$.

Then $\alpha^{(\beta+\gamma)} = \alpha^\beta \cdot \alpha^\gamma$, and also $(\alpha^\beta)^\gamma = \alpha^{\beta\cdot\gamma}$.

Using these arithmetical operations we can now give some more examples of trees.

$$\omega \cdot 2 = \omega + \omega = (\omega + n)_{n\in\mathbf{N}},$$

$$\omega^2 = \omega \cdot \omega = (\omega \cdot n)_{n\in\mathbf{N}},$$

$$\omega^\omega = (\omega^n)_{n\in\mathbf{N}}$$

and similarly, for $\Omega := \Omega_1$,

$$\Omega \cdot 2 = \Omega + \Omega = (\Omega + \zeta)_{\zeta\in\mathcal{T}_1},$$

$$\Omega^2 = \Omega \cdot \Omega = (\Omega \cdot \zeta)_{\zeta \in \mathcal{T}_1},$$

$$\Omega^\Omega = (\Omega^\zeta)_{\zeta \in \mathcal{T}_1}.$$

From now on we restrict attention to the tree class $\mathcal{T}_\nu$ for some fixed $\nu < \omega$. We define *collapsing functions* $\mathcal{D}_\sigma : \mathcal{T}_\nu \to \mathcal{T}_{\sigma+1}$ for all $\sigma < \nu$, by induction on $\mathcal{T}_\nu$.

i. $\mathcal{D}_\sigma 0 := \Omega_\sigma$

ii. $\mathcal{D}_\sigma(\alpha + 1) := (\mathcal{D}_\sigma(\alpha) \cdot (n+1))_{n \in \mathrm{N}}$

iii. If $\varrho \leq \sigma$, then $\mathcal{D}_\sigma((\alpha_\zeta)_{\zeta \in \mathcal{T}_\varrho}) := (\mathcal{D}_\sigma \alpha_\zeta)_{\zeta \in \mathcal{T}_\varrho}$.

iv. If $\sigma < \mu + 1$, then $\mathcal{D}_\sigma((\alpha_\zeta)_{\zeta \in \mathcal{T}_{\mu+1}}) := (\mathcal{D}_\sigma \alpha_{\zeta_n})_{n \in \mathrm{N}}$ where $\zeta_0 := \Omega_\mu$, $\zeta_{n+1} := \mathcal{D}_\mu \alpha_{\zeta_n}$.

For $\alpha \in \mathcal{T}_{\sigma+1}$, $\mathcal{D}_\sigma \alpha$ is similar to $\Omega_\sigma \cdot \omega^\alpha$; the only difference is that in ii we have $n+1$ instead of n (for technical reasons; cf. the proof of the Cut Elimination Lemma in Section 4). However, the crucial clause in the definition of $\mathcal{D}_\sigma$ is iv, which makes $\mathcal{D}_\sigma$ defined for trees beyond $\mathcal{T}_{\sigma+1}$, and hence makes it a collapsing function.

For example, $\mathcal{D}_0\Omega_1 = (\mathcal{D}_0\zeta_n)_{n \in \mathcal{T}_0}$ where $\zeta_0 = \omega$, $\zeta_{n+1} = \mathcal{D}_0\zeta_n$. Since $\mathcal{D}_0\alpha$ for $\alpha \in \mathcal{T}_1$ is similar to the exponential ω^α, we can conclude that $\mathcal{D}_0\Omega_1$ is similar to the tree $(1, \omega, \omega^\omega, \omega^{\omega^\omega}, \ldots)$, which is usually denoted by ε_0.

From now on we restrict attention to trees built up from $0, 1$ by $+$ and $\mathcal{D}_\sigma$, for $\sigma < \nu$. Such trees can clearly be denoted by elements of an appropriate free algebra, hence by finitary notations.

2. TREE NOTATIONS

Let D_σ for $\sigma < \nu$ be unary function symbols; again $\nu < \omega$ is a fixed number (and $\nu = 2$ in Sections 4 and 5). We define a set T of terms and simultaneously a set HT of principal terms inductively by

i. $1 \in \mathrm{HT}$.

ii. If $a \in \mathrm{T}$ and $\sigma < \nu$, then $D_\sigma a \in \mathrm{HT}$.

iii. If $a_1, \ldots, a_k \in \mathrm{HT}$ with $k \geq 0$, then $(a_1, \ldots, a_k) \in \mathrm{T}$.

The empty list $()$ is denoted by 0. For the one element list (a) we often write a; in this sense we have $\mathrm{HT} \subset \mathrm{T}$.

The elements of T are called *tree notations*, since for any $a \in \mathrm{T}$ we can define its *value* $\mathrm{val}(a) \in \mathcal{T}_\nu$ by

i. $\mathrm{val}(1) = 1$,

ii. $\mathrm{val}(D_\sigma a) = \mathcal{D}_\sigma(\mathrm{val}(a))$,

iii. $\mathrm{val}(a_1,\dots,a_k) = \mathrm{val}(a_1) + \dots + \mathrm{val}(a_k)$.

For $a, b \in \mathrm{T}$ we define $a+b$ to be the concatenation of the lists a and b. Then clearly $a+b \in \mathrm{T}$, and also $a+0 = 0+a = a$, $a+(b+c) = (a+b)+c$. We abbreviate $a + \dots + a$ by $a \cdot n$.

The subsets $\mathrm{T}_\sigma \subseteq \mathrm{T}$ and $\mathrm{HT}_\sigma \subseteq \mathrm{HT}$ are to consist of those terms containing D_μ with $\sigma \leq \mu$ only in a context $D_\rho a$ with $\rho < \sigma$. More precisely, for any $\sigma < \nu$ the sets T_σ and HT_σ are defined by

i. $1 \in \mathrm{HT}_\sigma$.

ii. If $a \in \mathrm{T}$ and $\rho < \sigma$, then $D_\rho a \in \mathrm{HT}_\sigma$.

iii. If $a_1, \dots, a_k \in \mathrm{HT}_\sigma$ with $k \geq 0$, then $(a_1, \dots, a_k) \in \mathrm{T}_\sigma$.

Clearly $\mathrm{val}(a) \in \mathcal{T}_\sigma$ for $a \in \mathrm{T}_\sigma$.

T_0 consists of $0, 1, 1+1, 1+1+1, \dots$ and hence is identified with the set $\mathbf{N}$ of natural numbers. Let $\omega := D_0 0, \Omega := D_1 0$ and generally $\Omega_\sigma := D_\sigma 0$. The Ω_σ's as well as 1 are called *regular* tree notations.

For any $a \in \mathrm{T}$ we have $\mathrm{val}(a) \in \mathcal{T}_\nu$, i.e. $\mathrm{val}(a)\colon I \to \mathcal{T}_\nu$ with $I = \emptyset, \{0\}$ or $\mathcal{T}_\sigma$ for some $\sigma < \nu$. We now want to recover from a its *type* $\tau(a)$, which is to be $0, 1$ or Ω_σ if I is $\emptyset, \{0\}$ or $\mathcal{T}_\sigma$, respectively. In the case $\tau(a) = \Omega_\sigma$, we also want to recover from a its *fundamental sequence*, i.e. notations $a[z] \in \mathrm{T}$ with value $\mathrm{val}(a)(\mathrm{val}(z))$, for all $z \in \mathrm{T}_\sigma$.

Let $|0| := \emptyset, |1| := \{0\}$ and $|\Omega_\sigma| := \mathrm{T}_\sigma$. For $a \in \mathrm{T}$ we define $\tau(a) \in \{0,1\} \cup \{\Omega_\sigma : \sigma < \nu\}$ and $a[z] \in \mathrm{T}$ for $z \in |\tau(a)|$ by induction on a, as follows.

i. For $a \in \{0,1\} \cup \{\Omega_\sigma : \sigma < \nu\}$ let $\tau(a) := a$ and $a[z] := z$.

ii. For $D_\sigma a$ with $\tau(a) = 1$ let $\tau(D_\sigma a) := \omega$ and $(D_\sigma a)[n] = (D_\sigma a[0]) \cdot (n+1)$.

iii. For $D_\sigma a$ with $\tau(a) = \Omega_\varrho$ with $\varrho \leq \sigma$ let $\tau(D_\sigma a) := \Omega_\varrho$ and $(D_\sigma a)[z] := D_\sigma a[z]$.

iv. For $D_\sigma a$ with $\tau(a) = \Omega_{\mu+1}$ with $\sigma < \mu+1$ let $\tau(D_\sigma a) := \omega$ and $(D_\sigma a)[n] := D_\sigma a[z_n]$ with $z_0 := \Omega_\mu$, $z_{n+1} := D_\mu a[z_n]$.

v. $\tau(a_1, \dots, a_k) := \tau(a_k)$ and $(a_1, \dots, a_k)[z] := (a_1, \dots, a_{k-1}, a_k[z])$.

Then clearly $\tau(a) = 0 \iff a = 0$ and $\tau(a) = 1 \iff a = a[0] + 1$. Also, if $a \in \mathrm{T}_\sigma$ and $\tau(a) \neq 0, 1$, then $\tau(a) = \Omega_\varrho$ for some $\varrho < \sigma$, and in this case we have $a[z] \in \mathrm{T}_\sigma$ for all $z \in \mathrm{T}_\varrho = |\tau(a)|$.

Lemma 2.1. *If* $a \in \mathrm{T}$ *and* $z \in |\tau(a)|$, *then* $\mathrm{val}(z) \in \mathrm{dom}(\mathrm{val}(a))$ *and* $\mathrm{val}(a[z]) = \mathrm{val}(a)(\mathrm{val}(z))$.

Proof. First note that if $\tau(a) = 0, 1$ or Ω_σ, then $\mathrm{dom}(\mathrm{val}(a)) = \emptyset, \{0\}$ or $\mathcal{T}_\sigma$. We prove the Lemma by induction on a, and treat only Case iv, i.e. $D_\sigma a$ with

$\tau(a) = \Omega_{\mu+1}$ and $\sigma < \mu+1$. Let $\mathrm{val}(a) = (\alpha_\zeta)_{\zeta \in \mathcal{T}_{\mu+1}}$. Then by induction hypothesis

$$\mathrm{val}((D_\sigma a)[n]) = \mathrm{val}(D_\sigma a[z_n]) = \mathcal{D}_\sigma(\mathrm{val}(a[z_n])) = \mathcal{D}_\sigma \alpha_{\mathrm{val}(z_n)}$$

with $z_0 = \Omega_\mu, z_{n+1} = D_\mu a[z_n]$, and

$$\mathrm{val}(D_\sigma a)(n) = \mathcal{D}_\sigma(\mathrm{val}(a))(n) = \mathcal{D}_\sigma \alpha_{\zeta_n}$$

with $\zeta_0 = \Omega_\mu, \zeta_{n+1} = \mathcal{D}_\mu \alpha_{\zeta_n}$. Hence it suffices to prove $\mathrm{val}(z_n) = \zeta_n$. This follows by induction on n from $\mathrm{val}(\Omega_\mu) = \Omega_\mu$ and

$$\mathrm{val}(z_{n+1}) = \mathcal{D}_\mu \mathrm{val}(a[z_n]) = \mathcal{D}_\mu \alpha_{\mathrm{val}(z_n)} = \mathcal{D}_\mu \alpha_{\zeta_n} = \zeta_{n+1}. \square$$

As a consequence, we can infer the principle of *transfinite induction on* T_σ, i.e.

$$(\forall a \in \mathrm{T}_\sigma . \forall z \in |\tau(a)| : \varphi(a[z]) \to \varphi(a)) \to \forall a \in \mathrm{T}_\sigma : \varphi(a),$$

from the principle of transfinite induction on $\mathcal{T}_\sigma$ in Section 1. To see this, assume the premise and let $a \in \mathrm{T}_\sigma$. We use transfinite induction on $\mathrm{val}(a) \in \mathcal{T}_\sigma$. It suffices to prove $\forall z \in |\tau(a)| : \varphi(a[z])$. So let $z \in |\tau(a)|$. By Lemma 2.1 $\mathrm{val}(z) \in \mathrm{dom}(\mathrm{val}(a))$ and $\mathrm{val}(a[z]) = \mathrm{val}(a)(\mathrm{val}(z))$. Hence $\mathrm{val}(a[z])$ comes before $\mathrm{val}(a)$ in the sense of the inductive generation of $\mathcal{T}_\sigma$. So $\varphi(a[z])$ by induction hypothesis.

3. THE SLOW GROWING HIERARCHY

Given a tree notation $a \in \mathrm{T}_1$ and a natural number n, we may decide to climb down the tree (which grows downwards), using n as a parameter. This is done as follows. If the node we are at is formed by the successor operation, then we have to do some work to climb down one step. If on the other hand the node is formed as a sequence (which must be of length ω, since $a \in \mathrm{T}_1$), then we don't have to work but just slip down to the n–th element of the sequence.

If we count the pieces of work we have done until we reach a bottom node, we get a natural number $G_a(n)$. These functions $G_a\colon \mathrm{N} \to \mathrm{N}$ for $a \in \mathrm{T}_1$ form the so–called *slow growing hierarchy*; the formal definition is by transfinite induction on $a \in \mathrm{T}_1$, as follows

i. $G_0(n) = 0$,

ii. $G_{a+1}(n) = G_a(n) + 1$,

iii. $G_a(n) = G_{a[n]}(n)$ if $\tau(a) = \omega$.

Note that $G_{a+b}(n) = G_a(n) + G_b(n)$; this can be proved easily by transfinite induction on $b \in \mathrm{T}_1$.

For example,

$$\begin{aligned}
G_k(n) &= k,\\
G_\omega(n) &= G_{\omega[n]}(n) = G_n(n) = n,\\
G_{D_01}(n) &= G_{\omega\cdot(n+1)}(n) = (n+1)\cdot G_\omega(n) = (n+1)\cdot n,\\
G_{D_02}(n) &= G_{(D_01)\cdot(n+1)}(n) = (n+1)\cdot G_{D_01}(n) = (n+1)^2\cdot n,\\
G_{D_0\omega}(n) &= G_{D_0n}(n) = (n+1)^n\cdot n.
\end{aligned}$$

Hence, the functions G_a with a built up from $0, 1$ by $+$ and D_0 but without nesting of D_0 are all polynomials.

The functions G_a with a of the form $D_0D_1^m0$ will be used in Section 4 to estimate the instances provided by existential proofs in arithmetic. We will also show in Section 5 that this result is best possible, since any such G_a is bounded by a function provably total in arithmetic.

In order to achieve these results we need some monotonicity properties of the G_a. Since $G_kn = k$ and $G_\omega n = n$, we cannot have that $\mathrm{val}(a) < \mathrm{val}(b)$ implies $G_an \le G_bn$, for all n. Hence we introduce appropriate relations $<_k$ such that $a <_k b$ implies $G_an \le G_bn$ for all $n \ge k$.

For $a \ne 0$ let $a^- := a[0]$ if $\tau(a) = 1$ or $\tau(a) = \omega$, and $a^- := a[\Omega_\mu]$ if $\tau(a) = \Omega_{\mu+1}$. Let $a <_k b$ iff we have a finite sequence $a = a_0, a_1, \ldots, a_n = b$ with $n > 0$ such that for all $i < n$ either $a_i = a_{i+1}^-$ or $\tau(a_{i+1}) = \omega$ and $a_i = a_{i+1}[j]$ for some j with $1 \le j \le k$. Note that from $a <_k b$ and $k \le l$ we can obviously conclude that $a <_l b$. We write $a \le_k b$ for $a <_k b$ or $a = b$.

Some of our later arguments will be by induction on $\mathrm{length}(a)$, which is defined by

$$\begin{aligned}
\mathrm{length}(0) &= 0,\\
\mathrm{length}(1) &= 1,\\
\mathrm{length}(D_\sigma a) &= \mathrm{length}(a) + \sigma + 1,\\
\mathrm{length}(a_1, \ldots, a_k) &= \mathrm{length}(a_1) + \cdots + \mathrm{length}(a_k).
\end{aligned}$$

Note that $\mathrm{length}(a + b) = \mathrm{length}(a) + \mathrm{length}(b)$.

Lemma 3.1.

i. *If* $a \ne 0$, *then* $(D_\sigma a)^- = D_\sigma a^-$.

ii. *If* $b \ne 0$, *then* $(a + b)^- = a + b^-$.

iii. *If* $a \ne 0$, *then* $\mathrm{length}(a^-) < \mathrm{length}(a)$.

Proof. i. If $\tau(a) = 1$ or ω, then $\tau(D_\sigma a) = \omega$ and

$$(D_\sigma a)^- = (D_\sigma a)[0] = (D_\sigma a[0]) \cdot 1 = D_\sigma a^-.$$

If $\tau(a) = \Omega_{\mu+1}$ with $\mu + 1 \leq \sigma$, then $\tau(D_\sigma a) = \Omega_{\mu+1}$ and

$$(D_\sigma a)^- = (D_\sigma a)[\Omega_\mu] = D_\sigma a[\Omega_\mu] = D_\sigma a^- .$$

If $\tau(a) = \Omega_{\mu+1}$ with $\sigma < \mu + 1$, then $\tau(D_\sigma a) = \omega$ and

$$(D_\sigma a)^- = (D_\sigma a)[0] = D_\sigma a[\Omega_\mu] = D_\sigma a^- .$$

ii. The claim follows from $\tau(a+b) = \tau(b)$ and $(a+b)[z] = a + b[z]$.
iii. By induction on a. Case $1, \omega$. Then $a^- = a[0] = 0$ and the claim is immediate. Case $D_\sigma a$. For $a = 0$ this is clear. For $a \neq 0$ we have by i

$$\begin{aligned} \mathsf{length}((D_\sigma a)^-) &= \mathsf{length}(D_\sigma a^-) \\ &= \mathsf{length}(a^-) + \sigma + 1 \\ &< \mathsf{length}(a) + \sigma + 1 \\ &= \mathsf{length}(D_\sigma a). \end{aligned}$$

Case $a + b$. Similarly, using ii. □

Lemma 3.2.

i. *If* $b_0 <_k b$, *then* $a + b_0 <_k a + b$.
ii. *If* $c \neq 0$, *then* $1 \leq_1 c$.
iii. *If* $b <_k a$, *then* $D_\sigma b <_k D_\sigma a$.
iv. $(D_\sigma^m a) + 1 \leq_1 D_\sigma^m(a+1)$.

Proof. i. The claim follows from Lemma 3.1 ii together with $\tau(a+b) = \tau(b)$ and $(a+b)[z] = a + b[z]$.
ii. By induction on $\mathsf{length}(c)$. Case 1. $1 \leq_1 1$. Case ω. $1 = \omega[1] <_1 \omega$. Case $\Omega_{\mu+1}$. $1 \leq_1 \Omega_\mu <_1 \Omega_{\mu+1}$; here $1 \leq \Omega_\mu$ holds by induction hypothesis. Case $D_\sigma a$ with $a \neq 0$. $1 \leq_1 D_\sigma a^- = (D_\sigma a)^- <_1 D_\sigma a$; here the first inequality follows by induction hypothesis, since $\mathsf{length}(a^-) < \mathsf{length}(a)$. Case $a + b$ with $a, b \neq 0$. $1 \leq_1 a <_1 a + 1 \leq_1 a + b$; in the last inequality we have used i and the induction hypothesis.
iii. This follows from $(D_\sigma a)^- = D_\sigma a^-$ and the fact that, if $\tau(a) = \omega$, then $\tau(D_\sigma a) = \omega$ and $(D_\sigma a)[n] = D_\sigma a[n]$.
iv. By induction on m. For 0 there is nothing to show, and in the induction step we have $(D_\sigma^{m+1} a) + 1 \leq_1 (D_\sigma^{m+1} a) \cdot 2 = (D_\sigma((D_\sigma^m a) + 1))[1] <_1 D_\sigma((D_\sigma^m a) + 1) \leq_1 D_\sigma^{m+1}(a+1)$, where in the first inequality we have used ii, and in the last one we have used the induction hypothesis and iii. □

Lemma 3.3.

i. *If* $\tau(c) = \Omega_{\mu+1}$ *and* $x, y \in |\tau(c)|$ *and* $x <_k y$, *then* $c[x] <_k c[y]$.
ii. *If* $\tau(c) = \Omega_{\mu+1}$ *and* $x \in |\tau(c)|$, *then* $c[x] + 1 \leq_1 c[x+1]$.

iii. If $\tau(c) = \omega$, then $c[n] + 1 \leq_1 c[n+1]$.

Proof. i. By induction on $\mathrm{length}(c)$. Case $\Omega_{\mu+1}$. Obvious, since $\Omega_{\mu+1}[z] = z$. Case $D_\sigma a$. Then $\tau(a) = \Omega_{\mu+1}$ and $\mu + 1 \leq \sigma$, hence

$$(D_\sigma a)[x] = D_\sigma a[x] <_k D_\sigma a[y] = (D_\sigma a)[y],$$

by induction hypothesis and Lemma 3.2 iii. Case $a+b$. Then $\tau(b) = \Omega_{\mu+1}$ and

$$(a+b)[x] = a + b[x] <_k a + b[y] = (a+b)[y].$$

ii. By induction on $\mathrm{length}(c)$. Case $\Omega_{\mu+1}$. Obvious, since $\Omega_{\mu+1}[z] = z$. Case $D_\sigma a$. Then $\tau(a) = \Omega_{\mu+1}$ and $\mu + 1 \leq \sigma$, hence

$$(D_\sigma a)[x] + 1 = D_\sigma a[x] + 1 \leq_1 D_\sigma(a[x] + 1) \leq_1 D_\sigma a[x+1] = (D_\sigma a)[x+1]$$

where in the first inequality we have used Lemma 3.2 iv, and in the second one the induction hypothesis and Lemma 3.2 iii. Case $a+b$. Then $\tau(b) = \Omega_{\mu+1}$ and

$$(a+b)[x] + 1 = a + b[x] + 1 \leq_1 a + b[x+1] = (a+b)[x+1].$$

iii. By induction on $\mathrm{length}(c)$. Case ω. Then $\omega[n] + 1 = n + 1 = \omega[n+1]$. Case $D_\sigma a$ with $\tau(a) = 1$. Then

$$(D_\sigma a)[n] + 1 = (D_\sigma a[0]) \cdot (n+1) + 1 \leq_1 (D_\sigma a[0]) \cdot (n+2) = (D_\sigma a)[n+1].$$

Case $D_\sigma a$ with $\tau(a) = \omega$. Then

$$(D_\sigma a)[n] + 1 = (D_\sigma a[n]) + 1 \leq_1 D_\sigma(a[n] + 1) \leq_1 D_\sigma a[n+1] = (D_\sigma a)[n+1].$$

Case $D_\sigma a$ with $\tau(a) = \Omega_{\mu+1}, \sigma < \mu + 1$. Then $(D_\sigma a)[n] = D_\sigma a[z_n]$ with $z_0 = \Omega_\mu$, $z_{n+1} = D_\mu a[z_n]$. It suffices to prove

$$z_n + 1 \leq_1 z_{n+1}, \tag{1}$$

for then we obtain

$$(D_\sigma a[z_n]) + 1 \leq_1 D_\sigma(a[z_n] + 1) \leq_1 D_\sigma a[z_n + 1] \leq_1 D_\sigma a[z_{n+1}],$$

using ii, (1) and i. We prove (1) by induction on n. The base case follows from

$$\Omega_\mu + 1 \leq_1 (D_\mu 0) \cdot 2 = (D_\mu 1)[1] <_1 D_\mu 1 \leq_1 D_\mu a[\Omega_\mu],$$

and the induction step follows from

$$(D_\mu a[z_n]) + 1 \leq_1 D_\mu(a[z_n] + 1) \leq_1 D_\mu a[z_n + 1] \leq_1 D_\mu a[z_{n+1}],$$

where we have used ii, the induction hypothesis and i. Case $a+b$ with $\tau(b)=\omega$. Then

$$(a+b)[n]+1 = a+b[n]+1 \leq_1 a+b[n+1] = (a+b)[n+1]. \square$$

Now we can prove the monotonicity properties of the functions G_a we were looking for.

Lemma 3.4. (Monotonicity Properties of the G_a)

i. If $b <_k a$ and $k \leq n$, then $G_b(n) \leq G_a(n)$

ii. $G_a(n) \leq G_a(n+1)$.

Proof. i. By transfinite induction on $a \in \mathrm{T}_1$. Case a^-. If $\tau(a)=1$, we have

$$G_{a^-}(n) < G_{a^-}(n)+1 = G_a(n).$$

If $\tau(a)=\omega$, we have $a^- = a[0]$, and by Lemma 3.3 iii we know $a[0] \leq_1 a[n]$. Hence by induction hypothesis

$$G_{a^-}(n) = G_{a[0]}(n) \leq G_{a[n]}(n) = G_a(n).$$

Case $a[j]$ with $1 \leq j \leq k$. Then again by Lemma 3.3 iii we have $a[j] \leq_1 a[n]$, hence by induction hypothesis

$$G_{a[j]}(n) \leq G_{a[n]}(n) = G_a(n).$$

ii. By transfinite induction on $a \in \mathrm{T}_1$. Case 0. $G_0(n) = 0 = G_0(n+1)$. Case $a+1$.

$$G_{a+1}(n) = G_a(n)+1 \leq G_a(n+1)+1 = G_{a+1}(n+1),$$

by induction hypothesis. Case $\tau(a)=\omega$.

$$G_a(n) = G_{a[n]}(n) \leq G_{a[n]}(n+1) \leq G_{a[n+1]}(n+1) = G_a(n+1)$$

by induction hypothesis and Lemma 3.3 iii together with i. □

We finally prove a Lemma on the functions G_a which enables us to shift a depence on n from the index into the argument. This will be used in Section 4.

Lemma 3.5. *Let $a = D_0(c\cdot(n+1))$ with $c = D_\sigma^m(\Omega_\sigma \cdot m)$ and $1 \leq m \leq n$, and furthermore $d = D_0 D_\sigma^{m+2} 0$. Then we have $G_a(1) \leq G_d(n)$.*

Proof. First note that

$$\begin{aligned} c\cdot(n+1) &= (D_\sigma^m(\Omega_\sigma \cdot m))\cdot(n+1) \\ &= (D_\sigma(D_\sigma^{m-1}(\Omega_\sigma \cdot m)+1))[n] \\ &\leq_n D_\sigma(D_\sigma^{m-1}(\Omega_\sigma \cdot m)+1) \\ &\leq_1 D_\sigma^m(\Omega_\sigma \cdot (m+1)) \\ &= D_\sigma^m((D_\sigma 1)[m]) \\ &\leq_m D_\sigma^{m+1} 1 \end{aligned}$$

Hence we obtain, using Lemma 3.4 i and ii

$$G_{D_0(c\cdot(n+1))}(1) \le G_{D_0(c\cdot(n+1))}(n) \le G_{D_0 D_\sigma^{m+2} 0}(n). \square$$

4. AN ESTIMATE OF INSTANCES IN EXISTENTIAL PROOFS

We now set up a formal system of terms involving recursion operators and on top of it a formal system of derivations involving induction axioms. It is possible and convenient to treat both simultaneously; we use r, s, t to denote terms as well as derivations.

For any term/derivation r we define inductively what it means for r to be SDH–generated with size a (a tree notation) and rank m; we write $\vdash_m^a r$ for this. Note that r is a finite term/derivation here, i.e. it is not expanded into an infinite object using some kind of ω–rule. The transfinite analysis of r comes in at the level of transfinite SDH generation trees for such r, whose size is measured by our notations for (transfinite) trees treated in Section 2. The inductive definition of $\vdash_m^a r$ involves an ω^+–Rule, which is used for an appropriate analysis of terms/derivations containing recursion/induction.

We start out with the easy observation that for any term/derivation r we can find an SHD generation tree of size Ωk and rank m, for some k and m reflecting the levels of recursion/induction operators in r. Hence we get $\vdash_m^{\Omega k} r$. Then we use a Cut Elimination Lemma to bring the rank m down to zero, at the expense of rising the size Ωk to $D_1^m(\Omega k)$; so we get

$$\vdash_0^{D_1^m(\Omega k)} r.$$

Now we can apply a First Collapsing Lemma, which says that if $\vdash_0^a r$ with r closed, then $\vdash_0^{D_0(a)} |r|$, where $|r|$ is the numeral denoting

- the value of r in case r is a term, or
- the value of some correct instance provided by r in case r is a closed derivation of an existential formula $\exists y \, \mathrm{Spec}(n, y)$ with an atomic formula Spec.

Hence we get

$$\vdash_0^{D_0 D_1^m(\Omega k)} |r|.$$

But SDH generation trees of rank 0 for numerals can be easily analysed: From $\vdash_0^a n$ with $a \in T_1$ we first get $\vdash_0^{G_a(1)} n$ by the Second Collapsing Lemma and then $n < G_a(1)$ by the Value Lemma. So altogether we have

$$|r| < G_{D_0 D_1^m(\Omega k)}(1),$$

and by Lemma 3.5 this essentially suffices for our desired estimate.

We now carry out this program. First we have to say exactly what we mean by a term and by a formal proof. Our definition is guided by the following considerations. It should be possible to

- view a formal proof as a λ–term (i.e. a SCHEME procedure) which has the derived formula as its type, such that normalization will correspond to evaluation, and to
- carry out an ordinal (or better tree) analysis of formal proofs by the SDH–technique.

Since the SDH–technique also refers to λ–terms, it seems appropriate to use the $\rightarrow\forall$–fragment of Gentzens natural deduction calculus, for then the logical rules are just introduction and elimination rules for $\rightarrow$ and $\forall$, which correspond exactly to λ–abstraction and application.

The first thing to note is that we don't loose anything by this restriction to the $\rightarrow\forall$–fragment, and in particular don't need any special axioms to recover classical arithmetic. To see this, we first show that for any atomic formula $A(\vec{x})$ we can derive its stability $\forall\vec{x}.\neg\neg A(\vec{x}) \rightarrow A(\vec{x})$. This is done as follows. Atomic formulas are taken as terms $\mathsf{atom}(r)$ of type prop, with r a term of type boole and atom a constant of type $\mathsf{boole} \rightarrow \mathsf{prop}$. In particular, falsity $\perp$ is defined as $\mathsf{atom}(\mathtt{ff})$ with $\mathtt{ff}$ a constant of type boole; then $\neg\varphi$ is defined to be $\varphi \rightarrow \perp$. Using boolean induction $\varphi(\mathtt{tt}) \rightarrow \varphi(\mathtt{ff}) \rightarrow \forall p \varphi(p)$, first prove

$$\forall p, q.\, \mathsf{atom}(p \supset q) \leftrightarrow (\mathsf{atom}(p) \rightarrow \mathsf{atom}(q))$$

where $\supset$ is a constant of type $\mathsf{boole} \rightarrow \mathsf{boole} \rightarrow \mathsf{boole}$ corresponding to implication; here we need the truth axiom $\mathsf{atom}(\mathtt{tt})$. Using this it is easy to prove

$$\forall p.\neg\neg\, \mathsf{atom}(p) \rightarrow \mathsf{atom}(p),$$

again by boolean induction.

Now we can substitute a boolean term $r(\vec{x})$ for p, and with $A(\vec{x}) :\equiv \mathsf{atom}(r(\vec{x}))$ we obtain $\forall\vec{x}.\neg\neg A(\vec{x}) \rightarrow A(\vec{x})$.

Using induction on $\rightarrow\forall$–formulas it is easy to derive $\neg\neg\varphi \rightarrow \varphi$ from the stability of atomic formulas. Now defining

$$\begin{array}{lll} \varphi \vee \psi & \text{by} & \neg\varphi \rightarrow \neg\psi \rightarrow \perp, \\ \exists x \varphi(x) & \text{by} & \neg\forall x \neg\varphi(x). \end{array}$$

we can derive exactly the same formulas as in classical arithmetic.

For brevity we do not give all details of our notion (and implementation) of term/derivation, but only collect those features which are relevant for our later arguments.

1. *Terms* have *types*, built up from ground types (here, for simplicity, just nat) by $(\varrho \to \sigma)$. Particular terms are the constructor constants 0 of type nat, S of type nat → nat and the recursion constants

   R of type $\varrho \to (\mathsf{nat} \to \varrho \to \varrho) \to \mathsf{nat} \to \varrho$.

   Any type has a *level*, defined by

   $$\begin{aligned} \mathsf{lev}(\tau) &= 0, \\ \mathsf{lev}(\varrho \to \sigma) &= \max(\mathsf{lev}(\varrho) + 1, \mathsf{lev}(\sigma)) \end{aligned}$$

   with τ a ground type.

2. Since formulas are for derivations what types are for terms, we also need the notion of the *level* of a formula

   $$\begin{aligned} \mathsf{lev}(A) &= 0, \\ \mathsf{lev}(\varphi \to \psi) &= \max(\mathsf{lev}(\varphi) + 1, \mathsf{lev}(\psi)), \\ \mathsf{lev}(\forall x \varphi(x)) &= \max(\mathsf{lev}(\mathsf{nat}) + 1, \mathsf{lev}(\varphi)) \end{aligned}$$

   with A atomic. Note that for simplicity we only consider formulas with quantified variables of level 0.

3. *Derivations* derive formulas. Particular derivations are the induction axioms

   R of the formula $\varphi(0) \to (\forall x.\varphi(x) \to \varphi(Sx)) \to \forall x \varphi(x)$,

   the truth axiom of the formula $\top$ and possibly some other axioms (or constructor constants for derivations) of true Π–formulas, i.e. formulas with only quantifier-free premises in implications.

4. If r derives ψ, then $\lambda u^{\varphi} r$ derives $\varphi \to \psi$. Similarly, if r has type σ, then $\lambda x^{\varrho} r$ has type $\varrho \to \sigma$. If r derives $\varphi(x)$ and if no assumption variable free in r assumes a formula with x among its free variables (this is known as *variable condition*), then $\lambda x r$ derives $\forall x \varphi(x)$.

5. If r derives $\varphi \to \psi$ and s derives φ, then rs derives ψ. Similarly, if r has type $\varrho \to \sigma$ and s has type ϱ, then rs has type σ. If r derives $\forall x \varphi(x)$ and s has type nat, then rs derives $\varphi(s)$.

6. Any term/derivation has a uniquely determined long normal form, where for R we have the usual conversion rules $Rrs0 \mapsto r$ and $Rrs(St) \mapsto st(Rrst)$. For example, if F is of type (nat → nat) → (nat → nat) and g is of type nat → nat, then the long normal form of

   g is $\lambda x.gx$

F is $\lambda zx.F(\lambda y.zy)x$
Fg is $\lambda x.F(\lambda y.gy)x$

We identify terms/derivations with the same long normal form.

For any term/derivation r let $\mathrm{lev}(r)$ denote the level of its type/formula.

For terms/derivations r with $\mathrm{lev}(r) = 0$, tree notations $a \in \mathrm{T}$ (see Section 2, taken with $\nu = 2$) and $m \in \mathbf{N}$ we define inductively the relation $\vdash^a_m r$, to be read r is SDH–*generated* with size a and rank m, by the following rules.

- *Variable Rule.* If $\vdash^a_m t_i\vec{y_i}$ for $i = 1, \dots, n$ with $n \geq 0$, then $\vdash^{a+1}_m xt_1 \dots t_n$.
- *Closure Rule 0.* $\vdash^1_m 0$.
- *Closure Rule S.* If $\vdash^a_m r$, then $\vdash^{a+1}_m Sr$.
- *Lemma Rule.* Let L be a lemma asserting a true Π–formula φ. If $\vdash^a_m r_i\vec{y_i}$ for $i = 1, \dots, n$ with $n \geq 0$, then $\vdash^{a+1}_m L\vec{r}$.
- ω^+*–Rule.* If $\tau(a) = \Omega, \vdash^{a^-}_m t$ and $\forall z \in \mathrm{T}_1 \forall n. \vdash^z_0 n \to \vdash^{a[z]}_m Rrsn\vec{t}$, then $\vdash^a_m Rrst\vec{t}$.
- $<_1$*–Rule.* If $\vdash^b_m r$ and $b <_1 a$, then $\vdash^a_m r$.
- *Cut Rule.* If $\vdash^a_m r\vec{y}$ with $\mathrm{lev}(r) \leq m$ and $\vdash^a_m t_i\vec{y_i}$ for $i = 1, \dots, n$ with $n \geq 1$, then $\vdash^{a+1}_m rt_1 \dots t_n$.

More precisely, we first inductively define $\vdash^a_m r$ for $a \in \mathrm{T}_1$ by the rules given excluding the ω^+–Rule, and based on this relation we then define $\vdash^a_m r$ for $a \in \mathrm{T}$ by all the rules given.

Variable Lemma 4.1. *If* $c \neq 0$ *and* $\mathrm{lev}(x) < k$, *then* $\vdash^{c \cdot k}_m x\vec{y}$.

Proof. By induction on $\mathrm{lev}(x)$. By induction hypothesis $\vdash^{c\cdot(k-1)}_m y_i\vec{z_i}$, hence $\vdash^{c\cdot(k-1)+1}_m x\vec{y}$ by the Variable Rule, hence $\vdash^{c\cdot k}_m x\vec{y}$ by the $<_1$–Rule. □

Substitution Lemma 4.2. *If* $\vdash^a_m r$ *and* $\vdash^b_m s_j\vec{y_j}$ *with* $\mathrm{lev}(s_j) \leq m$ *for* $j = 1, \dots, n$, *then* $\vdash^{b+a}_m r_{\vec{x}}[\vec{s}]$.

Proof. By induction on $\vdash^a_m r$. We write t^* for $t_{\vec{x}}[\vec{s}]$. *Variable Rule.* By induction hypothesis $\vdash^{b+a}_m t^*_i\vec{y_i}$, hence $\vdash^{b+a+1}_m xt^*_1 \dots t^*_n$ by the Variable Rule. Now if x is one of the variables x_j to be substituted by s_j, we must use the Cut Rule instead of the Variable Rule. This is possible since $\mathrm{lev}(s_j) \leq m$ by hypothesis and $\vdash^{b+a}_m s_j\vec{y_j}$ by hypothesis and the $<_1$–Rule. Then (if $n > 0$) the Cut Rule yields $\vdash^{b+a+1}_m s_jt^*_1 \dots t^*_n$, as required. In case $n = 0$ there are no t_i's and we have used the Variable Rule to generate $\vdash^{a+1}_m x_j$. But then $\vdash^{b+a+1}_m s_j$ holds by hypothesis and the $<_1$–Rule. For all other rules the claim follows easily from the induction hypothesis and the same rule. □

Cut Elimination Lemma 4.3. *If* $\vdash^{a}_{m+1} r$, *then* $\vdash^{D_1 a}_{m} r$.

Proof. By induction on $\vdash^{a}_{m+1} r$. *Variable Rule.* By induction hypothesis $\vdash^{D_1 a}_{m} t_i\vec{y_i}$, hence $\vdash^{(D_1 a)+1}_{m} xt_1 \dots t_n$ by the Variable Rule, hence $\vdash^{D_1(a+1)}_{m} xt_1 \dots t_n$ by the $<_1$–Rule. *Closure Rule 0.* Note that $1 <_1 D_1 1$. Hence $\vdash^{D_1 1}_{m} 0$ by the $<_1$–Rule. *Closure Rule S.* By induction hypothesis $\vdash^{D_1 a}_{m} r$, hence $\vdash^{(D_1 a)+1}_{m} Sr$ by the Closure Rule S, hence $\vdash^{D_1(a+1)}_{m} Sr$ by the $<_1$–Rule. *Lemma Rule.* By induction hypothesis $\vdash^{D_1 a}_{m} r_i\vec{y_i}$, hence $\vdash^{(D_1 a)+1}_{m} L\vec{r}$ by the Lemma Rule, hence $\vdash^{D_1(a+1)}_{m} L\vec{r}$ by the $<_1$–Rule. *ω^+–Rule.* Then $\vdash^{a}_{m+1} Rrst\vec{t}$ has been inferred from $\tau(a) = \Omega$, $\vdash^{a^-}_{m+1} t$, and

$$\forall z \in \mathrm{T}_1 \forall n. \vdash^{z}_{0} n \to \vdash^{a[z]}_{m+1} Rrsn\vec{t}.$$

By induction hypothesis $\vdash^{D_1 a^-}_{m} t$, and

$$\forall z \in \mathrm{T}_1 \forall n. \vdash^{z}_{0} n \to \vdash^{D_1 a[z]}_{m} Rrsn\vec{t}.$$

Now $D_1 a^- = (D_1 a)^-$, and since $\tau(a) = \Omega$ we have $D_1 a[z] = (D_1 a)[z]$. Hence $\vdash^{D_1 a}_{m} Rrst\vec{t}$ by the ω^+–Rule. *$<_1$–Rule.* By induction hypothesis $\vdash^{D_1 b}_{m} r$. Since from $b <_1 a$ we can infer $D_1 b <_1 D_1 a$, we get $\vdash^{D_1 a}_{m} r$ by the $<_1$–Rule. *Cut Rule.* By induction hypothesis $\vdash^{D_1 a}_{m} r\vec{y}$ and $\vdash^{D_1 a}_{m} t_i\vec{y_i}$. Since $\mathrm{lev}(r) \leq m + 1$, we have $\mathrm{lev}(t_i) \leq m$ and hence $\vdash^{D_1 a + D_1 a}_{m} rt_1 \dots t_n$ by the Substitution Lemma. But $(D_1(a + 1))[1] = D_1 a + D_1 a$, so $\vdash^{D_1(a+1)}_{m} rt_1 \dots t_n$ by the $<_1$–Rule. □

We now want to prove the Collapsing Lemma mentioned above. For its formulation we need the notion of the *first instance* $|r|$ *provided by a refutation* r *of* Π*–assumptions.* So let r be such a refutation, i.e. a derivation of a closed false atomic formula from assumptons $u_i{:}\,\varphi_i$, φ_i closed Π–formulas and φ_i true if φ_i is quantifier–free. Note that we identify derivations with the same long normal form, so we can always assume r to be normal. Hence r must be of one of the two forms below. In particular, it cannot contain induction axioms any more.

Case $r \equiv u_i\vec{r}$. If all derivations r_i among $\vec{r}$ actually derive true formulas (which can be decided, since the formulas are quantifier–free and can clearly be assumed to be closed), let $|r|$ be the list of all $|r_j|$, r_j term among $\vec{r}$. Otherwise, let $|r|$ be $|r_i\vec{u}|$, where r_i is the first derivation among $\vec{r}$ deriving a false quantifier–free formula and $\vec{u}$ are lemmas or assumptions of true formulas.

Case $r \equiv L\vec{r}$ with L a lemma. Then some r_i among $\vec{r}$ must derive a false quantifier–free formula, since r derives a false formula and the lemma L is assumed to be true. Let $|r|$ be $|r_i\vec{u}|$ for the first such r_i, with $\vec{u}$ lemmas or assumptions of true formulas.

First Collapsing Lemma 4.4. *Suppose $\vdash_0^a r$ with r a closed term of type* nat *or else a refutation of* Π*–assumptions. Let $|r|$ be the numerical value of r (in case r is a term), or the maximum value in the first instance provided by r (in case r is a derivation). Then $\vdash_0^{D_0 a} |r|$.*

Proof. By induction on $\vdash_0^a r$. *Variable Rule.* Then $r \equiv u_i\vec{r}$, and $\vdash_0^{a+1} u_i\vec{r}$ has been inferred from $\vdash_0^a r_i\vec{y}$. If all derivations r_i among $\vec{r}$ derive true formulas, then by definition $|r|$ is the list of all $|r_j|$, r_j term among $\vec{r}$, and $\vdash_0^{D_0 a}|r_j|$ holds by induction hypothesis, hence $\vdash_0^{D_0(a+1)}|r_j|$ by the $<_1$–Rule. Otherwise, again by definition $|r| = |r_i\vec{u}|$ where r_i is the first derivation among $\vec{r}$ deriving a false quantifier–free formula and $\vec{u}$ lemmas or assumptions of true formulas. Then $\vdash_0^{D_0 a} |r_i\vec{u}|$ holds by induction hypothesis, hence $\vdash_0^{D_0(a+1)} |r_i\vec{u}|$ by the $<_1$–Rule. *Closure Rule 0.* Clear, since $\vdash_0^{D_0 1} 0$ by the $<_1$–Rule. *Closure Rule S.* By induction hypothesis $\vdash_0^{D_0 a} |r|$, hence $\vdash_0^{D_0 a+1} S|r|$ by the Closure Rule S, hence $\vdash_0^{D_0(a+1)} S|r|$ by the $<_1$–Rule, and $|Sr| = S|r|$. *Lemma Rule.* Then $r \equiv L\vec{r}$, and $\vdash_0^{a+1} L\vec{r}$ has been inferred from $\vdash_0^a r_i\vec{y}$. By definition $|r| = |r_i\vec{u}|$ for some r_i among $\vec{r}$ deriving a false quantifier–free formula with $\vec{u}$ lemmas or assumptions of true formulas. By induction hypothesis$\vdash_0^{D_0 a}|r_i\vec{u}|$, hence $\vdash_0^{D_0(a+1)}|r_i\vec{u}|$ by the $<_1$–Rule. ω^+*–Rule.* Then $\vdash_0^a Rr_0st\vec{t}$ has been inferred from $\tau(a) = \Omega$, $\vdash_0^{a^-} t$, and

$$\forall z \in \mathrm{T}_1 \forall n. \vdash_0^z n \to \vdash_0^{a[z]} Rr_0sn\vec{t}.$$

We have to show $\vdash_0^{D_0 a}|Rr_0sk\vec{t}|$ with $k := |t|$. From $\tau(a) = \Omega$ we get $\tau(D_0 a) = \omega$ and $(D_0 a)[n] = D_0 a[z_n]$ with $z_0 = \omega$, $z_{n+1} = D_0 a[z_n]$, hence $z_1 = D_0 a[\omega] = D_0 a^-$. Since $D_0 a^- = (D_0 a)^-$, the induction hypothesis yields $\vdash_a^{(D_0 a)^-} k$. Since $(D_0 a)^- \in T_1$, we get $\vdash_0^{a[(D_0 a)^-]}|Rr_0sk\vec{t}|$ from our assumption, so again the induction hypothesis yields $\vdash_0^{D_0 a[(D_0 a)^-]} |Rr_0sk\vec{t}|$. But $D_0 a[z_1] = (D_0 a)[1]$, so the $<_1$–Rule gives $\vdash_0^{D_0 a}$ $|Rr_0sk\vec{t}|$. $<_1$*–Rule.* By induction hypothesis $\vdash_0^{D_0 b} |r|$, hence $\vdash_0^{D_0 a} |r|$ by the $<_1$–Rule. □

Second Collapsing Lemma 4.5. *If $\vdash_0^a n$ with $a \in \mathrm{T}_1$, then $\vdash_0^{G_a(1)} n$.*

Proof. By induction on $\vdash_0^a n$. *Closure Rule 0.* $\vdash_0^{G_1(1)} 0$ since $G_1(1) = 1$. *Closure Rule S.* By induction hypothesis $\vdash_0^{G_a(1)} n$, hence $\vdash_0^{G_{a+1}(1)} Sn$ by the Closure Rule S, since $G_{a+1}(1) = G_a(1) + 1$. $<_1$*–Rule.* By induction hypothesis $\vdash_0^{G_b(1)} n$, hence $\vdash_0^{G_a(1)} n$ by the $<_1$–Rule, since from $b <_1 a$ we can conclude $G_b(1) \leq G_a(1)$. □

Value Lemma 4.6. *If $\vdash_0^k n$, then $n < k$.*

Proof. By induction on $\vdash_0^k n$. *Closure Rule 0.* Clear, since $|0| = 0$. *Closure Rule S.* By induction hypothesis $n < k$, hence $Sn < k + 1$. $<_1$*–Rule.* Note first that from $b <_1 k$ we can conclude $b \in T_0$ and hence that b is a numeral l with $l < k$. By induction hypothesis $n < l$, hence $n < k$. □

We now construct an "initial" SDH–generation tree for any term/derivation r. The main point here is that by using the Cut Rule we can sort of "shortcut" an enormous SDH–generation tree. Consider for example an application term $(\lambda x r)s$ where s may be complex and r may contain many occurrences of x. The shortcut is achieved by considering instead of $(\lambda x r)s$, which may have a complex normal form, the two terms $(\lambda x r)y$ (with a variable y) and s separately.

Embedding Lemma 4.7. *Assume that all subterms/subderivations of r have levels $\leq m$. Then we can find k such that $\vdash_m^{\Omega \cdot k} r\vec{y}$*

Proof. By induction on r. *Case x.* The claim follows from the Variable Lemma with $k := \mathrm{lev}(x)+1$. *Case L.* By the Variable Rule, the Lemma Rule and the $<_1$–Rule we have $\vdash_0^{\Omega(k+1)} L\vec{y}$ if $\mathrm{lev}(y_i) < k$. *Case R.* Consider $Rywx\vec{y}$. We want to apply the ω^+–Rule to get $\vdash_m^{\Omega \cdot (k+1)} Rywx\vec{y}$ with $k = \mathrm{lev}(y)$. Since trivially $\tau(\Omega \cdot (k+1)) = \Omega$ and $\vdash_0^{(\Omega \cdot (k+1))^-} x$ we only have to show

$$\forall z \in \mathrm{T}_1 \forall n. \vdash_0^z n \rightarrow \vdash_m^{\Omega \cdot k + z} Rywn\vec{y}.$$

This is done by induction on $\vdash_0^z n$. *Closure Rule 0.* Since $Ryw0\vec{y}$ is identified with $y\vec{y}$ it suffices to show $\vdash_m^{\Omega \cdot k+1} y\vec{y}$. Since $\mathrm{lev}(y_i) < k$, this follows from the Variable Lemma and the Variable Rule. *Closure Rule S.* By induction hypothesis $\vdash_m^{\Omega \cdot k+z} Rywn\vec{y}$. Since $Ryw(Sn)\vec{y}$ is identified with $wn(Rywn)\vec{y}$ it suffices to show $\vdash_m^{\Omega \cdot k+z+1} wn(Rywn)\vec{y}$. This follows from the Variable Rule, since from $\vdash_0^z n$ we get $\vdash_0^{\Omega \cdot k+z} n$ by the Substitution Lemma. *$<_1$–Rule.* The claim follows from the induction hypothesis and the $<_1$–Rule. *Case* 0. Obvious. *Case S.* By the Variable Rule $\vdash_m^1 y$, hence $\vdash_m^\Omega y$ by the $<_1$–Rule, hence $\vdash_m^{\Omega+1} Sy$ by the Closure Rule S, hence $\vdash_m^{\Omega \cdot 2} Sy$ again by the $<_1$–Rule. *Case $\lambda x r$.* By induction hypothesis $\vdash_m^{\Omega \cdot k} r\vec{y}$, which says $\vdash_m^{\Omega \cdot k} (\lambda x r)x\vec{y}$, since both terms have the same normal form and hence are identified. *Case ts.* By induction hypothesis $\vdash_m^{\Omega \cdot k} ty\vec{y}$ and $\vdash_m^{\Omega \cdot k} s\vec{x}$. Also $\vdash_m^{\Omega \cdot k} y_i\vec{y_i}$. Note that by the $<_1$–Rule we can assume that we have the same k in all cases. Then $\vdash_m^{\Omega \cdot k+1} ts\vec{y}$ by the Cut Rule, hence $\vdash_m^{\Omega \cdot (k+1)} ts\vec{y}$ by the $<_1$–Rule. □

Now we obtain the desired estimate of instances in existential proofs, in terms of the slow growing hierarchy.

Theorem 4.8. *Let r be a closed term of type* nat → nat. *Then there is an m such that for all $n \geq m$*

$$|rn| \leq G_{D_0 D_1^{m+2} 0}(n).$$

Similarly, let r be a closed derivation of a closed formula $\forall x \exists y \, \mathrm{Spec}(x,y)$ with $\mathrm{Spec}(x,y)$ atomic, and let $u{:}\forall y \neg \mathrm{Spec}(x,y)$ be an assumption variable. Then there is an m such that for all $n \geq m$

$$|rnu| \leq G_{D_0 D_1^{m+2} 0}(n).$$

Proof. We only treat the second part, since the proof of the first part is identical (just leave u out). Consider $r(Sx)u$. By the Embedding Lemma 4.7 we find an m such that $\vdash_m^{\Omega\cdot m} r(Sx)u$. Hence $\vdash_0^c r(Sx)u$ with $c := D_1^m(\Omega \cdot m)$ by the Cut Elimination Lemma 4.3. Since $\vdash_0^{c\cdot n}(n-1)$ (by the Closure Rules and the $<_1$–Rule) we get $\vdash_0^{c\cdot(n+1)} rnu$ by the Substitution Lemma 4.2, and then $\vdash_0^a |rnu|$ with $a := D_0(c \cdot (n+1))$ by the First Collapsing Lemma 4.4. Hence $|rnu| < G_a(1)$ by the Second Collapsing Lemma 4.5 and the Value Lemma 4.6. So by Lemma 3.5 we get $|rnu| < G_d(n)$ with $d := D_0 D_1^{m+2}0$, for all $n \geq m$. □

5. COMPLEX EXISTENTIAL PROOFS

We have just seen that for any closed derivation r of a closed formula of the form $\forall x \exists y \, \mathrm{Spec}(x,y)$ with $\mathrm{Spec}(x,y)$ atomic there is an m such that for all $n \geq m$ the instance $|rn|$ provided by the existential derivation $\exists y \, \mathrm{Spec}(n,y)$ is bounded by $G_{D_0 D_1^{m+2}0}(n)$. We now show that this bound is sharp: We consider the particular specification

$$(D_0 D_1^{m+2}0)[x]^y = 0.$$

By Section 2 we already know that

$$\forall x \exists y (D_0 D_1^{m+2}0)[x]^y = 0$$

is true; here we show that for any m this formula is actually derivable in an arithmetical system. Since by the definition of the slow growing hierarchy in Section 3 the least k such that $(D_0 D_1^{m+2}0)[n]^k = 0$ is $\geq G_{D_0 D_1^{m+2}0}(n)$ (if we let $a[n] := a[0]$ for $\tau(a) = 1$), we can conclude that the bound of Section 4 is best possible.

The arithmetical system we work with is taken to deal with tree notations directly. However, of course we do not use any kind of transfinite induction but only structural induction on the build-up of tree notations. More precisely, we take the system T of tree notations treated in Section 2 based on the fixed $\nu = 2$; let a, b, c, z range over T. Then the function $a[z]$ giving the z–th element of the fundamental sequence for a is defined by the structural recursion in Section 2. We also use $a[z]^k := a[z][z]\ldots[z]$.

Let $W_n = \{a \in \mathrm{T}_1 | \exists k a[n]^k = 0\}$, and call a formula $\varphi(a)$ *n–progressive* if

$$\forall a . \forall z \in \|\tau(a)\|_n : \varphi(a[z]) \rightarrow \varphi(a)$$

where $\|\Omega\|_n := W_n$, $\|\omega\|_n := \{n\}$, $\|1\|_n := \{n\}$ and $\|0\|_n := \emptyset$.

Lemma 5.1. *Let $\varphi(a)$ be the formula $D_0 a \in W_n$. Then $\varphi(a)$ is n–progressive, i.e.*

$$\forall a . \forall z \in \|\tau(a)\|_n : D_0 a[z] \in W_n \rightarrow D_0 a \in W_n$$

Proof. Let a be fixed. Assume

$$\forall z \in \|\tau(a)\|_n : D_0 a[z] \in W_n. \tag{1}$$

We have to show that $D_0 a \in W_n$, i.e. that $\exists k (D_0 a)[n]^k = 0$. *Case* $a = 0$. The claim follows from $\omega[n] = n$ and $m[n]^k = m - k$. *Case* $\tau(a) = 1$. Note first that the set W_n clearly is closed against addition. Since $(D_0 a)[n] = (D_0 a[n]) \cdot (n+1)$, the claim follows from (1), which in our case is $D_0 a[n] \in W_n$. *Case* $\tau(a) = \omega$. Then $(D_0 a)[n] = D_0 a[n]$, and $D_0 a[n] \in W_n$ by (1). *Case* $\tau(a) = \Omega$. Then $(D_0 a)[0] = D_0 a[\omega]$, $(D_0 a)[m+1] = D_0 a[(D_0 a)[m]]$. We show $(D_0 a)[m] \in W_n$ by induction on m. For $m = 0$ we get $D_0 a[\omega] \in W_n$ by (1), since $\omega \in W_n$ (see Case $a = 0$). For the induction step we can assume $(D_0 a)[m] \in W_n$. But then $D_0 a[(D_0 a)[m]] \in W_n$ by (1). □

Lemma 5.2. *If the formula $\psi(a)$ is n–progressive, then so is*

$$\psi^*(a) :\equiv \forall c . \psi(c) \to \psi(c + D_1 a).$$

Proof. Let $\psi(a)$ be n–progressive, i.e.

$$\forall a . \forall z \in \|\tau(a)\|_n : \psi(a[z]) \to \psi(a). \tag{2}$$

We have to show that $\psi^*(a)$ is n–progressive. So let a be given and assume that

$$\forall z \in \|\tau(a)\|_n \forall c . \psi(c) \to \psi(c + D_1 a[z]). \tag{3}$$

We must show $\psi^*(a)$. So let also c be given and assume $\psi(c)$. We have to show $\psi(c + D_1 a)$. By (2) it suffices to prove

$$\forall z \in \|\tau(D_1 a)\|_n : \psi(c + (D_1 a)[z]). \tag{4}$$

Case $a = 0$. We must show $\forall z \in W_n : \psi(c + z)$, i.e. $z \in \mathrm{T}_1 \to z[n]^k = 0 \to \psi(c + z)$. This is done by induction on k. For $k = 0$ the claim follows from our assumption $\psi(c)$. For the induction step, assume $z \in \mathrm{T}_1$. Then $\|\tau(z)\|_n = \{n\}$ (if $z \neq 0$, but the case $z = 0$ is obvious). Hence by (2) with $c + z$ for a it suffices to show $\psi(c + z[n])$. But since $(z[n])[n]^k = z[n]^{k+1} = 0$ this follows from the induction hypothesis. *Case* $\tau(a) = 1$. We must show $\psi(c + (D_1 a)[n])$, i.e. $\psi(c + (D_1 a[n]) \cdot (n+1))$. We prove $\psi(c + (D_1 a[n]) \cdot m)$ by induction on m. For $m = 0$ we have $\psi(c)$ by our assumption. For the induction step, (3) with $c + (D_1 a[n]) \cdot m$ for c and the induction hypothesis yield $\psi(c + (D_1 a[n]) \cdot m + D_1 a[n])$. *Case* $\tau(a) = \omega, \Omega$. Since in this case $\tau(D_1 a) = \tau(a)$ and $(D_1 a)[z] = D_1 a[z]$ we get the claim (4) from (3) and our assumption $\psi(c)$. □

Theorem 5.3. *For any m, we can formally prove in arithmetic*

$$\forall x \exists y (D_0 D_1^m 0)[x]^y = 0.$$

Proof. We give an informal proof which can easily be formalized. Let n be fixed. Since the formula $\varphi(a) \equiv D_0 a \in W_n$ is n–progressive by Lemma 5.1, we know from Lemma 5.2 that also $\varphi^*(a)$, $\varphi^{**}(a), \ldots, \varphi^m(a)$ are n–progressive. Hence we have $\varphi^m(0)$, hence $\varphi^{m-1}(D_1 0)$ by the definition of ψ^*, hence inductively $\varphi^*(D_1^{m-1} 0)$, hence $\varphi(D_1^m 0)$, hence $\exists y (D_0 D_1^m 0)[n]^y = 0$. □

Note that in this proof we had to use induction axioms of complexity (alternating quantifiers) depending on m. The main idea here, i.e. the definition of $\psi^*(a)$ as a kind of "lift by an exponential of a", again goes back to early work of Gentzen.

APPENDIX: AN IMPLEMENTATION OF PROOFS

We now describe in some more detail our implementation of arithmetical terms and proofs, specifically of the proofs of the combinatorial theorems in Section 5. These proofs are of particular interest since — in spite of their simplicity — they exhaust the strength of arithmetic, in the sense that although $\forall n \exists k (D_0 D_1^m 0)[n]^k = 0$ is provable for any fixed m, the general theorem $\forall m \forall n \exists k (D_0 D_1^m 0)[n]^k = 0$ is not. We also use these implemented proofs to discuss the proofs–as–programs paradigm, and report on some experiments where we have used some of these proofs as programs. Finally we have some comments on how our implementation of proofs relates to others (like Lambda and Isabelle).

Obviously normalization of proofs is the central subject for the use of proofs as programs, and as already mentioned in the introduction, we implement normalization by the built–in evaluation mechanism of SCHEME. This is possible since we can work with the $\rightarrow\forall$–fragment of Gentzen's natural deduction calculus, for then the logical rules are just introduction and elimination rules for $\rightarrow$ and $\forall$, which correspond exactly to λ–abstraction and application.

However, to carry out this plan we have to overcome a difficulty: Derivations in natural deduction style generally contain free assumption variables $u{:}\varphi$ where φ can be any formula. Now when we want to normalize this derivation by evaluating it, we have to assign something to the variable $u{:}\varphi$. A moments reflection will show that this should be a procedure of the same "arity" $\alpha_1 \rightarrow \ldots \rightarrow \alpha_n \rightarrow 0$ as the formula φ (where the arity $\alpha(\varphi)$ of a formula φ is defined by $\alpha(A) = 0$ for an atomic formula A, $\alpha(\varphi \rightarrow \psi) = \alpha(\varphi) \rightarrow \alpha(\psi)$ and $\alpha(\forall x \varphi) = 0 \rightarrow \alpha(\varphi)$), and should be such that when it is applied to argument procedures $f_1, \ldots, f_n$, then the outcome should be a term $ur_1 \ldots r_n$, where r_i is a lambda–term whose value is f_i. We call this procedure the result of making u self–evaluating at the arity of the formula φ attached to u. From the above description it is rather clear that a precise definition of how u is made self–evaluating will involve another operation of independent interest, which inverts evaluation in the following sense: When it is given a procedure obtained by evaluating a (typable) lambda–term, it returns a lambda–term which evaluates to this very procedure.

An implementation of such operations **make-self-evaluating** and **proc->expr** in SCHEME can be given by simultaneous recursion: (For a treatment of the same problem in the general theory of functional programming languages, which avoids SCHEME's operational **gensym**–construct for creating new bound variables, cf. (Berger and Schwichtenberg 1991)).

```
(define (mse expr arity) ;mse for make-self-evaluating
  (if (equal? 0 arity)
      expr
      (lambda (arg)
        (mse (list expr (proc->expr arg (arg-arity arity)))
             (val-arity arity)))))

(define (proc->expr proc arity)
  (if (equal? 0 arity)
      proc
      (let* ((arity0 (arg-arity arity))
             (symbol (gensym-of-arity arity0)))
        (list 'lambda (list symbol)
              (proc->expr
                (proc (mse symbol arity0))
                (val-arity arity))))))
```

Another point in our implementation of proofs is that we want to shift the "computational part" of an arithmetical proof as much as possible to a rewrite system. Hence terms occurring in formulas are always normalized according to the recursive definitions of the function symbols occurring in the term. Since normalization is done by evaluation, we have to deal with the same problem just discussed for derivations. The solution is the same: make the free variables self–evaluating. However, since we only treat first–order terms we can make a variable self–evaluating by just quoting it. So we want e.g.

```
((+-nat 2) 3) => 5
((+-nat 2) 'x) => ((+-nat 2) x)
((+-nat 'x) 0) => x
((+-nat 'x) 2) => (suc-nat (suc-nat x))
((+-nat 2) (suc-nat 'x)) => (suc-nat ((+-nat 2) x))
```

This can be achieved if we define +-nat by:

```
(define +-nat (lambda (m) (lambda (n)
  (cond ((zero-nat? n) m)
        ((suc-nat? n) (suc-nat ((+-nat m) (pred-nat n))))
        (else (list (list '+-nat m) n))))))
```

Here we have used

```
(define (suc-nat x) (if (and (integer? x) (not (negative? x)))
                        (+ 1 x)
                        (list 'suc-nat x)))

(define (zero-nat? x) (and (integer? x) (zero? x)))

(define (suc-nat? x)
  (or (and (integer? x) (positive? x))
      (and (pair? x) (equal? 'suc-nat (car x)))))

(define (pred-nat x)
  (cond ((and (integer? x) (positive? x)) (- x 1))
        ((and (pair? x) (equal? 'suc-nat (car x))) (cadr x))
        (else (error "can't form" 'pred-nat x))))
```

*-nat is defined similarly.

To implement proofs we first have to build formulas from terms. Since the formulas serve as types in derivations and in particular determine which rules are to be applied we must be careful never to "compute" say propositional formulae, since this will destroy their logical form and hence spoil the correctness of a derivation. So we distinguish between boolean objects on the one side (where the propositional connectives can be computed) and formulas or objects of type prop on the other side (where such computations are not allowed). Specifically, equality is taken as a binary boolean-valued function =-nat, i.e. ((=-nat 3) 5) evaluates to the boolean object #F (also denoted by ()), whereas the corresponding prime formula is given by (atom ((=-nat 3) 5)) (which evaluates to (atom ()), our prime formula for falsity), where atom transforms a boolean expression into an expression of a formula. The formal definitions of =-nat and atom are

```
(define  =-nat (lambda (m) (lambda (n)
  (cond ((zero-nat? m)
         (cond ((zero-nat? n) true)
               ((suc-nat? n) false)
               (else (list (list '=-nat m) n))))
        ((suc-nat? m)
         (cond ((zero-nat? n) false)
               ((suc-nat? n)
                ((=-nat (pred-nat m)) (pred-nat n)))
               (else (list (list '=-nat m) n))))
        (else (list (list '=-nat m) n))))))

(define (atom x) (list 'atom x))
```

<=-nat is defined similarly.

Also, we have two versions of implication: imp# to build boolean terms and imp to build formulas, such that e.g.

```
((imp# #T) #F) => ()
(imp (atom #T) (atom #F)) => (imp (atom #T) (atom ()))
```

This is achieved by defining

```
(define imp# (lambda (p) (lambda (q)
  (cond ((false? p) true)
        ((true? p) q)
        ((true? q) true)
        (else (list (list 'imp# p) q))))))

(define (imp x y) (list 'imp x y))
```

The universal quantifier over natural numbers is taken as all-nat of type (nat->prop)->prop. For instance, the formula $\forall n : 0 + n = n$ is represented as the self-evaluating SCHEME object

```
(all-nat (lambda (n) (atom ((=-nat ((+-nat 0) n)) n))))
```

The precise SCHEME definition of `all-nat` is

```
(define (all-nat proc)
  (let ((symbol (gensym "X^0_")))
    (list 'all-nat
          (list 'lambda (list symbol) (proc symbol)))))
```

where `gensym` is used to avoid clashes of bound variables.

For the implementation of proofs first note that we can describe a proof by three objects: a context, a formula and an expression which evaluates — provided the context symbols are made self-evaluating — to itself in case formula is of level 0, and to a procedure in case the formula is not of level 0. So the expression is a (type–free) lambda term corresponding to the build–up of the derivation from axiom–constants and assumption–variables by elimination and introduction rules. A context is an association list $((x_1\varrho_1)\dots(x_m\varrho_m)(u_1\varphi_1)\dots(u_n\varphi_n))$ (not necessarily in this order) assigning types and formulas to symbols, where

- the symbols $x_1,\dots,x_m,u_1,\dots,u_n$ are distinct,
- if y is a symbol in the context of an assumption formula φ_i, then y does not appear among $x_1,\dots,x_m$, and
- the contexts of all assumption formulas are consistent; their sup is called the critical context of the given context.

$(x_1\varrho_1)\dots(x_m\varrho_m)$ is called the free context, the sup of the free context and the critical context the object context, and $(u_1\varphi_1)\dots(u_n\varphi_n)$ the assumption context of the given context.

To give an example of an implemented proof, let us see how the usual inductive proof of $\forall n : 0 + n = n$ can be represented. We first write this proof in a self–explaining natural deduction notation, which in our implementation is also machine–readable.

```
(define 0+n=n-proof
  (elim (ind-axiom-at |(n)(0+n=n)|)
        truth-axiom
        (intro (var 'IH |0+n=n|)
               (list 'IH |0+n=n|)
```

```
                    (list 'n nat))))
```

where the induction axiom at $\forall n : 0 + n = n$ is of course an axiom–constant of the formula

$$0 + 0 = 0 \rightarrow (\forall n.0 + n = n \rightarrow 0 + Sn = Sn) \rightarrow \forall n : 0 + n = n.$$

Note that the formula $0 + 0 = 0$, i.e. (atom ((=-nat ((+-nat 0) 0)) 0)), evaluates to and hence is identified with (atom #T), and similarly the formula $0 + Sn = Sn$, i.e. (atom ((=-nat ((+-nat 0) (suc-nat n))) (suc-nat n))), evaluates to and hence is identified with $0 + n = n$, i.e. with (atom ((=-nat ((+-nat 0) n)) n)). This is the reason why in the above proof the initial case of the induction is just the truth axiom of the formula (atom #T), and in the induction step the induction hypothesis is identical with the claim.

The internal representation of this proof is a three–element list consisting of the context — which is empty in this case — , the derived formula and the expression

```
(((ind-at (quote ...))
   truth-axiom-symbol)
   (lambda (n) (lambda (IH) IH)))
```

where ... is to be replaced by the internal representation of the formula $\forall n : 0 + n = n$ we make induction on.

To give an example of a proof from assumptions, let us derive $\forall mn : n \leq n + m$ from $\forall n : n \leq Sn$ and reflexivity and transitivity of $\leq$.

```
(define leq-proof ;proves (mn)(n<=n+m)
  (elim (ind-axiom-at |(mn)(n<=n+m)|)
        (var 'Refl-<= |(n)(n<=n)|) ;init
        (intro ;step
          (elim (var 'Trans-<= |(nmk)(n<=m->m<=k->n<=k)|)
                |n| |n+m| |S(n+m)|
                (elim (var 'IH |(n)(n<=n+m)|) |n|)
                (elim (var 'Lemma |(n)(n<=Sn)|) |n+m|))
          (list 'n nat)
          (list 'IH |(n)(n<=n+m)|)
          (list 'm nat))))
```

In the internal representation of this proof, the context now consists of the three pairs (Lemma ...1), (Refl-<= ...2) and (Trans-<= ...3), where ...1, ...2 and ...3 are the internal representations of the respective formulas. The expression in this case is

```
(((IND-AT (QUOTE ...))
    REFL-<=)
    (LAMBDA (M) (LAMBDA (IH) (LAMBDA (N)
      (((((TRANS-<= N) ((+-NAT N) M)) (SUC-NAT ((+-NAT N) M)))
            (IH N))
            (LEMMA ((+-NAT N) M))))))))
```

There is one more point which can be demonstrated in this example. If from the above proof of $\forall mn : n \leq n + m$ we want to conclude $\forall n : n \leq n + 2$ by the rule of $\forall$–elimination, we just apply the procedure defined by the expression above to 2, that is we evaluate

```
(((IND-AT (QUOTE ...))
     REFL-<=)
     step)
     2)
```

where step stands for the expression above. Now (ind-at (quote ...)) is defined to be a procedure which when supplied with init, step and arg where arg is a natural number gives the result of applying step as many times as arg says to init. In our case, we get the outcome of the evaluation of

```
((step 1) ((step 0) Refl-<=))
```

that is of

```
((step 1) (LAMBDA (n) (((((TRANS-<= n) n) (SUC-NAT n))
                             (REFL-<= n))
                             (LEMMA n))))
```

that is of

```
(LAMBDA (n)
  (((((TRANS-<= n) (SUC-NAT n)) (SUC-NAT (SUC-NAT n)))
        (((((TRANS-<= n) n) (SUC-NAT n))
              (REFL-<= n))
              (LEMMA n)))
        (LEMMA (SUC-NAT n))))
```

So $\forall n : n \leq n+2$ is derived in the indicated way by two applications of the transitivity of $\leq$ and one application of the reflexivity of $\leq$, but without an application of an induction axiom. The precise definition of ind-at is

```
(define (ind-at all-formula)
  (lambda (init)
    (lambda (step)
      (lambda (arg)
        (cond ((zero-nat? arg) init)
               ((suc-nat? arg)
                ((step (pred-nat arg))
                 ((((ind-at all-formula)
                    init) step) (pred-nat arg))))
               (else ...))))))

(define (ind-axiom-at all-formula)
  (list (context-of-formula all-formula) ;should be empty
        (formula-of-ind-at all-formula)
        (list 'ind-at (list 'quote all-formula))))
```

Here ... describes how the expression is to be reproduced in case arg is neither 0 nor a successor. Written out fully ... is

```
(let* ((init-arity (arity-of-formula
                     (specialize all-formula zero-nat-term)))
       (step-arity (cons-arity 0 (cons-arity init-arity
                                             init-arity))))
  (mse (list (list (list (list 'ind-at (list 'quote
                                             all-formula))
                         (proc->expr init init-arity))
                   (proc->expr step step-arity))
             arg)
       init-arity))
```

We now treat a slightly more complex example of an arithmetical proof, which can be used to demonstrate the proofs-as-programs paradigm in an easy case.

We prove by induction on n that n can be divided by $m+1$ with some quotient q and remainder r, as follows

```
(define quot-rem-proof
  (elim
    (ind-axiom-at |(nm)(Eqr)(n=(m+1)*q+r&r<=m)|)
    (intro ;init
      (elim (var 'u1 |(qr)(0=(m+1)*q+r->r<=m->falsity)|)
            |0| |0| truth-axiom truth-axiom)
      (list 'u1 |(qr)(0=(m+1)*q+r->r<=m->falsity)|)
      (list 'm nat))
    (intro ;step
      (elim
        (var 'IV |(m)(Eqr)(n=(m+1)*q+r&r<=m)|)
        |m|
        (intro
          (elim
            (var 'Lemma-<=
                 |(mr)(r<=m->-(r+1<=m)->-(r=m)->falsity)|)
            |m| |r|
            (var 'u2 |r<=m|)
            (intro
              (elim
                (var 'u3 |(qr)(n+1=(m+1)*q+r->r<=m->falsity)|)
                |q| |r+1|
                (var 'u4 |n=(m+1)*q+r|)
                (var 'u5 |r+1<=m|))
              (list 'u5 |r+1<=m|))
            (intro
              (elim
                (var 'u3 |(qr)(n+1=(m+1)*q+r->r<=m->falsity)|)
                |q+1| |0|
                (elim
                  (var 'Lemma-=
                       |(nmqr)(r=m->n=(m+1)*q+r->n=(m+1)*q+m)|)
                  |n| |m| |q| |r|
                  (var 'u6 |r=m|)
                  (var 'u4 |n=(m+1)*q+r|))
                truth-axiom)
              (list 'u6 |r=m|)))
          (list 'u2 |r<=m|)
      (list 'u4 |n=(m+1)*q+r|)
      (list 'r nat)
      (list 'q nat)))
  (list 'u3 |(qr)(n+1=(m+1)*q+r->r<=m->falsity)|)
  (list 'm nat)
  (list 'IV |(m)(Eqr)(n=(m+1)*q+r&r<=m)|)
  (list 'n nat)))
```

This clearly formalizes the informal proof by cases: If $r < m$ let $q' = q$ and $r' = r + 1$, and if $r = m$ let $q' = q + 1$ and $r' = 0$. Note that in the formal proof above we have freely used true Π–assumptions as lemmata, since they have no computational content and hence don't affect the use of this proof as a program.

If we now specialize this proof to particular numbers (using ∀–elimination) and then normalize it, all uses of induction axioms disappear as we just have demonstrated. As described in Section 4 we can then read off the first instance provided by the resulting refutation from Π–assumptions. Formally, we can easily implement a procedure `instance-from-refutation-of-Pi-assumptions` by the same recursion as in Section 4, and with

```
(define (qr n m)
  (instance-from-refutation-of-Pi-assumptions
    (normal-form-of-proof
      (elim quot-rem-proof n m))))
```

we obtain (in a few seconds even on a PC)

```
(qr 7-term 2-term) => ((() NAT 2) (() NAT 1))
```

since 7 divided by 3 has quotient 2 and remainder 1.

Finally we come to an implementation of the proofs in Section 5. The tree notations we have to deal with are viewed as a free algebra generated from 0 by one unary constructor S and two binary constructors C_0 and C_1. Intuitively, $C_0(a, b)$ corresponds to the tree notation written $a + D_0 b$ in Section 2, and $C_1(a, b)$ corresponds to $a + D_1 b$; for example, $C_0(0, 0) = \omega$ and $C_1(0, 0) = \Omega$. Now `=-tree`, `+-tree` and `*-tree` can be defined in the obvious way, similarly to what we did for `nat`. The functions $\tau(a)$, $a[z]$, $a[z]^k$ and the predicate T_1 can be defined by the same recursions as in Section 2.

To implement our proofs we have to start with the initial 0–case of Lemma 5.1 and prove $D_0 0 \in W_n$, i.e. $\exists k(D_0 0)[n]^k = 0$. Note that $D_0 0$ denotes ω and is represented here as $C_0(0,0)$. We obtain this proof easily from the Lemma $(D_0 0)[n]^{n+1} = 0$. The next case of Lemma 5.1 is that of a successor tree Sa, i.e. we prove

$$D_0 a \in W_n \rightarrow D_0(Sa) \in W_n \tag{0.1}$$

or more explicitely $\exists k(D_0 a)[n]^k = 0 \rightarrow \exists k(D_0 Sa)[n]^k = 0$. Again the construction of the second k from the first one can be given explicitely with our tree functions available, and hence we can use the Π–formula $(D_0 a)[n]^k = 0 \rightarrow (D_0 Sa)[n]^{S(kn+k)} = 0$ as a lemma. The complete proof is

```
(define P-01
;proves (na)((Ek)(C00a)[n]^k=0->(Ek)(C00(Sa))[n]^k=0)
  (intro
    (elim
      (var 'u |-(k)-(C00a)[n]^k=0|)
      (intro
        (elim
          (var 'v |(k)-(C00(Sa))[n]^k=0|)
          (elim
            (var 'Lemma-01
|(nak)((C00a)[n]^k=0->(C00(Sa))[n]^(S(k*n+k))=0)|)
            |n| |a| |k|
            (var 'w |(C00a)[n]^k=0|)))
        (list 'w |(C00a)[n]^k=0|)
        (list 'k nat)))
    (list 'v |(k)-(C00(Sa))[n]^k=0|)
    (list 'u |-(k)-(C00a)[n]^k=0|)
    (list 'a tree)
    (list 'n nat)))
```

The ω–case of Lemma 5.1, i.e. the proof of

$$\tau(a) = \omega \rightarrow D_0 a[n] \in W_n \rightarrow D_0 a \in W_n \tag{0.ω}$$

is similar; we use $\tau(a) = \omega \rightarrow (D_0 a[n])[n]^k = 0 \rightarrow (D_0 a)[n]^{k+1} = 0$ as a lemma. More interesting is the final Ω–case of Lemma 5.1, i.e. the proof of

$$\tau(a) = \Omega \rightarrow (\forall z \in W_n : D_0 a[z] \in W_n) \rightarrow D_0 a \in W_n \tag{0.Ω}$$

or somewhat more explicitely of

$$\tau(a) = \Omega \rightarrow (\forall kz . z \in \mathrm{T}_1 \rightarrow z[n]^k = 0 \rightarrow \exists k(D_0 a[z])[n]^k = 0) \rightarrow \exists k(D_0 a)[n]^k = 0.$$

As in the proof of Lemma 5.1 we use here an auxiliary theorem

$$\tau(a) = \Omega \to (\forall kz. z \in \mathrm{T}_1 \to z[n]^k = 0 \to \exists k(D_0 a[z])[n]^k = 0) \to \exists k(D_0 a)[m][n]^k = 0,$$

which is proved separately by induction on m, using the lemmata

$$\tau(a) = \Omega \to (D_0 a)[m] \in \mathrm{T}_1$$

$$\tau(a) = \Omega \to (D_0 a)[0] = D_0 a[\omega]$$

$$\tau(a) = \Omega \to (D_0 a)[m+1] = D_0 a[(D_0 a)[m]]$$

$$(D_0 0)[n]^{n+1} = 0$$

Note that we do not give an explicit form of our theorem (i.e. with open premises), as in the previous cases. The reason is that the resulting k is obtained roughly by n-fold iteration of the function given by the premise

$$\forall kz. z \in \mathrm{T}_1 \to z[n]^k = 0 \to \exists k(D_0 a[z])[n]^k = 0$$

and our term language does not contain a construct for function-iteration. Rather, we use the proofs-as-programs paradigm here to provide such a construct.

The last proof we have implemented is that of the initial 0–case (which in fact is the most complex case) of Lemma 5.2 for the formula $D_0 a \in W_n$, which was proved to be n–progressive in Lemma 5.1. So we have to prove

$$D_0 c \in W_n \to D_0(c + \Omega) \in W_n. \tag{1.0}$$

This is obtained from $(0.\Omega)$ with $c + \Omega$ for a. So we have to prove its second premise from the assumption $D_0 c \in W_n$, i.e.

$$\exists k(D_0 c)[n]^k = 0 \to z \in \mathrm{T}_1 \to z[n]^k = 0 \to \exists k(D_0(c + z))[n]^k = 0. \tag{1.0 – aux1}$$

This is done by induction on k, where in the induction step we need the lemma $z \in \mathrm{T}_1 \to z[n] \in \mathrm{T}_1$ and

$$z \in \mathrm{T}_1 \to \exists k(D_0(c + z[n]))[n]^k = 0 \to \exists k(D_0(c + z))[n]^k = 0. \tag{1.0 – aux2}$$

This is proved by cases on z (formally by tree-induction on z), using our previous theorems (0.1), $(0.\omega)$ and also the lemma $((c + a) + D_0 b)[n] = c + (a + D_0 b)[n]$. Note that we do not give an explicit form of (1.0-aux2), since this would require a lazy if-then-else-construct in our term language, which we don't have. In fact, addition of such a construct to our term language is impossible in our present SCHEME-implementation, since SCHEME employs eager evaluation. Rather, we again use the proofs-as-programs paradigm here to provide such a construct.

Now when we come to actually use these proofs as programs, there is the obvious practical difficulty that the proofs were just designed to require very fast growing functions to instantiate the existential quantifier. For instance, already the proof of $D_0\omega \in W_n$ requires an exponential function (roughly n^n) for its instantiation. So only very small initial cases like $D_0 2 \in W_n$ can actually be tested, and we have successfully done so.

Now let us look back and ask ourselves what we have achieved by our implementation of these proofs. First, of course, we have machine-checked them and can be sure that we have not overlooked some cases or assumptions; this has been the motivation for de Bruijn's Automath-project (cf. (de Bruijn 1990) for a recent survey). On the other hand, since we have the whole proof available (without eating up too much space, since we don't need to code formulas in the proof, but just the build-up from introduction and elimination rules for → and ∀, i.e. just a type-free lambda term), we can use it e.g. as a program, as done here.

A further possibility is to modify the proof if the specification is changed, or else to "prune" it (this is the terminology of (Goad 1980)) when we have some additional knowledge on the input data and use this knowledge to derive some of the assumptions in case distinctions, making these case distinctions superfluous and hence the whole proof prunable. Note that the new pruned proof can yield an extensionally different program for the same specification; hence we have a program transformation here which in fact changes the function computed by the program.

All this is a sufficient reason for us to actually carry the whole proof along in an interactive proof system. This is in contrast to e.g. Paulson's Isabelle (cf. Paulson 1990) or Fourman's Lambda system (cf. Finn, Fourman, Francis and Harris 1990), who do the opposite and only save theorems, throwing their proofs away.

BIBLIOGRAPHY

1. Abelson, H., Sussman, G.J. (1985) Structure and interpretation of computer programs. MIT Press, Cambridge.

2. Arai, T. (1989) A Slow Growing Analogue of Buchholz' Proof. Nagoya University, Department of Mathematics. To appear in Annals Pure Appl. Logic.

3. Berger, U., Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed lambda calculus. Proc. 6th IEEE Symp. on Logic in Computer Science, pp. 203–211.

4. Buchholz, W. (1987) An independence result for $(\Pi^1_1 - CA) + BI$. Annals Pure Appl. Logic *33*, 131–155.

5. Buchholz, W., Wainer, S. (1987) Provably computable functions and the fast growing hierarchy. Logic and Combinatorics. Contemp. Math. 65, AMS, pp. 179–198.

6. Constable, R. et al. (1986) Implementing mathematics with the Nuprl proof development system. Prentice Hall, Englewood Cliffs, New Jersey.

7. de Bruijn, N.G. (1990) A plea for weaker frameworks. In: BRA Logical Frameworks Workshop, Sophia Antipolis.

8. Finn, S., Fourman, M.P., Francis, M., Harris, R. (1990) Formal System Design — Interactive Synthesis Based on Computer–Assisted Formal Reasoning. In: L.J.M Claasen (ed.). Formal VLSI Specification and Synthesis, I, North–Holland, Amsterdam, pp. 139–152.

9. Girard, J.Y. (1981) Π^1_2-Logic. Annals Math. Logic *21*, 75–219.

10. Goad, C. (1980) Computational uses of the manipulation of formal proofs. Stanford Dept. of Computer Science, Report No. STAN-CS-80-819.

11. Howard, W.A. (1972) A system of abstract constructive ordinals. J. Symbolic Logic 37, *2*, 355–374.

12. Howard, W.A. (1980) Ordinal analysis of terms of finite type. J. Symbolic Logic *45*, 3, 493–504.

13. Martin–Löf, P. (1980) Constructive mathematics and computer programming. In: Logic, Methodology and the Philosophy of Science VI. North Holland, Amsterdam, pp. 153–175

14. Paulson, L.P. (1990) Isabelle: the Next 700 Theorem Provers. In: P. Odifreddi (editor), Logic and Computer Science. Academic Press, London, pp. 361–386.

15. Rees, J., Clinger, W. (eds) (1986) Revised^3 report on the algorithmic language Scheme. AI Memo 848a, MIT, Cambridge.

16. Schmerl, U. (1982) Number theory and the Bachmann/Howard ordinal. In: J. Stern (ed.) Proc. Herbrand Symposium. North Holland, Amsterdam, pp. 287–298.

17. Schwichtenberg, H. (1986) A normal form for natural deductions in a type theory with realizing terms. Atti del Congresso Logica e Filosofia della Scienza, oggi. San Gimignano 1983. Vol. I - Logica, CLUEB, Bologna, pp. 95–138.