Recursion and Coding in PA by Techniques of Computer Programming

Paul J. Voda
Institute of Informatics, Comenius University Bratislava,
Mlynská dolina, 842 15 Bratislava, Slovakia,
email: voda@fmph.uniba.sk
homepage: www.fmph.uniba.sk/∼voda

December 2001

1 Introduction

The recursive development of Peano Arithmetic (or its fragments) is needed in logic in order to derive the Incompleteness Theorems of Gödel. It is also needed in the Theory of Programming when one bases a programming language on natural numbers ($\mathbb{N}$) as it is the case with CL [KV98, Vo01]. Both goals require a schema of encoding of finite sequences into $\mathbb{N}$. Gödel used his β-function to encode the sequences (see [Sho67]). Although the encoding was simplified over time, most notably by Quine (see [Smu92]), Kripke (see [BJ80]), and Nelson (see [Nel86, HP93]), the basic approach: first codes of sequences then primitive recursion has not.

In this paper we reverse the situation. We first show how to introduce into PA in a surprisingly simple way the primitive recursive functions by showing PA closed under course of values recursion. Essentially no coding is needed for this. We do this by a simplified use of markers invented by [Nel86] (see also [HP93]). We use the markers also in Sect. 3 to introduce the graph of exponentiation $2^x = y$ as a Σ_0-predicate. This was discovered by Bennett [Ben62]. Ours is probably the simplest known definition which we did independently (see [Vo98]) of a quite similar (but slightly more complicated) one by Immerman [Imm98] who was inspired by Lindell [Lind9?]. Immerman did not work in PA but in a slightly different context of finite models (which is a branch of complexity theory).

We use primitive recursive functions in Sect. 4 to develop the coding. We were inspired here by computer programming which, because of its natural emphasis on readability (programs tend to be long), has developed a host of simple modern coding techniques which we think should be of interest to logicians. For examples in the programming languages Haskell and CL see [Tho99, KV98, Vo01].

We accordingly introduce into PA codes of finite sequences - called *lists* in computer programming - together with a number of basic list processing functions. Lists are obtained by the use of a pairing function which, in contrast to its standard use in logic, has a single atom (a number not in the range of the function). This simple modification has far reaching consequences because it enables to use in PA the coding techniques of the seminal programming language LISP which has strongly influenced modern programming. We illustrate a simple arithmetization of syntax with what is known in computer programming as union types.

The development can be stopped here in logic because the proofs of both Incompleteness Theorems of Gödel can be now given. For a programming language such as CL which is based on PA the development must go on because one needs a programming comfort of a strong schema of recursion and of a readable programming langauge whose statements are theorems of PA.

The strong schema needed in computer programming is *course of values recursion with measure* and it can be traced (minus the measure) to the early work of R. Péter [Pet32]. We have investigated the schema in [KV98] and give a much simpler proof of it being closed under primitive recursion in Sect. 5. Moreover, the proof is carried out in $I\Sigma_1$ for the first time. This is not a case of straightforward formalization as the proof requires a rather subtle use of a relatively unknown theorem of Parsons on Π_2-induction rules [Par72].

The strong schema of recursion gives the programmers a strong tool for the writing of programs but as it is formulated the programs are barely readable. Since readabilty is crucial to computer programming, Sect. 6 introduces a simple language of (Horn) clauses which is reduced to the schema. The language, called CL, is our contribution to the theory of programming and we hope that also logicians will find it interesting.

The author would like to acknowledge the great contribution of his colleague Ján Komara. In a fruitful collaboration spanning over the last ten years we have designed the syntax and semantic of CL, implemented its compiler and intelligent proof assistant for PA, developed four courses based on CL, and prepared lecture notes for the courses. The courses which we teach at our university are: Introduction to declarative programming, First-order logic and arithmetic, Specification and verification of computer programs, and Introduction to computability theory. All together about four hundred students pass our undergraduate courses yearly.

1.1 The language of PA. The language of PA is formed from the constant 0 and function symbols for $S(x)$, $x+y$ and $x \cdot z$. We use *numerals* $\overline{n}$ as abbreviations for terms $\overbrace{S \ldots S}^{n}(0)$, we also abbreviate the numeral $\overline{d}$ by the decimal numeral denoting d, and we write x^2 instead of $x \cdot x$.

The *standard* structure $\mathcal{N}$ for this language has the domain $\mathbb{N}$ and the usual interpretation of the function symbols. It satisfies the usual six axioms of Peano Arithmetic plus infinitely many induction axioms (see [HP93]). We will work in

PA extended by explicit definition:

$$x \leq y \leftrightarrow \exists z\, x + z = y \ .$$

Bounded quantifiers are $\exists x{\leq}\tau\, \phi$ and $\forall x{\leq}\tau\, \phi$ which we view as abbreviations for $\exists x(x \leq \tau \wedge \phi)$ and and $\forall x(x \leq \tau \rightarrow \phi)$ respectively. Σ_0*-formulas* are constructed from the atomic formulas $\tau_1 = \tau_2$ and $\tau_1 \leq \tau_2$ by propositional connectives and bounded quantification.

A formula $\phi[\vec{x}]$ is Σ_0 *in* $\mathcal{N}$ if $\mathcal{N} \models \forall \vec{x}(\phi[\vec{x}] \leftrightarrow \psi[\vec{x}])$ for a Σ_0-formula $\psi[\vec{x}]$. A predicate P is Σ_0 if for a ϕ which is Σ_0 in $\mathcal{N}$ we have $P(\vec{x})$ iff $\mathcal{N} \models \phi[\vec{x}]$. Σ_0-formulas are in the literature also designated by Δ_0.

1.2 Fragments of PA. The $I\Sigma_0$*-fragment* of PA retricts the induction axioms to Σ_0-formulas. For a fragment T of PA a formula ϕ is Σ_0 *in* T iff $T \vdash \phi \leftrightarrow \psi$ for a Σ_0-formula ψ.

A formula ϕ is Σ_1 *in* $\mathcal{N}$ (Σ_1 *in* T) iff $\mathcal{N} \models \phi \leftrightarrow \exists x\psi$ ($T \vdash \phi \leftrightarrow \exists x\psi$) for a Σ_0-formula ψ. Note that $I\Sigma_0$-proves $\exists x\exists y\phi \leftrightarrow \exists z\exists x{\leq}z\exists y{\leq}z\phi$. We similarly define Π_1 formulas from $\forall x\psi$ and also all formula Σ_i and Π_i where $i \geq 2$. Δ_{k+1} *in* T formulas are both Σ_{k+1} and Π_{k+1} in T.

The *fragment* $I\Sigma_k$ of PA has the induction axioms restricted to Σ_k-formulas. We will use below the well known fact (see [HP93]) that $I\Sigma_k$ proves Π_k-induction axioms and when we introduce into a fragment T a function by

$$f(\vec{x}) = y \leftrightarrow \phi[\vec{x}, y]$$

with ϕ Σ_k in T then ϕ is Δ_k in T because of the existence and uniqueness conditions where we have $T \vdash \phi[\vec{x}, y] \leftrightarrow \forall z(\phi[\vec{x}, z] \rightarrow z = y)$.

We assume in the following the initial development of arithmetic in $I\Sigma_0$ with the modified subtraction ($\dot{-}$), integer division ($\div$), and remainder (mod) functions introduced and with the usual properties of the basic functions proved.

2 Introduction of Primitive Recursion

In this section we will show $I\Sigma_1$ closed under course of values recursion. This is done with almost no coding except by a few simple predicates introduced in the following paragraph.

2.1 Some Σ_0-predicates. The predicate $x \mid y$ of *divisibility* holding if x divides y is a Σ_0-predicate by:

$$x \mid y \leftrightarrow \exists z\, y = x{\cdot}z \ .$$

A positive number p is a power of two, in symbols $Pow_2(p)$, iff all its divisors $d > 1$ are themselves divisible by 2. This is captured by the following Σ_0 in $I\Sigma_0$ predicate:

$$Pow_2(p) \leftrightarrow 0 < p \wedge \forall d(d \mid p \rightarrow d = 1 \vee 2 \mid d) \ .$$

By Σ_0-induction on x one easily proves there is always a power of two p above any number x:

$$\vdash \exists p(p \leq 2{\cdot}x + 1 \wedge x < p \wedge Pow_2(p)) \ . \tag{1}$$

We define the four place predicate $Sgm(x,p,q) \simeq s$ which Σ_0 in $I\Sigma_0$ and it is the graph of a partial function:

$$Sgm(x,p,q) \simeq s \leftrightarrow \exists a \exists b(x = a{\cdot}p + s{\cdot}q + b \wedge b < q \wedge s{\cdot}q + b < p)$$

because $I\Sigma_0$ proves:

$$\vdash 0 < p \wedge p < q \rightarrow \exists s Sgm(x,p,q) \simeq s \tag{2}$$
$$\vdash Sgm(x,p,q) \simeq s_1 \wedge Sgm(x,p,q) \simeq s_2 \rightarrow s_1 = s_2 \ . \tag{3}$$

We will use the predicate with p and q powers of two and then s is the segment in the binary representation of x between p and q.

pos.:	$x{+}1$	x	$x{-}1$	...	1	0
m:	1	1	1	...	1	1
a:	$*$	x	$x{-}1$	...	1	0
v:	$*$	$f(x)=$ $h(x{-}1, f(\tau[x{-}1]))$	$f(x{-}1)=$ $h(x{-}2, f(\tau[x{-}2]))$	...	$f(1)=$ $h(0, f(\tau[0]))$	$f(0)=$ g

Figure 1: Course of values sequences.

2.2 Course of values recursion. Suppose that we have extended PA with functions $g(\vec{y})$ and $h(x, \vec{y}, z_1, \ldots, z_k)$. Further suppose that the current extension proves for its terms $\tau_1, \ldots, \tau_k$ in the variables $x, \vec{y}$:

$$\vdash \tau_1 \leq x \wedge \cdots \wedge \tau_k \leq x \ . \tag{1}$$

We wish to extend by definition PA with the function symbol $f(x, \vec{y})$ defined by *course of values recursion* to satisfy the recurrences:

$$\vdash f(0,\vec{y}) = g(\vec{y}) \qquad \vdash f(x+1,\vec{y}) = h(x,\vec{y}, f(\tau_1,\vec{y}), \ldots, f(\tau_k,\vec{y})) \ . \tag{2}$$

Notice that *primitive recursion*:

$$\vdash f(0,\vec{y}) = g(\vec{y}) \qquad \vdash f(x+1,\vec{y}) = h(x,\vec{y}, f(x,\vec{y})) \tag{3}$$

is a special when $k = 1$ and $\tau_1 \equiv x$.

2.3 Course of values sequences. We encode the computation of f from recurrences 2.2(2) into three *course of values sequences* m, a, and v as shown in Fig. 1 where we have suppressed the parameters $\vec{y}$ and set $k := 1$, The sequence m contains $x{+}2$ single bits serving as *markers*. The purpose of markers is to delimit the values in the parallel sequences a and v. The *argument* sequence a contains the successive values of the recursive argument x and the corresponding values in the *value* sequence v are $f(x,\vec{y})$. The values designated by $*$ are immaterial.

In order to shorten the presentation we introduce two auxiliary Σ_0 in $I\Sigma_0$ predicates:

$$\begin{aligned}
&\mathit{Nbs}(m,p,q) \leftrightarrow \mathit{Pow}_2(p) \land \mathit{Pow}_2(q) \land \mathit{Sgm}(m,2{\cdot}p,p) \simeq 1 \land \mathit{Sgm}(m,p,q) \simeq 1\\
&\mathit{In}(m,a,v,x,z) \leftrightarrow \exists p \exists q(\mathit{Nbs}(m,p,q) \land \mathit{Sgm}(a,p,q) \simeq x \land \mathit{Sgm}(v,p,q) \simeq z) \ .
\end{aligned}$$

$\mathit{Nbs}(m,p,q)$ holds iff p and q are two neighboring markers in m and the predicate $\mathit{In}(m,a,v,x,z)$ iff for two neighboring markers in m the pair x and z is in the corresponding sequences a and v.

The predicate $\mathit{Sqs}(m,a,v,\vec{y})$ constraining the course of values sequences is defined as follows:

$$\begin{aligned}
&\mathit{Sqs}(m,a,v,\vec{y}) \leftrightarrow \forall p \forall q(\mathit{Nbs}(m,p,q) \rightarrow\\
&\quad q = 1 \land \mathit{Sgm}(a,p,q) \simeq 0 \land \mathit{Sgm}(v,p,q) \simeq g(\vec{y}) \lor\\
&\quad \exists x \exists z_1 \ldots \exists z_k(\mathit{Sgm}(a,p,q) \simeq x{+}1 \land \mathit{Sgm}(v,p,q) = h(x,\vec{y},z_1,\ldots,z_k) \land\\
&\qquad \mathit{In}(q+m \bmod q,a,v,\tau_1,z_1) \land \cdots \land \mathit{In}(q+m \bmod q,a,v,\tau_k,z_k))) \ .
\end{aligned}$$

The predicate is $\Sigma_0(\vec{g})$ in $I\Sigma_0(\vec{g})$ where $\vec{g}$ contains g, h, and all introduced functions applied in $\tau_1, \ldots, \tau_k$. Such is also the following auxiliary predicate:

$$\mathit{Sqsw}(m,a,v,\vec{y},x,z) \leftrightarrow \mathit{Sqs}(m,a,v,\vec{y}) \land \mathit{In}(m,a,v,x,z) \ .$$

2.4 Closure of $I\Sigma_1(\vec{g})$ under course of values recursion. The function f defined by course of values recursion as in Par. 2.2 is introduced into PA by:

$$f(x,\vec{y}) = z \leftrightarrow \exists m \exists a \exists v\, \mathit{Sqsw}(m,a,v,\vec{y},x,z) \ . \tag{1}$$

The formula on the right-hand-side is $\Sigma_1(\vec{g})$ where $\vec{g}$ are as in Par. 2.3. We outline in the following the proof in $I\Sigma_1(\vec{g})$ of the uniqueness (cf. 2.5(4)) and existence (cf. 2.6(3)) conditions as well as of the recurrences 2.2(3) (cf. Par. 2.7).

2.5 Uniqueness properties of *Sqs*.

$$\vdash \mathit{Sqsw}(m,a,v,\vec{y},0,z) \rightarrow z = g(\vec{y}) \tag{1}$$

$$\begin{aligned}
\vdash \mathit{Sqsw}(m,a,v,\vec{y},x+1,z) \rightarrow \exists z_1 \ldots \exists z_k(z = h(x,\vec{y},z_1,\ldots,z_k) \land&\\
\mathit{In}(m,a,v,\tau_1,z_1) \land \cdots \land \mathit{In}(m,a,v,\tau_k,z_k))&
\end{aligned} \tag{2}$$

$$\vdash \mathit{Sqsw}(m_1,a_1,v_1,\vec{y},x,z_1) \land \mathit{Sqsw}(m_2,a_2,v_2,\vec{y},x,z_2) \rightarrow z_1 = z_2 \ . \tag{3}$$

(1): We work in $\Sigma_0(\vec{g})$ where from the assumption we have $\mathit{Sqs}(m,a,v,\vec{y})$, $\mathit{Nbs}(m,p,q)$, $\mathit{Sgm}(a,p,q) \simeq 0$, and $\mathit{Sgm}(v,p,q) \simeq z$. It must be the case that $q = 1$ and so $z = g(\vec{y})$.
(2): We work in $\Sigma_0(\vec{g})$ where from the assumption we have $\mathit{Sqs}(m,a,v,\vec{y})$, $\mathit{Nbs}(m,p,q)$, $\mathit{Sgm}(a,p,q) \simeq x+1$, and $\mathit{Sgm}(v,p,q) \simeq z$. It cannot be the case

that $q = 1$ and so $In(m,a,v,\tau_1,z_1)$, ..., $In(m,a,v,\tau_k,z_k)$ for some $z_1,\dots,z_k$ such that $z = h(x,\vec{y},z_1,\dots,z_k)$.
(3): By $\Sigma_0(\vec{g})$-induction on x. In the base case we have $z_1 = g(\vec{y}) = z_2$ by (1). In the inductive case we have by (2) $z_1 = h(x,\vec{y},u_1,\dots,u_k)$ and $z_2 = h(x,\vec{y},w_1,\dots,w_k)$ for some $u_1,w_1,\dots,u_k,w_k$ such that $In(m_1,a_1,v_1,\tau_1,u_1)$, ..., $In(m,a,v,\tau_k,u_k)$, $In(m_2,a_2,v_2,\tau_1,w_1)$, ..., $In(m_2,a_2,v_2,\tau_k,w_k)$. But then $u_1 = w_1$, ..., $u_k = w_k$ by IH.

The Uniqueness condition for 2.4(1):

$$\vdash \exists m\exists a\exists v Sqsw(m,a,v,\vec{y},x,z_1) \land \exists m\exists a\exists v Sqsw(m,a,v,\vec{y},x,z_2) \to z_1 = z_1 \quad (4)$$

follows directly from the assumptions by (3).

2.6 Extension properties of Sqs. $I\Sigma_0(\vec{g})$ proves the following extension properties for $Sqsw$:

$$\vdash \exists m\exists a\exists v Sqsw(m,a,v,\vec{y},0,g(\vec{y})) \quad (1)$$

$$\begin{aligned}&\vdash Sqsw(m,a,v,\vec{y},x,z) \to \exists m\exists a\exists v\exists z_1\dots\exists z_k(\\ &\quad Sqsw(m,a,v,\vec{y},x+1,h(x,\vec{y},z_1,\dots,z_k)) \land\\ &\quad In(m,a,v,\tau_1,z_1) \land \dots \land In(m,a,v,\tau_k,z_k))\ . \end{aligned} \quad (2)$$

(1): Let p be a power of two s.t. $p > \max(1,g(\vec{y}))$. We then have

$$Sqsw(p+1,0,g(\vec{y}),\vec{y},0,g(\vec{y}))\ .$$

(2): We have $Sqs(m,a,v,\vec{y})$, $Nbs(m,q,r)$, $Sgm(a,q,r) \simeq x$, and $Sgm(v,q,r) \simeq z$ for some q,r. We consider two cases. If $In(m,a,v,x+1,w)$ for some w then the property follows from from 2.5(2). Otherwise q must be the highest power in m and, since $\tau_1,\dots,\tau_k \le x$, for $a_1 = a \bmod q$, and $v_1 = v \bmod q$ we have $In(m,a_1,v_1,\tau_1,z_1)$, ..., $In(m,a_1,v_1,\tau_k,z_k)$ for some z_1, ..., z_k. For a power of two $p > \max((x+1)\cdot q, h(x,\vec{y},z_1,\dots,z_k)\cdot q)$ we then get:

$$Sqsw(p+m,(x+1)\cdot q+a_1,h(x,\vec{y},z_1,\dots,z_k)\cdot q+v_1,\vec{y},x+1,h(x,\vec{y},\tau_1,\dots,\tau_k))\ .$$

By $\Sigma_1(\vec{g})$-induction on x we now prove the Existence condition for 2.4(1):

$$\vdash \exists z\exists m\exists a\exists v Sqsw(m,a,v,\vec{y},x,z)\ . \quad (3)$$

In the base case we use (1). In the inductive case we have $Sqsw(m,a,v,\vec{y},x,z)$ for some z,m,a,v from IH and we use (2) .

2.7 Proof of recurrences for f. We can now prove in $I\Sigma_1(\vec{g})$ the recurrences 2.2(3) for f. We have $Sqsw(m,a,v,\vec{y},0,f(0,\vec{y}))$ for some m,a,v from the definition and $f(0,\vec{y}) = g(\vec{y})$ by 2.5(1).

Similarly, we have $Sqsw(m,a,v,\vec{y},x{+}1,f(x{+}1,\vec{y}))$ for some m,a,v from the definition and $f(x{+}1,\vec{y}) = h(x,\vec{y},z_,\dots,z_k)$ for some $z_1,\dots,z_k$ such that $In(m,a,v,\tau_1,z_1)$, ..., $In(m,a,v,\tau_k,z_k)$ by 2.5(2). Since

$$Sqsw(m_1,a_1,v_1,\tau_1,f(\tau_1,\vec{y})) \land \dots \land Sqsw(m_k,a_k,v_k,\tau_k,f(\tau_k,\vec{y}))$$

for some $m_1,a_1,v_1,\dots,m_k,a_k,v_k$ from the definition of f, we get $z_1 = f(\tau_1,\vec{y})$, ..., $z_k = f(\tau_k,\vec{y})$ by 2.5(3).

3 A Simple Σ_0-Derivation of $2^x = y$

The fragment $I\Sigma_0$ is not closed under primitive recursion but we can do a surprising amount of coding in it (see [HP93]) once we introduce the graph $2^x = y$ of exponentiation which we write as a partial function $2^x \simeq y$.

3.1 Computing 2^x by recursion on notation. We cannot bound in $I\Sigma_0$ the primitive recursive sequences for 2^x as in Fig. 1. However, when we express 2^x by a much faster *recursion on notation*, which is known to computer programmers as the *logarithmic program* for 2^x:

$$2^0 = 1 \qquad 2^{2\cdot x+1} = 2\cdot(2^x)^2 \qquad x > 0 \to 2^{2\cdot x+0} = (2^x)^2 \ ,$$

then we succeed in bounding the sequences for the graph $2^x \simeq y$ by a polynomial in x, y. We can do even better. By constraining the marker sequence m it can also play the role of the value sequence v.

If $2^x = y$ then there is a finite course of values sequence of powers of two $p_n, p_{n-1}, \dots, p_0$ which are needed in the above computation of 2^x. The sequence is such that $p_n = y = 2^x$, $p_0 = 1 = 2^0$, and for all $n > i \geq 0$ we have $p_{i+1} = (k_i{+}1)\cdot p_i^2$ for a unique $k_i = 0, 1$. The numbers x_i such that $2^{x_i} = p_i$ satisfy $x_n = x$, $x_0 = 0$, and $x_{i+1} = 2\cdot x_i{+}k_i$ for all $n > i \geq 0$.

For some $m < y$ we thus have $y + m = \sum_{i \leq n} p_i = \sum_{i \leq n} 2^{x_i}$ and for some $a < y$ we have $x\cdot y + a = \sum_{i \leq n} x_i\cdot p_i = \sum_{i \leq n} x_i\cdot 2^{x_i}$. This suggests a simple arithmetical (as opposed to metamathematical) encoding of the course of values sequences for the powers p_i and exponents x_i. Figure 2 shows the course of values sequences for the computation of 2^{14}.

bitno.	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$y{+}m$	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1
$x\cdot y{+}a$	1	1	1	0	0	0	0	0	1	1	1	0	0	1	1	0	1	0

Figure 2: Course of values sequences $y{+}m$ and $x\cdot y{+}a$ encoding the computation of $2^x = y$ for $x = 14$ and $y = 16,384$.

3.2 Graph of exponentiation $2^x \simeq y$ is a Σ_0-predicate. The following Σ_0-predicate $Seqs(m, a)$ holds iff $m = \sum_{i \leq n} 2^{x_i}$ and $a = \sum_{i \leq n} x_i\cdot 2^{x_i}$ for some n such that $x_0 = 0$ and for all $i < n$ there is a $k = 0, 1$ s.t. $p_{i+1} = (k{+}1)\cdot p_i^2$ and $x_{i+1} = 2\cdot x_i{+}k$:

$$\begin{aligned} &Seqs(m,a) \leftrightarrow \forall p(Pow_2(p) \land Sgm(m, 2\cdot p, p) \simeq 1 \to p = 1 \land 2 \not| a \lor \exists q \exists k{\leq}1 \exists x(\\ &\ p = (k{+}1)\cdot q^2 \land Sgm(m,p,q) \simeq 1 \land Sgm(a,p,q) \simeq x \land Sgm(a,p^2,p) \simeq 2\cdot x{+}k)) \ . \end{aligned}$$

Exponentiation 2^x is now defined as a Σ_0-partial function by:

$$2^x \simeq y \leftrightarrow x < y \land Pow_2(y) \land \exists m{<}y \exists a{<}y\ Seqs(y + m, x\cdot y + a) \ .$$

$I\Sigma_0$ easily proves first the recurrences

$$\vdash 2^0 \simeq 1 \qquad \vdash x > 0 \wedge 2^x \simeq y \to 2^{2\cdot x} \simeq y^2$$

and then the primitive recursive one:

$$\vdash 2^x \simeq y \to 2^{x+1} \simeq 2 \cdot y \ .$$

4 Coding in Arithmetic

We investigate the coding in PA by modern coding techniques of computer science which were originally inspired by the programming language LISP. The methods are used in declarative programming languages such as Haskell [Tho99] (albeit not in the domain $\mathbb{N}$) or directly in PA by the language CL.

4.1 A suitable pairing function. We can can obtain a one-to-one correspondence between finite sequences over $\mathbb{N}$ and $\mathbb{N}$ by introducing into PA a binary *pairing* function designated by x, y and satisfying:

$$\vdash x, y = v, w \to x = v \wedge y = w \tag{1}$$
$$\vdash x < x, y \wedge y < x, y \tag{2}$$
$$\vdash x = 0 \vee \exists v \exists w \, x = v, w \ . \tag{3}$$

(1) is the usual *pairing property*, property (2) is needed so we can do induction over pairs. We also get from it that 0 is an *atom*, i.e. $0 \neq x, y$, because otherwise we would get $x < 0$. That 0 is the only atom follows from (3).

We can easily see that every every natural number is then uniquely denoted in $\mathcal{N}$ by a *pair numeral* which is a term formed from 0 by pairing (τ_1, τ_2). If we are interested in characterization of small classes of computational complexity (such as PTIME) we should require that the number of symbols needed to write down the pair numeral denoting x be on the order of $\log(x)$.

It is well-known that the Cantor's diagonal function satisfies the first pairing property but, since $0, 0 = 0$, there are no atoms. If we increase the function by one, i.e. if we introduce into $I\Sigma_0$ the *modified Cantor's function*, by:

$$x, y = (x + y) \cdot (x + y + 1) \div 2 + x + 1 \ , \tag{4}$$

then the three above pairing properties are provable there. It was shown in [Vo94] that the modified Cantor's function fails the size requirement. A suitable pairing function was introduced there. The function is also discussed at length in [Vo01]. However, the modified Cantor's function can be used as long as we are only interested in primitive recursive functions (and more). Since this is the case in this paper, we fix the pairing function (4).

4.2 Pairing conventions. We adopt a pairing convention such that whenever the infix pairing comma in (x, y) cannot be mistaken for a comma separating arguments we may drop the parentheses around pairing. Thus, for instance, if f is a binary function then $f((v,w),x,y)$ abbreviates $f((v,w),(x,y))$. Notice that the first set of parentheses cannot be omitted. We let pairing group to the right where x,y,z abbreviates $x,(y,z)$. We also assign to the pairing function the lowest precedence. Thus, for instance, $x,y+z$ stands for $x,(y+z)$.

4.3 Coding of finite sequences. Any pairing function satisfying the three pairing properties from Par. 4.1 establishes a one-to-one correspondence between finite sequences over $\mathbb{N}$ and $\mathbb{N}$. The empty sequence is coded by 0 and $a_1 \dots a_n$ by $a_1,\dots,a_n,0$. Notice the use of 0 to recognize the end of the code. This is not possible without atoms. Codes of finite sequences are called *lists* in computer programming where they were introduced in the late nineteen fifties by J. McCarthy in his path-breaking programming language LISP. Our representation in $\mathbb{N}$ of lists reduces without loss of power the denumerable infinitude of LISP's atoms to the single one: 0. As a consequence every natural number is a unique list.

4.4 Projection functions. We wish to introduce the unary *projection* functions H (head) and T (tail) satisfying:

$$\vdash H(0)=0 \qquad \vdash H(v,w)=v \tag{1}$$

$$\vdash T(0)=0 \qquad \vdash T(v,w)=w\ . \tag{2}$$

Notice that $H(v,w)$ is an abbreviation for $H((v,w)$.

In the view of pairing properties the functions are $\Sigma_0(,)$-definable:

$$H(x)=v \leftrightarrow x=0 \land v=0 \lor \exists w\, x=v,w$$
$$T(x)=w \leftrightarrow x=0 \land w=0 \lor \exists v\, x=v,w\ .$$

$I\Sigma_0$ proves the above properties of projection functions.

4.5 Schema of pair recursion. The schema of *pair recursion* defines f from functions g and h and terms $\tau_1, \dots \tau_k$ such that

$$(\tau_1 \le x \lor \tau_1 \le y) \land \cdots \land (\tau_k \le x \lor \tau_k \le y) \tag{1}$$

and f satisfies:

$$f(0,\vec{z}) = g(\vec{z}) \qquad f((x,y),\vec{z}) = h(x,y,\vec{z},f(\tau_1,\vec{z}),\dots,f(\tau_k,\vec{z}))\ .$$

$I\Sigma_1$ is closed under pair recursion, i.e it proves the recurrences for f whenever the condition (1) is provable. This is because pair recursion is reducible to course of values recursion since $x,y=z+1$ for some z, and if $a \le x$ or $a \le y$ then $a \le z$.

4.6 List indexing. We wish to introduce a binary *list indexing* function $(x)_i$ selecting the i-th element a_i of the list $x = a_0,a_1,\dots,a_{n-1},0$ provided $i < n$.

The function should satisfy:

$$\vdash (0)_i = 0 \qquad \vdash (v,w)_0 = v \qquad \vdash (v,w)_{i+1} = (w)_i \ . \tag{1}$$

Notice that, although the recursion in $(x)_i$ is primitive recursion in the variable i and pair recursion in the variable x, the recurrences are with *substitution in parameters* where the parameters are not kept constant in recursive applications.

To surmount this complication we introduce into $I\Sigma_1$ the binary function $T^i(x)$ iterating the tail function by primitive recursion such that:

$$\vdash T^0(x) = x \qquad \vdash T^{i+1}(x) = T\,T^i(x) \ . \tag{2}$$

$I\Sigma_1$ then proves:

$$\vdash T^{i+1}(x) = T^i T(x) \tag{3}$$

$$\vdash \exists i\, T^i(x) = 0 \tag{4}$$

$$\vdash T^i(0) = 0 \ . \tag{5}$$

(3): By induction on i. In the base case we have $T^{0+1}(x) = T\,T^0(x) = T(x) = T^0 T(x)$. In the inductive case we have

$$T^{i+1+1}(x) = T\,T^{i+1}(x) \overset{IH}{=} T\,T^i T(x) = T^{i+1}T(x) \ .$$

(4): By complete induction on x. We consider two cases by 4.1(3). If $x = 0$ we have $T^0(0) = 0$. If $x = v, w$ for some v and w we have $w < x$ by 4.1(2). Thus $T^i(w) = 0$ for some i by IH and so $T^{i+1}(v,w) \overset{(3)}{=} T^i T(v,w) = T^i(w) = 0$.

(5): By induction on i. In the base case we have $T^0(0) = 0$ and in the inductive case $T^{i+1}(0) = T\,T^i(0) \overset{IH}{=} T(0) = 0$.

We now explicitly introduce the list indexing function to satisfy:

$$(x)_i = H\,T^i(x) \ . \tag{6}$$

We prove (1) as follows: $(0)_i = H\,T^i(0) \overset{(5)}{=} H(0) = 0$; $(v,w)_0 = H\,T^0(v,w) = H(v,w) = v$; and

$$(v,w)_{i+1} = H\,T^{i+1}(v,w) \overset{(3)}{=} H\,T^i T(v,w) = H\,T^i(w) = (w)_i \ .$$

4.7 Basic list processing operations. In addition to list indexing $(x)_i$, there are three basic *list processing* operations. The list $x = a_0, a_1, \ldots, a_{n-1}, 0$ has the *length* n, which is yielded by the unary *length* function $L(x) = n$. For any (list) y the list $a_0, a_1, \ldots, a_{n-1}, y$ is yielded by the binary *list concatenation* function $x \oplus y$. A number a is an *element* of x, in writing $a \in x$, iff $a = a_i$ for some $i < n$.

Both functions and the *list membership* predicate $a \in x$ are introduced into $I\Sigma_1$ by pair recursion to satisfy:

$$\vdash L(0) = 0 \qquad \vdash L(x,y) = L(y) + 1 \tag{1}$$

$$\vdash 0 \oplus z = 0 \qquad \vdash (x,y) \oplus z = x, y \oplus z \tag{2}$$

$$\vdash a \notin 0 \qquad \vdash a \in (x,y) \leftrightarrow a = x \lor a \in y \ . \tag{3}$$

Notice that the list membership is introduced as $\vdash a \in x \leftrightarrow (a \in_* x) = 1$ through its characteristic function:

$$\vdash (a \in_* 0) = 0 \qquad \vdash (a \in_* x, y) = D((a =_* x), 1, (a \in_* y))$$

where the ternary *case analysis* function D satisfies:

$$\vdash D(0, y, z) = z \qquad \vdash D(x+1, y, z) = y$$

and $x =_* y$ is the characteristic function of identity. $I\Sigma_1$ then proves by complete induction:

$$\begin{array}{ll} \vdash x \oplus 0 = x & \vdash (x \oplus y) \oplus z = x \oplus (y \oplus z) \\ \vdash L(x \oplus y) = L(x) + L(y) & \vdash a \in x \oplus y \leftrightarrow a \in x \vee a \in y \\ \vdash a \in x \leftrightarrow \exists i{<}L(x)\,(x)_i = a & \vdash i \geq L(x) \rightarrow (x)_i = 0\ . \end{array}$$

4.8 Example of arithmetization of syntax. We will now show how to arithmetize syntactic objects. We will arithmetize certain terms which will be needed in Sect. 5. Such terms are constructed from variables x_1, x_2, ... and numerals $\overline{n}$ by pairing, and by applications of function symbols f, g_1, ..., g_k of arities n, n_1, ..., n_k respectively.

We will code the terms with the help of - what is in computer science called - *constructors*. They are the following explicitly defined functions: $\mathbf{x}_i = 0, i$; $\mathbf{N}(n) = 1, n$; $\langle a, b \rangle = 2, a, b$; $\mathbf{g}_i(a) = 3, i, a$; and $\mathbf{f}(a) = 4, a$.

We abbreviate $\langle x, \langle y, z \rangle\rangle$ to $\langle x, y, z \rangle$ The constructors are used to assign the codes, denoted by terms $\ulcorner \tau \urcorner$, to the terms τ to satisfy:

$$\begin{gathered} \ulcorner x_{i+1} \urcorner \equiv \mathbf{x}_{\overline{i}} \qquad \ulcorner \overline{n} \urcorner \equiv \mathbf{N}(\overline{n}) \qquad \ulcorner \tau_1, \tau_2 \urcorner \equiv \langle \ulcorner \tau_1 \urcorner, \ulcorner \tau_2 \urcorner \rangle \\ \ulcorner f(\tau_1, \ldots, \tau_n) \urcorner = \mathbf{f}(\langle \ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_n \urcorner, \mathbf{N}(0) \rangle) \\ \ulcorner g_{i+1}(\tau_1, \ldots, \tau_{n_{i+1}}) \urcorner = \mathbf{g}_{\overline{i}}(\langle \ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_{n_{i+1}} \urcorner, \mathbf{N}(0) \rangle)\ . \end{gathered}$$

Notice that we have

$$\begin{gathered} \ulcorner f(\tau_1, \ldots, \tau_n) \urcorner \equiv \mathbf{f}(\ulcorner \tau_1, \ldots, \tau_n, 0 \urcorner) \\ \ulcorner g_{i+1}(\tau_1, \ldots, \tau_{n_{i+1}}) \urcorner \equiv \mathbf{g}_{\overline{i}}(\ulcorner \tau_1, \ldots, \tau_{n_{i+1}}, 0 \urcorner)\ . \end{gathered}$$

The predicate $\mathit{Trm}(c)$ holding of codes c of terms is introduced into the declarative programming languages by a *union type* (a class of data structures):

$$\mathit{Trm} = \mathbf{x}_{\mathbb{N}} \mid \mathbf{N}(\mathbb{N}) \mid \langle \mathit{Trm}, \mathit{Trm} \rangle \mid \mathbf{f}(\mathit{Trm}) \mid \mathbf{g}_{\mathbb{N}}(\mathit{Trm})$$

and into PA from its characteristic function defined by pair recursion:

$$\begin{aligned} &\vdash \mathit{Trm}_*(0) = 0 \\ &\vdash \mathit{Trm}_*(x, y) = D((x{=_*}0) + (x{=_*}1), 1, \\ &\qquad\qquad D(x{=_*}2, \mathit{Trm}_* H(y) \cdot \mathit{Trm}_* T(y), \\ &\qquad\qquad\quad D(x{=_*}3, \mathit{Trm}_* T(y), D(x{=_*}4, \mathit{Trm}_*(y), 0)))\ . \end{aligned}$$

If the reader is not impressed by this almost unreadable definition he should compare it with the definition in CL given in Fig. 7.

PA easily proves properties of codes of terms such as

$$\vdash \mathit{Trm}(\mathbf{x}_i) \quad \vdash \mathit{Trm}(\langle a,b\rangle) \leftrightarrow \mathit{Trm}(a) \land \mathit{Trm}(b) \quad \vdash \mathit{Trm}(\mathbf{g}_i(a)) \leftrightarrow \mathit{Trm}(a) \ .$$

5 A Strong Schema of Primitive Recursion

The schema of course of values recursion suffices for the arithmetization of syntax as needed in the proof of both Incompleteness Theorems of Gödel. Thus logicians have no motivation to look into stronger schemas yielding primitive recursive functions.

When we wish to explicate in PA a programming language such as CL we have to go further because, powerful as it is, course of values recursion is not very comfortable to program in. The problem is mainly in the need to keep the parameters constant because most programs change them in recursive applications. To a lesser extent one also needs the nesting of recursive applications.

5.1 Course of values recursion with measure. The most general schema of recursion which does not lead outside primitive recursion is *course of values recursion with measure*:

$$f(\vec{x}) = \tau[\vec{x}] \tag{1}$$

where τ is a term in the language of the current extension of PA with variables from $\vec{x}$ and applying functions $g_1, \ldots, g_k$. The term may also contain recursive applications of f with arbitrary arguments possibly containing nested recursive applications but always in a context:

$$D(\mu[\vec{\rho}] <_* \mu[\vec{x}], f(\vec{\rho}), 0)$$

where the application $f(\vec{\rho})$ is *guarded* by the *measure* term $\mu[\vec{x}]$ which does not apply f but may apply $g_1, \ldots, g_k$.

We have investigated this schema, which can be traced to the work of Péter [Pet32], in [KV98]. We have shown there primitive recursive functions closed under the schema. We will now formalize this in $I\Sigma_1$. This stronger result is crucial for the characterization of the programming language CL. For the rest of this section we will work in $I\Sigma_1$ by assuming that the functions $g_1, \ldots, g_k$ have been previously introduced there.

5.2 Computation trees. We will introduce a **unary** predicate $\mathit{Ct}(t)$ intended to hold of t when t encodes the computation tree for the evaluation of f defined by course of values recursion $f(\vec{x}) = \tau[\vec{x}]$ with measure $\mu[\vec{x}]$. The tree t consists of nodes of the form a, x, y, s where a codes a subterm ρ of τ, x is an *assignment* giving the values of its variables $\vec{x}$, y is the value (denotation) of ρ in the assignment x, and s codes the subtrees needed in the evaluation of ρ.

$$
\begin{array}{l}
\vdash a[x] \simeq y,t \leftrightarrow \\
\quad \exists i\,(a = \mathbf{x}_i \land y = (x)_i \land t = 0) \lor a = \mathbf{N}(y) \land t = 0 \lor \\
\quad \exists a_1 \exists a_2 \exists y_1 \exists y_2 \exists t_1 \exists t_2 (a = \langle a_1, a_2 \rangle \land y = y_1, y_2 \land \\
\qquad t = (a_1, x, y_1, t_1), (a_2, x, y_2, t_2) \land a_1[x] \simeq y_1, t_1 \land a_2[x] \simeq y_2, t_2) \lor \\
\quad \exists a_1 \exists i \exists y_1 \exists t_1 (a = \mathbf{g}_i(a_1) \land t = a, x, y_1, t_1 \land a[x] \simeq y_1, t_1 \land \\
\qquad (i = 0 \land y = g_1((y_1)_0, \ldots, (y_1)_{\overline{n_1 - 1}}) \lor \cdots \lor \\
\qquad\; i = \overline{k{-}1} \land y = g_k((y_1)_0, \ldots, (y_1)_{\overline{n_k - 1}}))) \lor \\
\quad \exists a_1 \exists y_1 \exists t_1 \exists t_2 (a = \mathbf{f}(a_1) \land \mu[(y_1)_0, \ldots, (y_1)_{\overline{n-1}}] < \mu[(x)_0, \ldots, (x)_{\overline{n-1}}] \land \\
\qquad t = (a_1, x, y_1, t_1), (\ulcorner \tau \urcorner, y_1, y, t_2) \land a_1[x] \simeq y_1, t_1 \land \ulcorner \tau \urcorner [y_1] \simeq y, t_2) \lor \\
\quad \exists a_1 \exists y_1 \exists t_1 (a = \mathbf{f}(a_1) \land y = 0 \land \mu[(y_1)_0, \ldots, (y_1)_{\overline{n-1}}] \geq \mu[(x)_0, \ldots, (x)_{\overline{n-1}}] \land \\
\qquad t = a_1, x, y_1, t_1 \land a_1[x] \simeq y_1, t_1)
\end{array}
$$

Figure 3: Computation trees for course of values recursion with measure.

The term τ is coded as in Par. 4.8 except that we ignore the guards around f and assign the codes to the guarded subterms to satisfy:

$$\ulcorner D(\mu[\vec{\rho}] <_* \mu[\vec{x}], f(\vec{\rho}) \urcorner \equiv \ulcorner f(\vec{\rho}) \urcorner \equiv \mathbf{f}(\ulcorner \rho_1, \ldots, \rho_n, 0 \urcorner) \tag{1}$$

The initial term to be evaluated is τ, i.e. $a = \ulcorner \tau \urcorner$, in the assignment given by the list $x_1, x_2, \ldots, x_n, 0$ corresponding to the arguments of $f(\vec{x})$.

The predicate Ct is introduced into $I\Sigma_1$ by pair recursion (through its characteristic function Ct_*) to satisfy:

$$\vdash \neg Ct(0) \qquad \vdash \neg Ct(a, 0) \qquad \vdash \neg Ct(a, x, 0)$$

and the equivalence given in Fig. 3 where we abbreviate $Ct(a, x, y, t)$ by $a[x] \simeq y, t$. The abbreviation is intended to suggest that we are evaluating the term coded by a in the assignment x with the denotation of the term being y and with t coding the subcomputations. The choice of this particular encoding of computation trees was dictated by the need for the recursive applications $Ct(s)$ to be to the subtrees s of t for which we have on accord of 4.1(2) $s < t$.

Notice the evaluation of recursive applications of f in Fig. 3. First, the code a_1 of the argument is evaluated into y_1 in the assignment x, and then the measure test $\mu[(y_1)_0, \ldots, (y_1)_{\overline{n-1}}] < \mu[(x)_0, \ldots, (x)_{\overline{n-1}}]$ is performed. If the measure decreases then the code of τ is evaluated in the assignment y_1 to yield y. Otherwise the evaluation yields 0.

We trust the reader with the definition of Ct_* by pair recursion where he should replace the propositional connectives in Fig. 3 by the case analysis function D and the existential quantifiers by projections H and T.

5.3 Closure of $I\Sigma_1$ under course of values recursion with measure. The partial function notation for computation trees is justified because we will show in Par. 5.4 and Par. 5.5 that $I\Sigma_1$ proves:

$$\vdash \exists y \exists t \ulcorner \tau \urcorner[x] \simeq y, t \tag{1}$$

$$\vdash a[x] \simeq y_1, t_1 \wedge a[x] \simeq y_2, t_2 \to y_1 = y_2 \wedge t_1 = t_2 \tag{2}$$

respectively. The properties trivially imply the Existence and Uniqueness conditions for the introduction of f by a Δ_1-formula:

$$f(x_1, \ldots, x_n) = y \leftrightarrow \exists t \ulcorner \tau \urcorner[x_1, \ldots, x_n, 0] \simeq y, t\ . \tag{3}$$

We will show the recurrence 5.1(1) in Par. 5.6

5.4 Existence properties of computation trees. If ρ is any term constructed similarly as τ from Par. 5.1 then we prove

$$\begin{aligned} \vdash \forall v(\mu[(v)_0, \ldots, (v)_{\overline{n-1}}] < \mu[(x)_0, \ldots, (x)_{\overline{n-1}}] \to \exists y \exists t \ulcorner \tau \urcorner[v] \simeq y, t) \\ \to \exists y \exists t \ulcorner \rho \urcorner[x] \simeq y, t \end{aligned} \tag{1}$$

by (metatheoretical) induction on the structure of ρ. The proof is quite straightforward except that we have to take into account the way we have assigned the codes to applications $h(\rho_1, \ldots, \rho_m)$ where h is f or $g_1, \ldots, g_k$. For the purposes of this proof we should treat h as if it were a unary function symbol applied to a pair, i.e. $h(\rho_1, \ldots, \rho_m, 0)$.

We also treat the applications of f in ρ together with their guards. Thus if

$$\rho \equiv D(\mu[\rho_1, \ldots, \rho_n] <_* \mu[x_1, \ldots, x_n], f(\rho_1, \ldots, \rho_n), 0) \tag{2}$$

then we have $\ulcorner \rho_1, \ldots, \rho_n, 0 \urcorner[x] \simeq y_1, t_1$ for some y_1, t_1 by IH and we consider two cases. If $\mu[(y_1)_0, \ldots, (y_1)_{\overline{n-1}}] < \mu[(x)_0, \ldots, (x)_{\overline{n-1}}]$ we have $\ulcorner \tau \urcorner[y_1] \simeq y, t_2$ for some y and t_2 from the assumption of (1). Because of 5.2(1) we then get $\ulcorner \rho \urcorner[x] \simeq y, t$ for

$$t = (\ulcorner \rho_1, \ldots, \rho_n, 0 \urcorner, x, y_1, t_1), (\ulcorner \tau \urcorner, y_1, y, t_2)\ .$$

If the measure test fails then we get $\ulcorner \rho \urcorner[x] \simeq y, t$ for $y = 0$ and

$$t = \ulcorner \rho_1, \ldots, \rho_n, 0 \urcorner, x, y_1, t_1\ .$$

We now prove:

$$\vdash \forall x(\mu[(x)_0, \ldots, (x)_{\overline{n-1}}] < 0 \to \exists y \exists t \ulcorner \tau \urcorner[x] \simeq y, t) \tag{3}$$

$$\vdash \forall x(\mu[(x)_0, \ldots, (x)_{\overline{n-1}}] < z \to \exists y \exists t \ulcorner \tau \urcorner[x] \simeq y, t) \tag{4}$$

$$\to \forall x(\mu[(x)_0, \ldots, (x)_{\overline{n-1}}] < z{+}1 \to \exists y \exists t \ulcorner \tau \urcorner[x] \simeq y, t) \tag{5}$$

where (3) is trivial and for the proof of (5) we assume the antecedent, take any x such that $\mu[(x)_0, \ldots, (x)_{\overline{n-1}}] < z{+}1$, and consider two cases. If

$$\mu[(x)_0, \ldots, (x)_{\overline{n-1}}] < z$$

then the conclusion holds from the assumption. If $\mu[(x)_0,\ldots,(x)_{\overline{n-1}}] = z$ then we use the assumption in (1) for $\rho := \tau$ to get the conclusion again.

All it remains is to use the just proved properties in the proof of

$$\vdash \forall x(\mu[(x)_0,\ldots,(x)_{\overline{n-1}}] < z \to \exists y \exists t \ulcorner\tau\urcorner[x] \simeq y, t) \tag{6}$$

by induction on z. We can then set $z := \mu[(x)_0,\ldots,(x)_{\overline{n-1}}] + 1$ to conclude 5.3(1).

The only problem is that we have proved 5.3(1) in $I\Sigma_2$ because we have used the Π_2-induction axiom for the formula (6). Fortunately, we can use a relatively unknown result of Parsons [Par72] (see also [Bus94, KV98]) which, specialized to Π_2-formulas, says that these are theorems of $I\Sigma_1$ iff they are theorems of $IR\Pi_2$. The latter theory has the induction axioms replaced by induction rules for Π_2-formulas $\phi[x]$:

$$\frac{\phi[0] \qquad \forall x(\phi[x] \to \phi[x+1])}{\phi[x]}\,.$$

Notice that although Π_2-induction rules permit stronger induction formulas than Σ_1-induction axioms, they are without any side formulas, i.e. the deduction theorem is not applicable (except for Π_1-formulas which can be incorporated into the induction formula ϕ).

We can thus prove the Π_2-formula in (6) by a Π_2-rule with (3) and (5) as premises. The second premise is proved from the schema of theorems (1) without induction. Note that the metainduction needed for the proof of (1) does not count because the last is needed for $\rho := \tau$ and the proofs of the finitely many instances of (1) with ρ a subterm of τ are directly included in the proof of (5). The proof of (5) is thus without induction from the equivalence in Fig. 3 and other theorems concerned with the introduction of functions $g_1, \ldots, g_k$. The equivalence and the other formulas are all Σ_1 in $I\Sigma_1$. and, by being provable in $I\Sigma_1$, they are provable in $IR\Pi_2$ by the theorem of Parson. Hence (3) and (5) are provable in $IR\Pi_2$ and so is (6). The detour in the proof of the Π_2-formula (6) through the Σ_3-formula (5) is then eliminated by the same theorem and consequently (6) is provable in $I\Sigma_1$.

If the reader has doubts whether the formula in Fig. 3 is Σ_1 because it is of a form $(\phi \to \psi) \wedge (\psi \to \phi)$ with both ϕ and ψ Σ_1 in $I\Sigma_1$, he should recall that the formulas are actually Δ_1 in $I\Sigma_1$.

5.5 Uniqueness properties of Computation trees. We can prove

$$\vdash \forall a \forall x \forall y_1 \forall y_2 \forall t_2 (a[x] \simeq y_1, t_1 \wedge a[x] \simeq y_2, t_2 \to y_1 = y_2 \wedge t_1 = t_2)$$

by a fairly straightforward, if tedious, induction on t_1. This trivially implies 5.3(2). The proof can be carried out in $I\Sigma_1$ because the above formula is Π_1 as both formulas in its antecedent are Σ_1 and induction on Π_1-formulas is admissible.

5.6 Proof of the recurrence 5.1(1). After we have introduced the function f from the schema of course of values recursion with measure by 5.3(3) we wish to prove the recurrence 5.1(1): $f(x_1, \ldots, x_n) = \tau[x_1, \ldots, x_n]$.

For any term ρ constructed similarly as τ from Par. 5.1 we we wish to show that that the denotation of ρ is correctly evaluated by proving

$$\vdash x = x_1, \ldots, x_n, 0 \wedge \ulcorner \rho \urcorner[x] \simeq y, t \rightarrow y = \rho \tag{1}$$

by metainduction on the construction of ρ. We just show here the most interesting case when ρ is a guarded application of f 5.4(2). We assume the antecedent and using 5.3(1) in 5.4(1) we obtain $\ulcorner \rho_1, \ldots, \rho_n, 0 \urcorner[x] \simeq y_1, t_1$ for some y_1 and t_1. By IH we then obtain $y_1 = \rho_1, \ldots, \rho_n, 0$ and from 5.3(3) we get $\ulcorner \tau \urcorner[y_1] \simeq f(\vec{\rho}), t_0$ for some t_0. We consider two cases. If $\mu[\vec{\rho}] < \mu[\vec{x}]$ then from the assumption we get $\ulcorner \tau \urcorner[y_1] \simeq y, t_2$ for some t_2. Hence $y = f(\vec{\rho})$ by 5.3(2). If $\mu[\vec{\rho}] \geq \mu[\vec{x}]$ then we get $y = 0$ from the assumption. Combining the two cases together yields

$$y = D(\mu[\vec{\rho}] <_* \mu[\vec{x}], f(\vec{\rho}), 0) = \rho \ .$$

With the schema of theorems (1) proved, we assume $x = x_1, \ldots, x_n, 0$, get $\ulcorner \tau \urcorner[x] \simeq f(\vec{x}), t$ for some t from 5.3(3), and use the schema with $\rho := \tau$ to obtain $f(\vec{x}) = \tau[\vec{x}]$.

6 Clausal Language

The schema of course of values recursion with measure is extremely powerful but also difficult to work with. Its main problem is that the definitions are not readable. In this section we will outline a readable language for recursive definitions consisting of Horn clauses. The language is called CL (*C*lausal *L*anguage).

We will reduce the definitions in CL to course of values recursion with measure through an intermediate language of *cased* definitions. In addition to their mediating purpose, cased definitions are suitable both as a compact representation for the intelligent proof assistant and for the evaluation of definitions by computers.

6.1 Case formulas. Cased definitions use *case* formulas which are formulas of PA of certain shape. A case formula α is either a formula of PA (which depends on the kind of the definition), called the *ground* formula, or else a cased formula proper of a form:

$$\forall \vec{y}_0 (\phi_0[\vec{y}_0] \rightarrow \alpha_0[\vec{y}_0]) \wedge \cdots \wedge \forall \vec{y}_{k-1} (\phi_{k-1}[\vec{y}_{n_{k-1}}] \rightarrow \alpha_{k-1}[\vec{y}_{n_{k-1}}]) \tag{1}$$

where $\alpha_0, \ldots, \alpha_{k-1}$ are case formulas.

Case formulas are not fixed but they have to satisfy certain conditions. Let $\vec{x}$ be all free variables, called *parameters*, of the case formula (1). The number $k > 0$ is the *arity* of α. For $0 \leq i < k$ the i-th conjunct, called the *i-th alternative*, may have *local* variables $\vec{y}_i \equiv y_1, \ldots, y_{n_i}$ where we do not exclude $n_i = 0$. We require that no local variable is among the parameters $\vec{x}$ (this can be assured by the renaming of local variables). The formula ϕ_i of PA is called the *guard* and the formula α_i the *body* of the alternative.

The case formula α has its associated *discriminator* which is a term $\tau[\vec{x}]$ of PA. The discriminator must satisfy the *existence* condition:

$$\vdash \exists \vec{y}_0, \tau[\vec{x}] = \overline{0}, y_1, \ldots, y_{n_0}, 0 \vee \cdots \vee \exists \vec{y}_{k-1}, \tau[\vec{x}] = \overline{k-1}, y_1, \ldots, y_{n_{k-1}}, 0 \quad (2)$$

where if $n_i = 0$ the corresponding disjunct is $\tau[\vec{x}] = \overline{i}, 0$. For $i < k$ every guard $\phi_i[\vec{y}_i, \vec{x}]$ must satisfy the *uniqueness* condition:

$$\vdash \phi_i[y_1, \ldots, y_{n_i}, \vec{x}] \leftrightarrow \tau[\vec{x}] = \overline{i}, y_1, \ldots, y_{n_i}, 0 \ . \quad (3)$$

Important properties of (1) are for all $i < k$:

$$\vdash \phi_i \rightarrow (\alpha \leftrightarrow \alpha_i) \ . \quad (4)$$

Indeed, if ϕ_i then $\tau = \overline{i}, \vec{y}_i, 0$ and $\forall \vec{y}_j \neg \phi_j$ where $i \neq j < k$ by (3). Hence $\alpha \leftrightarrow \alpha_i$.

6.2 Cased definitions of predicates. Assume that we are given a case formula $\alpha[\vec{x}]$ with $\vec{x}$ its parameters and, if α applies P, also a measure term $\mu[\vec{x}]$. Provided three additional conditions given below are met, we can use the formula in a (recursive) definition of a predicate P satisfying:

$$\vdash P(\vec{x}) \leftrightarrow \alpha[\vec{x}] \ . \quad (1)$$

The conditions are a) that the ground formulas of α must be the nullary connectives $\top$ or $\bot$. b) Recursive applications of P in the guards of case formulas in α must be *guarded* as follows:

$$\mu[\vec{\rho}] < \mu[\vec{x} \wedge P(\vec{\rho}) \ . \quad (2)$$

c) Recursive applications of P in the discriminarors of case formulas in α must be of the usual guarded form:

$$D(\mu[\vec{\rho}] <_* \mu[\vec{x}], P_*(\vec{\rho}), 0) \ . \quad (3)$$

If the three conditions are met we call (1) a *cased definition* of P. This definition is reduced to the course of values recursion with measure $P_*(\vec{x}) = \alpha^*[\vec{x}]$ defining the characteristic function of P.

We designate by α^* the *translation* of the case formula α which is a term suitable for a course of values definition with measure. The translation is defined by induction on the construction of α. If α is a ground formula then if $\alpha \equiv \top$ then $\alpha^* \equiv 1$, and if $\alpha \equiv \bot$ then $\alpha^* \equiv 0$. If α is of a form 6.1(1) with τ its discriminator then its translation is:

$$\begin{aligned} \alpha^* \equiv D(H(\tau) =_* 0, \alpha_0^*[(T(\tau))_0, \ldots, (T(\tau))_{n_0 - 1}], \ldots \\ D(H(\tau) =_* \overline{k-1}, \alpha_{k-1}^*[(T(\tau))_0, \ldots, (T(\tau))_{n_{k-1} - 1}], 0)) \ . \end{aligned}$$

It should be clear that for the function $P_*(\vec{x}) = \alpha^*[\vec{x}]$ and predicate $P(\vec{x}) \leftrightarrow P_*(\vec{x}) = 1$ PA proves the recurrence (1).

$$\begin{aligned}
&\mathit{Trm}(x) \leftrightarrow \forall i(x = \mathbf{x}_i \to \top) \wedge \forall n(x = \mathbf{N}(n) \to \top) \wedge \\
&\quad \forall a \forall b(x = \langle a, b\rangle \to (\mathit{Trm}'(a) \to (\mathit{Trm}'(b) \to \top) \wedge (\neg \mathit{Trm}'(b) \to \bot)) \wedge \\
&\qquad\qquad\qquad\qquad (\neg \mathit{Trm}'(a) \to \bot)) \wedge \\
&\quad \forall i \forall a(x = \mathbf{g}_i(a) \to (\mathit{Trm}'(a) \to \top) \wedge (\neg \mathit{Trm}'(a) \to \bot)) \wedge \\
&\quad \forall a(x = \mathbf{f}(a) \to (\mathit{Trm}'(a) \to \top) \wedge (\neg \mathit{Trm}'(a) \to \bot)) \wedge (\mathit{Notag}(x) \to \bot)
\end{aligned}$$

Figure 4: Cased definition of *Trm*.

6.3 Example of a cased definition: *Trm***.** Recall the course of values definition of the characteristic function of the predicate *Trm* from Par. 4.8. The same predicate is defined by the cased definition in Fig. 4 which uses two kinds of case formulas:

$$\begin{aligned}
&\forall i(x = \mathbf{x}_i \to \alpha_0) \wedge \forall n(x = \mathbf{N}(n) \to \alpha_1) \wedge \forall a \forall b(x = \langle a, b\rangle \to \alpha_2) \wedge \\
&\quad \forall i \forall a(x = \mathbf{g}_i(a) \to \alpha_3) \wedge \forall a(x = \mathbf{f}(a) \to \alpha_4) \wedge (\mathit{Notag}(x) \to \alpha_5) \\
&(P(\vec{\rho}) \to \alpha_0) \wedge (\neg P(\vec{\rho}) \to \alpha_1) \qquad (1)
\end{aligned}$$

The first kind is a *disjoint union* formula whose discriminator $\mathit{Dtr}(x)$ satisfies:

$$\begin{aligned}
&\vdash \mathit{Dtr}(\mathbf{x}_i) = 0, i, 0 \wedge \mathit{Dtr}(\mathbf{N}(n)) = 1, i, 0 \wedge \mathit{Dtr}(\langle a, b\rangle) = 2, a, b, 0 \wedge \\
&\quad \mathit{Dtr}(\mathbf{g}_i(a) = 3, i, a, 0 \wedge \mathit{Dtr}(\mathbf{f}(a)) = 4, a, 0
\end{aligned}$$

and $\mathit{Dtr}(x) = 5, 0$ otherwise. One of the cases always applies and with the definition $\mathit{Notag}(x) \leftrightarrow \mathit{Dtr}(x) = 5, 0$, 6.1(3) is satisfied. For instance, we have $\vdash x = \mathbf{g}_i(a) \leftrightarrow \mathit{Dtr}(x) = 3, i, a, 0$.

The second case formula is a schema of *negation* formulas for atomic formulas $P(\vec{\rho})$ with the discriminator term $(1 \dot{-} P_*(\vec{\rho})), 0$. Notice that if $P(\vec{\rho})$ holds then the term denotes $0, 0$ and $1, 0$ otherwise.

The measure term $\mu[x]$ which goes down in the recursive applications of *Trm* is the identity term x. Notice, for instance, that we have $\mu[a] = a < x = \mu[x]$ when $x = \langle a, b\rangle$. We have abbreviated in Fig. 4 the guarded recursive applications $\tau < x \wedge \mathit{Trm}(\tau)$ to $\mathit{Trm}'(\tau)$.

6.4 Cased definitions of functions. Assume that we given a case formula $\alpha[\vec{x}, y]$ with $\vec{x}, y$ its parameters and, if α applies P, also a measure term $\mu[\vec{x}]$. If two conditions given bellow are met we can use the formula in a (recursive) definition of a function f such that

$$\vdash f(\vec{x}) = y \leftrightarrow \alpha[\vec{x}, y] \qquad (1)$$

will hold. The conditions are a) the ground formulas of α must be of a form $\rho = y$ for a term ρ and the variable y should not appear anywhere else. b) All recursive applications of f in the case formulas and in the discriminarors of α must be in the usual guarded form: $D(\mu[\vec{\rho}] <_* \mu[\vec{x}], f(\vec{\rho}), 0)$.

If the two conditions are met, we call the formula (1) a *cased* definition of f. This is reduced by translation to the course of values recursion with measure $f(\vec{x}) = \alpha^*[\vec{x}]$ defining the same function. The translation converts the ground formulas $\rho = y$ to the terms ρ and the case formulas exactly as in Par. 6.2. It should be clear that PA then proves the recurrence (1).

$$
\begin{aligned}
[\sqrt{x}] = y \leftrightarrow (x < 4 \rightarrow (x = 0 \rightarrow 0 = y) \wedge (x > 0 \rightarrow 1 = y)) \wedge \\
(x \geq 4 \rightarrow \forall z \forall i (x = 4{\cdot}z{+}i \wedge i < 4 \rightarrow \\
\forall s([\sqrt{z}]' = s \rightarrow (x < (2{\cdot}s{+}1)^2 \rightarrow 2{\cdot}s = y) \wedge \\
(x \geq (2{\cdot}s{+}1)^2 \rightarrow 2{\cdot}s{+}1 = y))))
\end{aligned}
$$

Figure 5: Cased definition of $[\sqrt{x}]$.

6.5 Example of a cased definition: $[\sqrt{x}]$. The *greatest integer in square root function* $[\sqrt{x}]$ can be introduced into PA by:

$$[\sqrt{x}] = y \leftrightarrow y^2 \leq x < (y+1)^2 \ .$$

However, this is a bad definition to compute from (by trying out the successive values for y starting with 0) because it runs in time $\mathcal{O}(x)$. The cased definition in Fig. 5 defines the same function in a computationally optimal form with the running time $\mathcal{O}(\log(x))$.

There are two kinds of case formulas in the cased definition of $[\sqrt{x}]$ which are mere variations of the negation case formulas 6.3(1):

$$(\tau_1 < \tau_2 \rightarrow \alpha_0) \wedge (\tau_1 \geq \tau_2 \rightarrow \alpha_1) \qquad \text{and} \qquad (\tau = 0 \rightarrow \alpha_0) \wedge (\tau > 0 \rightarrow \alpha_1) \ .$$

There are also two case formulas of arity 1 which correspond to the *let* constructs of declarative programming languages:

$$\forall y(\tau = y \rightarrow \alpha_0) \quad \text{and} \quad \forall q \forall r(\tau = \overline{n+1}{\cdot}q + r < \overline{n+1} \rightarrow \alpha_0) \ .$$

The purpose of both cases formulas is to assign shorter name(s) to the term τ and to avoid its multiple evaluation when the local variables are referenced more than once in the body. The discriminator term for the first case formula is $0, \tau, 0$ and for the second one $0, \tau \div \overline{n+1}, \tau \bmod \overline{n+1}, 0$.

The measure is the identity term x and we use the abbreviation: $[\sqrt{z}]' \equiv D(z {<_*} x, [\sqrt{z}], 0)$.

6.6 Unfolding of cased definitions. The transformation of cased definitions to clausal definitions depends on

$$\vdash (\psi \leftrightarrow \alpha) \leftrightarrow (\phi_0 \rightarrow (\psi \leftrightarrow \alpha_0)) \wedge \cdots \wedge (\phi_{k-1} \rightarrow (\psi \leftrightarrow \alpha_{k-1})) \tag{1}$$

where α is the case formula 6.1(1) and the free variables of ψ are among the parameters of α. In the direction $\rightarrow$ assume $\psi \leftrightarrow \alpha$ and ϕ_i for $i < k$. Then $\psi \leftrightarrow \alpha \leftrightarrow \alpha_i$ by 6.1(4). Vice versa, assume the right-hand-side. We have $\tau = \bar{i}, \vec{y}_i, 0$ for some $i < k$ and $\vec{y}_i$ by 6.1(2) and ϕ_i by 6.1(3). Thus $\psi \leftrightarrow \alpha_i$ from the assumption and $\psi \leftrightarrow \alpha$ by 6.1(4).

Let $\vdash \psi \leftrightarrow \alpha$ be a cased definition of a predicate P ($\psi \equiv P(\vec{x})$) or a function f ($\psi \equiv f(\vec{x}) = y$). Starting from $\top \rightarrow (\psi \leftrightarrow \alpha)$, which we write as $\psi_0 \rightarrow (\psi \leftrightarrow \alpha)$, we obtain from (1):

$$\begin{aligned} \vdash (\psi_0 \rightarrow (\psi \leftrightarrow \alpha)) \leftrightarrow (\psi_0 \wedge \phi_0 \rightarrow (\psi \leftrightarrow \alpha_0)) \wedge \cdots \wedge& \\ (\psi_0 \wedge \phi_{k-1} \rightarrow (\psi \leftrightarrow \alpha_{k-1}))& \end{aligned}$$

by one *unfolding* of the case formula α. We then note that any of the conjuncts has for $\psi_1 \equiv \psi_0 \wedge \phi_i$ a form $\psi_1 \rightarrow (\psi \leftrightarrow \alpha')$ again and we unfold every single conjunct again. We continue in this way until we reach the ground formulas in all conjuncts.

The definition $\psi \leftrightarrow \alpha$ will be then equivalent to the conjunction of

$$\phi_{i_1} \wedge \phi_{i_2} \cdots \wedge \phi_{i_m} \rightarrow (\psi \leftrightarrow \beta)$$

for all possible combinations of guards in α with β a ground formula.

$$\begin{aligned}
&\mathit{Trm}(x) &&\leftarrow x = \mathbf{x}_i \\
&\mathit{Trm}(x) &&\leftarrow x = \mathbf{N}(n) \\
&\mathit{Trm}(x) &&\leftarrow x = \langle a, b\rangle \wedge \mathit{Trm}'(a) \wedge \mathit{Trm}'(b) \\
&\neg\,\mathit{Trm}(x) &&\leftarrow x = \langle a, b\rangle \wedge \mathit{Trm}'(a) \wedge \neg\,\mathit{Trm}'(b) \\
&\neg\,\mathit{Trm}(x) &&\leftarrow x = \langle a, b\rangle \wedge \neg\,\mathit{Trm}'(a) \\
&\mathit{Trm}(x) &&\leftarrow x = \mathbf{g}_i(a) \wedge \mathit{Trm}'(a) \\
&\neg\,\mathit{Trm}(x) &&\leftarrow x = \mathbf{g}_i(a) \wedge \neg\,\mathit{Trm}'(a) \\
&\mathit{Trm}(x) &&\leftarrow x = \mathbf{f}(a) \wedge \mathit{Trm}'(a) \\
&\neg\,\mathit{Trm}(x) &&\leftarrow x = \mathbf{f}(a) \wedge \neg\,\mathit{Trm}'(a) \\
&\neg\,\mathit{Trm}(x) &&\leftarrow \mathit{Notag}(x)
\end{aligned}$$

$$\begin{aligned}
&[\sqrt{x}] = 0 &&\leftarrow x < 4 \wedge x = 0 \\
&[\sqrt{x}] = 1 &&\leftarrow x < 4 \wedge x > 0 \\
&[\sqrt{x}] = 2{\cdot}s &&\leftarrow x \geq 4 \wedge x = 4{\cdot}z{+}i \wedge i < 4 \wedge [\sqrt{z}]' = s \wedge x < (2{\cdot}s{+}1)^2 \\
&[\sqrt{x}] = 2{\cdot}s{+}1 &&\leftarrow x \geq 4 \wedge x = 4{\cdot}z{+}i \wedge i < 4 \wedge [\sqrt{z}]' = s \wedge x \geq (2{\cdot}s{+}1)^2
\end{aligned}$$

Figure 6: Unfolded cased definitions.

Now, if $\psi \equiv f(\vec{x}) = y$ then $\beta \equiv \rho = y$ and so $(\psi \leftrightarrow \beta) \leftrightarrow f(\vec{x}) = \rho$ because y appears nowhere else. If $\psi \equiv P(\vec{x})$ then, since $\beta \equiv \top$ or $\beta \equiv \bot$, $\psi \leftrightarrow \beta$ is equivalent to $P(\vec{x})$ or $\neg P(\vec{x})$. We accordingly simplify all equivalences $(\psi \leftrightarrow \beta)$ whereby we obtain a conjunction of Horn formulas defining the same ψ as the cased definition $\psi \leftrightarrow \beta$.

Figure 6 shows the cased definitions Par. 6.3 and Par. 6.5 unfolded. The custom of writing the clauses with converse implications comes from the programming language PROLOG.

6.7 Clausal language. We can perform some obvious simplification like substituting for superfluous variables with Horn clauses in Fig. 6. In order to improve the readability we can conventionally eliminate the negative clauses in predicate definitions and clauses yielding 0 in function definitions. Such clauses are called *default* clauses. The simplified clauses are given in Fig. 7. Notice that we have removed the guards surrounding recursive applications because they are implied by assumptions. Such clauses are called *regular*.

$$
\begin{aligned}
&\mathit{Trm}(\mathbf{x}_i) \\
&\mathit{Trm}(\mathbf{N}(n)) \\
&\mathit{Trm}(\langle a, b\rangle) \leftarrow \mathit{Trm}(a) \wedge \mathit{Trm}(b) \\
&\mathit{Trm}(\mathbf{g}_i(a)) \leftarrow \mathit{Trm}(a) \\
&\mathit{Trm}(\mathbf{f}(a)) \quad \leftarrow \mathit{Trm}(a)
\end{aligned}
$$

$$
\begin{aligned}
&[\sqrt{0}] = 0 && \\
&[\sqrt{i{+}1}] = 1 && \leftarrow i \leq 3 \\
&[\sqrt{4{\cdot}x{+}i}] = 2{\cdot}s && \leftarrow x > 0 \wedge i < 4 \wedge [\sqrt{x}] = s \wedge 4{\cdot}x{+}i < (2{\cdot}s{+}1)^2 \\
&[\sqrt{4{\cdot}x{+}i}] = 2{\cdot}s{+}1 && \leftarrow x > 0 \wedge i < 4 \wedge [\sqrt{x}] = s \wedge 4{\cdot}x{+}i \geq (2{\cdot}s{+}1)^2
\end{aligned}
$$

Figure 7: Clausal definitions.

The clauses in Fig. 7 are in the *clausal language*. Experience with such a presentation of definitions shows that programmers quickly recognize the default situations without having to clutter the 'positive' clauses with pathological cases. The intelligent proof assistant of CL keeps internally cased definitions from which it derives properties of the defined function or predicate.

Cased definitions are also used by the interpreter of CL to evaluate applications f defined functions and predicates to numerals. The reader should visualize the evaluation as a simplification (reduction) process where a case formula is evaluated first by selecting the appropriate alternative and assigning (binding) values to its local variables (this is done by evaluating its discriminator term). Then the case formula in the selected alternative is reduced in the same way until a truth value of a predicate or a numeral as the value of a function is obtained. Notice that the *let* $[\sqrt{z}] = s$ in the clausal definition of $[\sqrt{x}]$ is thus evaluated as a part of its discriminator, the value of the recursive application is bound to s which is used twice in the rest of the clauses (once for a test and once to yield the result). The removal of the let construct by substituting for s would result in the exponential blowup of this efficient algorithm whose efficiency would be thus reduced to that of the 'naive' algorithm mentioned in Par. 6.5.

One of the FAQ's (frequently asked questions) we are asked about CL is how can be the evaluation of functions into numerals be efficient? Here it is widely assumed that the pairing function applied to two numerals is reduced to a numeral which is obviously quite inefficient. The interpreter (or compiler) of CL works (conceptually that is) as if it reduced the values of arguments and of local variables in the *mixed* notation for numerals. Numerals τ in the mixed notations are a mixture of binary and pair representations and they are constructed from 0 by the *binary successors* $2{\cdot}\tau_1$, $2{\cdot}\tau_1{+}1$ and by pairing (τ_1, τ_2). Binary parts of numerals are stored in computer's memory as binary numbers and pairs are stored as *cons-cells* of LISP. Conversions beteen pair and binary numerals are needed only in rare situations when a program calls for the evaluation of, say, $(3, 5) + 12$.

References

[Ben62] J. H. Bennett. *On Spectra.* Ph.D. dissertation. Princeton, 1962.

[BJ80] G. Boolos, R. Jeffrey. *Computability and Logic.* Cambridge Univ. Press, 1980.

[Bus94] S. R. Buss. *The witness function method and provably recursive functions of Peano arithmentic.* In D. Prawitz et al, editors Logic, Methodology and Philosophy of Science IX, 1991. North Holland 1994.

[HP93] P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic.* Springer Verlag, 1993.

[Imm98] N. Immerman. *Descriptive Complexity.* Springer 1998.

[KV98] J. Komara, P. J. Voda. *Theorems of Péter and Parson in Computer Programming.* Computer Science Logic Conf. (CSL-98), Brno The Czech Rep. 1998. Lecture Notes in Computer Science vol 1584, Springer Verlag 1999.

[Lind9?] S. Lindell. *How to define exponentiation from addition and multiplication in first-order logic on finite structures.* Manuscript 199?

[Nel86] E. Nelson. *Predicative Arithmetic.* Mathematical Notes. Princeton Univ. Pres, 1986.

[Par72] Ch. Parsons. *On n-Quantifier Induction.* JSL. vol 37, Sept. 1972.

[Pet32] R. Péter. *Über den Zusammenhang der verschiedenen Begriffe der rekursiver Funktion*. Math. Ann. vol, 110 (1932), pages 612-632.

[Sho67] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[Smu92] R. M. Smullyan. *Gödel's Incompleteness Theorem.* Oxford University Press, 1992.

[Tho99] S. Thompson. *Haskell The Craft of Functional Programming*. Addison Wesley 1999.

[Vo94] P. J. Voda. *Subrecursion as a basis for a feasible programming language*, in L. Pacholski and J. Tiuryn, editors, Proceedings of CSL'94, number 933 in LNCS Springer Verlag, 1995, pages 324-338.

[Vo98] P. J. Voda. *A Note On the Exponentiation Relation in Peano Arithmetic,* www.fmph.uniba.sk/∼voda, 1998.

[Vo01] P. J. Voda. *Metamathematics of Computer Programming,* (in preparation), see ww.fmph.uniba.sk/∼voda, 2001.