Theorems of Péter and Parsons in Computer Programming

Ján Komara, Paul J. Voda

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

1 Introduction

This paper describes principles behind a declarative programming language CL (*Clausal Language*) which comes with its own proof system for proving properties of defined functions and predicates. We use our own implementation of CL in three courses in the first and second years of undergraduate study. By unifying the domain of LISP's S-expressions with the domain $\mathbb{N}$ of natural numbers we have combined the LISP-like simplicity of coding with the simplicity of semantics. We deal just with functions over $\mathbb{N}$ within the framework of formal Peano arithmetic. We believe that most of the time this is as much as is needed. CL is thus an extremely simple language which is completely based in mathematics.

We illustrate in Sect. 2 how simple schemas of recursion and/or induction lead to definitions of functions which are not necessarily the most efficient ones to compute. For efficient computation one needs stronger schemas of recursion and/or induction. This calls for a delicate balance between the strength of the language and that of its formal system. The formal system must be strong enough to prove the recurrences of strong recursive schemas as theorems. On the other hand, the language must be *specification complete* and admit for every *specification* formula $\forall x \exists y \phi(x,y)$ (ϕ is quantifier-free) proved in the formal system a function f *witnessing* the formula: $A(x, f(x))$. Moreover, this must be provable in the formal system.

In Sect. 3 we introduce a very general schema of *recursion with measure* which combines expressivity with efficient computation. The justification of the schema is easy, it is defined by transfinite recursion. The *characterization* problem, i.e. the question of what class of functions is admitted by the schema, is far from trivial. We rely on a theorem of Péter [Pét32] and show that the schema admits exactly the primitive recursive functions.

The proof system of CL is a fragment of Peano arithmetic and its soundness is obviously guaranteed by the standard model $\mathcal{N}$ in natural numbers. In order to achieve the specification completeness we select in Sect. 4 the fragment called $I\Sigma_1$ whose witnessing functions are primitive recursive. A difficult problem arises: we wish to extract functions defined by the general schema from proofs where induction with Π_2-formulas is naturally called for. Unfortunately, $I\Sigma_1$ admits induction **axioms** with at most Π_1-formulas. Fortunately, a theorem of Parsons [Par70] comes to our rescue: induction **rules** for Π_2-formulas are admissible in proofs of Π_2-specifications.

What is amazing to us is that the relatively minor theorems of Péter and Parsons turn out to be crucially important in the characterization problem of a really existing and usable declarative programming language. What is even more amazing is that the theorems are closely connected. Namely, the proof of the special case of theorem of Parsons in Thm. 4.15 calls for functions from Thm. 3.7 by Péter. We hope that apart from the two quite technical proofs (which are both new), the rest of the paper will be accessible to any computer scientist with interest in logic but without the specialized knowledge in recursion and proof theory.

1.1 Pairing function. CL combines the domain of S-expressions of LISP with the domain of natural numbers $\mathbb{N}$ in order to achieve a productive symbiosis of easy coding in the first domain with the standard setting of theory of computation and formal arithmetic in the second domain. This is done with the help of a *pairing function* $\langle x, y\rangle$ such that 0 is the only *atom*, i.e. $0 \neq \langle x, y\rangle$ and every positive number is in the range of the pairing function. The function satisfies the *pairing property*:

$$\langle x_1, x_2\rangle = \langle y_1, y_2\rangle \rightarrow x_1 = y_1 \wedge x_2 = y_2$$

and we have $x < \langle x, y\rangle$ and $y < \langle x, y\rangle$. From the last two conditions we get *pair induction*:

$$\phi(0) \wedge \forall x \forall y(\phi(x) \wedge \phi(y) \rightarrow \phi\langle x, y\rangle) \rightarrow \forall x\, \phi(x)\ . \tag{1}$$

The simplest pairing function is the Cantor's *diagonal* function offset by one:

$$\langle x, y\rangle = \frac{(x+y)(x+y+1)}{2} + x + 1\ .$$

Projection functions H (*Head*) and T (*Tail*) satisfy $H(0) = T(0) = 0$, $H\langle v, w\rangle = v$, and $T\langle v, w\rangle = w$. All three functions are primitive recursive (see [Dav85]). *Pair size* $|x|$ of a number x is the number of pairing operations needed to construct x and we have $|0| = 0$ and $|\langle v, w\rangle| = |v| + |w| + 1$. This is a definition by *course of values* which is reducible to primitive recursion.

For $n \geq 3$ we abbreviate $\langle x_1, \langle x_2, \cdots, x_n\rangle\rangle$ to $\langle x_1, x_2, \cdots, x_n\rangle$. Because 0 is the only atom, every natural number is uniquely a *list*, i.e. a code of a finite sequence of natural numbers where the empty sequence is coded by the empty list 0 (LISP's *nil*) and a sequence $x_1\ x_2\ \ldots\ x_n$ is coded by the list $\langle x_1, x_2, \cdots, x_n, 0\rangle$. The *length* $L(x)$ of the list x is defined by $L(0) = 0$ and $L\langle v, w\rangle = L(w) + 1$.

In order to obtain a simple recursive characterization of subelementary complexity classes (such as P) one should use a pairing function such that $|x| = \Omega(\log(x))$ [Vod94]. CL uses such a function but for the purposes of this paper this requirement is not important and we use the simpler Cantor's function which does not satisfy the requirement.

1.2 Contractions. In the presence of pairing it suffices to deal only with unary functions because to each n-ary function f we can define its unary *contraction* function $\langle f\rangle$ such that

$$f(x_1, \cdots, x_n) = \langle f\rangle\langle x_1, \cdots, x_n\rangle\ .$$

The use of contractions in CL frees the symbol comma from its role of the separator of arguments. CL uses the infix pairing notation x, y instead of $\langle x, y\rangle$ and deals with unary functions and predicates only. However, such an use of comma requires some practice in reading definitions which we do not want to impose on a casual reader of this paper and so we continue to use here n-ary functions and denote the pairing by $\langle x, y\rangle$.

2 Examples of Clausal Definitions

In this section we give example function definitions in the form of clauses of CL. We do not dwell on the syntax of clauses as we consider them almost self-explanatory. The only requirement imposed on clausal definitions is that they should be mechanically transformable into a closed form of definitions discussed in the following section. We intend to demonstrate that simple recursion and/or induction does not always lead to a definition which, when used as rewriting rules, is efficient. A computationally optimal definition usually needs a more complex recursion and/or induction.

2.1 Pair recursion. The function $x \oplus y$ concatenating lists x and y has a following clausal definition by course of values recursion:

$0 \oplus y = y$
$\langle v, w\rangle \oplus y = \langle v, w \oplus y\rangle$.

The function Rt has a similar clausal definition:

$Rt(0) = 0$
$Rt\langle v, w\rangle = \langle 0, Rt(v) \oplus Rt(w)\rangle$.

The function Rt takes a number x and yields a list such that

$$L\, Rt(x) = |x|\ . \tag{1}$$

This list, so called *Right Tree*, contains at most zeros. Declarative programmers will recognize $Rt(x)$ as a function *flattening* the tree x. Recursion from $\langle v, w\rangle$ to v and/or w is called *pair recursion.*

Alternatively, the definition of Rt can be automatically extracted as a *witness* from the proof of its *specification*: $\exists z\, L(z) = |x|$. The proof is by pair induction on x. If $x = 0$ take $z = 0$. If $x = \langle v, w\rangle$ obtain z_1 and z_2 from IH and set $z = \langle 0, z_1 \oplus z_2\rangle$. We have

$$L(z) = L\langle 0, z_1 \oplus z_2\rangle = L(z_1) + L(z_2) + 1 \stackrel{IH}{=} |v| + |w| + 1 = |\langle v, w\rangle| \ .$$

The function Rt has a simple definition by pair recursion or it can be extracted from a proof by pair induction. Unfortunately, this is not a very efficient definition to use as rewrite rules for computation. The number of operations needed to evaluate $Rt(x)$ is quadratic in $|x|$ due to the repeated concatenation. We will remedy this situation in Par. 2.4.

2.2 Structural induction. As every worker in the field of declarative programming knows there is a need for a special kind of *structural* induction for recursive data structures. A typical example is a definition of the type of *binary trees with labels of type* T found in practically all functional languages: $Bt = E \mid Nd(T, Bt, Bt)$. Semantically a new sort freely generated by this definition is added to the domain of such a language. We cannot extend the domain $\mathbb{N}$ of CL but we can code binary trees into natural numbers. For that we need two *constructors*: a constant $E = \langle 0, 0\rangle$ and a function $Nd(n, a, b) = \langle 1, n, a, b\rangle$. The type Bt is in CL a predicate holding exactly of codes of binary trees:

$Bt(E)$
$Bt\, Nd(n, a, b) \leftarrow T(n) \wedge Bt(a) \wedge Bt(b)\ .$

Such a clausal definition is *minimal* in the sense that unless the truth of $Bt(x)$ follows from the clauses the predicate does not hold for x by *default.* The predicate is primitive recursive by course of values pair recursion because $a < Bt(n, a, b)$. We have the following principle of *structural induction* which can be used to prove that a property $\phi(x)$ holds of all (codes of) binary trees:

$$\phi(E) \wedge \forall n \forall a \forall b (T(n) \wedge \phi(a) \wedge \phi(b) \rightarrow \phi(Nd(n, a, b))) \wedge Bt(x) \rightarrow \phi(x)\ .$$

The principle is reduced to complete induction as follows. We assume its first two conjuncts and continue by complete induction on x with the induction formula $Bt(x) \rightarrow \phi(x)$. We take any x s.t. $Bt(x)$ and consider two cases (which follow from the minimality of Bt). If $x = E$ we use the first assumption. If $x = Nd(n, a, b)$ then $T(n)$, $Bt(a)$, and $Bt(b)$ and so, since $a, b < x$, we get $\phi(a)$ and $\phi(b)$ from IH. Thus we get $\phi(x)$ from the second assumption.

2.3 Substitution in parameters. Consider the following primitive recursive clausal definition of the minimum function $\min(x, y)$:

$\min(0, y) = 0$
$\min(x + 1, y) = y \leftarrow x + 1 = y$
$\min(x + 1, y) = \min(x, y) + 1 \leftarrow x + 1 \neq y\ .$

Using this definition as rewriting rules, and for sake of simplicity assuming that the evaluation of the identity predicate takes one step, we can see that the computation of $\min(x, y)$ for $x > 2 \cdot y$ will run longer (i.e. $(x - y)$-times) then it should (y-times). The last is is the case if minimum were computed from the following definition:

$\min(0, y) = 0$
$\min(x + 1, 0) = 0$
$\min(x + 1, y + 1) = \min(x, y) + 1\ .$

This is a definition by primitive recursion (on x) with *substitution in parameter* (y) which does not lead outside of primitive recursion. This reduction to primitive recursion is important extensionally for knowing that min is primitive recursive, but the function is to be computed directly from its clauses used as rewrite rules. Colson [Col91] has proved that this kind of intensional behavior is impossible with any primitive recursive derivation of the minimum function where the identities are used intensionally as rewriting rules.

The difference of the two definitions shows also in proofs of the property:

$$x \geq y \wedge \min(x,y) = y \vee x < y \wedge \min(x,y) = x \ . \tag{1}$$

The proof from the first derivation of min is by induction on x with the formula (1). The proof from the second derivation is also by induction on x but this time the induction formula has a greater quantifier complexity, namely $\forall y(1)$, in order to cater for the substitution to y.

We observe that even the second definition is computationally inefficient because the minimum can be computed exponentially faster by *recursion on notation* which goes from $2 \cdot x$ and $2 \cdot x + 1$ to x.

2.4 Nested recursion. We can obtain a more efficient definition of Rt by defining an auxiliary *accumulator* function Rta which does not apply concatenation. We will obtain the accumulator function as a witness of (1) to get $L\,Rta(x,a) = |x| + L(a)$. We can then clearly set $Rt(x) = Rta(x,0)$.

The accumulator function substitutes in the accumulator a so we prove

$$\forall a \exists z\, L(z) = |x| + L(a) \tag{1}$$

by pair induction on x. If $x = 0$ then it suffices to take $z = a$. If $x = \langle v, w\rangle$ then for a given a we obtain by IH a z_1 such that $L(z_1) = |w| + L(a)$. By another IH we obtain z_2 such that $L(z_2) = |v| + L(z_1)$. It now suffices to set $z = \langle 0, z_2\rangle$ because

$$L(z) = L\langle 0, z_2\rangle = L(z_2) + 1 \overset{IH}{=} |v| + L(z_1) + 1 \overset{IH}{=} |v| + |w| + L(a) + 1 = |\langle v, w\rangle| + L(a) \ .$$

The witness can be now automatically extracted from the proof to satisfy:

$Rta(0, a) = a$
$Rta(\langle v, w\rangle, a) = \langle 0, Rta(v, Rta(w, a))\rangle$.

This is a definition by *nested* pair recursion.

2.5 Assignments. The following definition by primitive recursion shows a construct which makes computation more efficient:

$f(0) = 1$
$f(x+1) = y, y \leftarrow f(x) = y$.

We could have defined the same function with the second clause $f(x+1) = f(x), f(x)$. The latter definition needs 2^x recursions to compute $f(x)$ whereas the *assignment* of $f(x)$ to the auxiliary variable y reduces the number of recursions to x.

2.6 Non-primitive recursive function. The well-known non-primitive recursive Ackermann-Péter function (see [Pét67,Ros82]) has the following clausal definition:

$A(0, y) = y + 1$
$A(x+1, 0) = A(x, 1)$
$A(x+1, y+1) = A(x, A(x+1, y))$.

By Thm. 3.9 this is a definition not permitted in CL.

3 Recursion with Measure

3.1 Syntax of recursion with measure. CL restricts the syntax of clausal definitions in such a way that the clauses can be mechanically transformed into *closed form*:

$$f(x_1, \cdots, x_n) = \tau(f, x_1, \cdots, x_n) . \tag{1}$$

The term τ is composed from the constant 0 and variables by applications of the successor function $S(x) = x + 1$, pairing $\langle x, y\rangle$, previously defined functions, recursive applications of f, and by three operators:

$$\begin{aligned} &\mathit{if}\ \tau_1 = S(x)\ \mathit{then}\ \tau_2(x)\ \mathit{else}\ \tau_3 &(2)\\ &\mathit{if}\ \tau_1 = \langle x, y\rangle\ \mathit{then}\ \tau_2(x, y)\ \mathit{else}\ \tau_3 &(3)\\ &\mathit{let}\ \tau_1 = x\ \mathit{in}\ \tau_2(x) . &(4)\end{aligned}$$

The *pattern* variables on the right-hand-side of the identities must be new and they are *bound* in terms with indicated variables. Such terms τ are well-known from LISP and from functional languages and they allow a fine degree of control over evaluation.

The actual implementation of CL has a slightly richer syntax of terms τ (for instance builtin arithmetic operators). The syntax of terms τ is not minimal because we wish to demonstrate that the coding of S-expressions into natural numbers brings the benefits of LISP-like programming without any deterioration of efficiency (see Paragraphs 3.4 and 3.5).

3.2 Semantics of recursion with measure. We assign the meaning to the three operators from Par. 3.1 as follows. The operator 3.1(4) denotes the same as the term $\tau_2(\tau_1)$. For the two *if* operators we introduce the *case discrimination* function D by primitive recursion: $D(0, y, z) = z$ and $D(x + 1, y, z) = y$. The operator 3.1(2) denotes the same as the term $D(\tau_1, \tau_2(\tau_1 \dot{-} 1), \tau_3)$ and the operator 3.1(3) denotes the same as the term $D(\tau_1, \tau_2(H(\tau_1), T(\tau_1)), \tau_3)$. The remaining constructs in terms τ have their standard meaning.

The definition 3.1(1) can be either *explicit* if f is not applied in τ or else it must be *regular*, in that there is a measure function $m : \mathbb{N}^n \mapsto \mathcal{O}$ into an initial segment of ordinal numbers. The recursion in τ *must go down in the measure* m in the sense that for every recursive application $f(\overline{\tau}_1, \cdots \overline{\tau}_n)$ in τ we have $m(\overline{\tau}_1, \cdots \overline{\tau}_n) < m(x_1, \cdots, x_n)$ under the assumption of all *conditions governing* the recursive application. If the application occurs in τ after some *then* (*in*) of an operator then it is governed by the condition after *if* (*let*). If the application occurs after some *else* of an operator then it is governed by $\tau_1 = 0$ for τ_1 from that operator.

The regularity condition guarantees that we have

$$\tau(f, x_1, \cdots, x_n) = \tau([f]_{x_1,\cdots,x_n}, x_1, \cdots, x_n)$$

where we denote by $[g]_{x_1,\cdots,x_n}$ the *m-restriction* of g, i.e the function satisfying:

$$[g]_{x_1,\cdots,x_n}(y_1, \cdots, y_n) = \begin{cases} g(y_1, \cdots, y_n) & \text{if } m(y_1, \cdots, y_n) < m(x_1, \cdots, x_n) \\ 0 & \text{otherwise} \end{cases}$$

It is well-known from set theory that there is a unique function f defined by transfinite recursion such that $f(x_1, \cdots, x_n) = \tau([f]_{x_1,\cdots,x_n}, x_1, \cdots, x_n)$ and f is thus the unique function satisfying 3.1(1).

3.3 Measures of example functions. The measure for the primitive recursion in both clausal definitions of min from Par. 2.3 as well as in the definition of f from Par. 2.5 is the identity function $I(x) = x < \omega$.

We can also use I as a measure for definitions by pair recursion of functions $|x|$, L, $\oplus$, Rt, and Rta but the measure $|x|$ is superior as the recursion in CL runs exponentially faster due to $|x| = \Omega(\log(x))$ (this, however, is not true for the Cantor's function used here).

The Ackermann-Péter function has the measure $m(x, y) = \omega \cdot x + y < \omega^2$. For instance, for the outer recursive application in the third clause we have

$$m(x, A(x, y)) = \omega \cdot x + A(x, y) < \omega \cdot (x + 1) + y + 1 = m(x + 1, y + 1) \ .$$

3.4 Numerals. *Unary numerals* are terms composed from the constant 0 by the application of the successor function $S(x) = x + 1$. The numeral $S^n(0)$, i.e. $\overbrace{S \cdots S}^{n}(0)$, denotes the number n. *Pair numerals* are terms composed from 0 by pairing $\langle \cdot, \cdot \rangle$. Clearly, to every natural number there is exactly one pair numeral denoting the number.

In order to facilitate not only effective but also *efficient* computation we introduce *mixed numerals* as terms composed from 0 by the successor function and pairing. It is now possible that two different mixed numerals denote the same number. Mixed numerals have a simple representation in computers. Unary numerals are represented in binary (say by the Bignum's of LISP). The mixed numeral $\langle \tau_1, \tau_2 \rangle$ is represented by a pointer to a LISP-cell with pointers to the representations of mixed numerals τ_1 and τ_2.

Conversions between mixed numerals are effectively computable in the sense that we can effectively find the mixed numerals denoting $\langle \tau_1, \tau_2 \rangle \dot{-} 1$ (this is used in 3.1(2)), $H\,S(\tau_1)$, and $T\,S(\tau_1)$ (this is used in 3.1(3)). The conversions are effective because the pairing and projection functions are primitive recursive.

3.5 Effective computability of recursion with measure. We say that an n-ary function f is *effectively computable* if there is a mechanical process which, given the mixed numerals $\tau_1, \ldots,$ τ_n denoting the numbers $k_1, \ldots\ k_n$ respectively, yields after a finite time a mixed numeral denoting the number $f(k_1, \ldots, k_n)$.

We now assume that all functions (expcept f) applied in a term τ from 3.1(1) are effectively computable. Suppose that we are given mixed numerals $\tau_1, \ldots, \tau_n$ as arguments for the (free) variables of τ. If the definition is explicit one can easily prove by induction on construction of τ that it is effectively computable into a mixed numeral. If the definition is recursive we assume that f can be computed for all mixed numerals $\overline{\tau}_1, \ldots, \overline{\tau}_n$ such that $m(\overline{\tau}_1, \cdots, \overline{\tau}_n) < m(\tau_1, \cdots, \tau_n)$ and show similarly that τ can be effectively computed into a mixed numeral (the outer induction is thus a transfinite induction on m).

Definitions of CL functions are almost always well-behaved in the sense that they perform list operations on numerals with outermost pairings and arithmetic operations on unary numerals. The need for the time consuming conversions is exceptional.

3.6 Multiply recursive functions. The Ackermann-Péter function A is an example of a 2-*recursive function* in the hierarchy of *multiply recursive functions* by Péter [Pét67,Ros82]. Its definition is by *nested double recursion.*

The function f is 1-*recursive* if it is defined by *nested simple recursion*:

$$f(0, \overline{x}) = g(x_1, \overline{x}) \tag{1}$$
$$f(z + 1, \overline{x}) = \tau(f(z, \cdot), z, \overline{x}) \tag{2}$$

where we denote by $\overline{x}$ the n-tuple of variables $x_1, \cdots, x_n$. The right-hand-side of the second identity should be understood as the term obtained from $\tau(f, z, \overline{x})$ by replacing all applications of $f(\overline{\tau}_1, \cdots, \overline{\tau}_n)$ by $f(z, \overline{\tau}_1, \cdots, \overline{\tau}_n)$. Nested simple recursion is a special case of recursion with measure where $m(z, \overline{x}) = z$. Note that the term τ does not need to be regular because the regularity is enforced by the restriction $f(z, \cdot)$.

For a measure function $m(\overline{x}) < \omega$ we can reduce a regular definition 3.1(1) into nested simple recursion by defining an auxiliary function $\overline{f}$:

$$\overline{f}(0, \overline{x}) = \tau(Z(\cdot), \overline{x})$$
$$\overline{f}(z + 1, \overline{x}) = \tau(\overline{f}(z, \cdot), \overline{x})$$

where $Z(\overline{x}) = 0$. We then explicitly set $f(\overline{x}) = \overline{f}(m(\overline{x}), \overline{x})$.

We now present a new proof of Péter's theorem [Pét32] done in a way which can be readily formalized by Thm. 4.9 in a small fragment of Peano arithmetic.

3.7 Péter's theorem. *Primitive recursive functions are closed under nested simple recursion.*

Proof. Because of pairing we may assume that the function f defined by a nested simple recursion is of the form $f(z, x)$. We assume that the function f is applied in τ at least once because otherwise there is nothing to prove. Finally, we assume that the *if* and *let* operators have been removed from the term τ as suggested in Par. 3.2. We 'unnest' from inside out and left to right the applications of f in τ by writing the definition in the clausal form:

$$\begin{aligned}&f(0, x) = g(x)\\ &f(z+1, x) = \tau_k(z, x, t_0, t_1, \ldots, t_{k-1}) \leftarrow f(z, \tau_0(z, x)) = t_0 \wedge f(z, \tau_1(z, x, t_0)) = t_1 \wedge \ldots \wedge\\ &\qquad f(z, \tau_{k-1}(z, x, t_0, t_1, \cdots, t_{k-2})) = t_{k-1} \quad .\end{aligned}$$

Here the terms τ_i for $0 \leq i \leq k$ contain at most the indicated variables and do not apply f. Let us until the end of the proof call a list t such that $L(t) \leq k$ a *table for* $f(z+1, x)$ if for all $0 \leq i < L(t)$ we have $(t)_i = f(z, \tau_i(z, x, (t)_0, (t)_1, \cdots, (t)_{i-1}))$. Here $(t)_i$ denotes the *list indexing* function $(t)_i = HT^i(t)$. The table is *partial* if $L(t) < k$ and *full* otherwise. Note that 0 is a partial table for every $f(z+1, x)$. We will use tables to hold the values of the auxiliary recursive applications needed in the evaluation of $f(z+1, x)$. A partial table t for $f(z+1, x)$ can be used to determine the value of the parameter $\tau_{L(t)}$ for the next recursive application to be evaluated. A full table determines the value of the term τ_k and hence, of the application $f(z+1, x)$. This is captured by a primitive recursive function V with an explicit clausal definition:

$$\begin{aligned}&V(z, x, 0) = \tau_0(z, x)\\ &V(z, x, \langle t_0, 0\rangle) = \tau_1(z, x, t_0)\\ &V(z, x, \langle t_0, t_1, 0\rangle) = \tau_2(z, x, t_0, t_1)\\ &\vdots\\ &V(z, x, \langle t_0, t_1, t_{k-2}, 0\rangle) = \tau_{k-1}(z, x, t_0, t_1, \cdots, t_{k-2})\\ &V(z, x, \langle t_0, t_1, t_{k-2}, t_{k-1}, 0\rangle) = \tau_k(z, x, t_0, t_1, \cdots, t_{k-2}, t_{k-1}) \ .\end{aligned}$$

Thus if t is a table for $f(z+1, x)$ then $V(z, x, t) = \tau_{L(t)}(z, x, (t)_0, (t)_1, \cdots, (t)_{L(t)-1})$. Hence, $V(z, x, t)$ yields for a partial t the parameter of the next recursive application of f to be evaluated. If t is full then we have $f(z+1, x) = V(z, x, t)$.

In order to evaluate the application $f(z, x)$ we will keep *lists* $\overline{t}$ *of tables for* $f(z, x)$ where $\overline{t}$ is a list of length z such that $(\overline{t})_i$ is a partial table for $f(z - i, x_i)$ for each $i < z$. The parameters x_i are such that $x_0 = x$ and $x_{i+1} = V(z - (i+1), x_i, (\overline{t})_i)$ for $i + 1 < z$. Note that for $i < z$ the parameters can be determined as $x_i = B(i, x, \overline{t})$ by a primitive recursive function B:

$$\begin{aligned}B(0, x, \overline{t}) &= x\\ B(i+1, x, \overline{t}) &= V(L(\overline{t}) \dot{-} (i+1), B(i, x, \overline{t}), (\overline{t})_i) \ .\end{aligned}$$

We note that $x_z = B(z, x, \overline{t})$ is the parameter for the application $f(0, x_z)$ to be computed next in the last table $(\overline{t})_{z-1}$ of the list of tables $\overline{t}$ for $f(1, x_{z-1})$. We also note that $\overline{t}_0 = \sigma(z)$, where $\sigma(0) = 0$ and $\sigma(x+1) = \langle 0, \sigma(x)\rangle$, is a list of tables for every $f(z, x)$ and that every list of tables for $f(0, x)$ must be 0.

In order to evaluate $f(z, x)$ for $z > 0$ we need the values of k recursive applications $f(z-1, \tau_i)$. If $z > 1$ each such an application needs again k evaluations of f. Thus we need k^z evaluations of f in order to compute $f(z, x)$. This suggests a computation strategy for f where we start the evaluation of $f(z, x)$ with the list of tables $\overline{t}_0 = \sigma(z)$ and after $j < k^z$-steps we will have list of tables for $f(z, x)$: $\overline{t}_j$ with each partial table $(\overline{t}_j)_i$ for $i < z$ having the length given by the i-th digit of the expansion of j in the positional representation with k digits 0 through $k - 1$. We do not suppress the leading zeros and count the digits from left to right starting with 0 and ending with $z - 1$.

The step $j+1$ of the evaluation will thus resemble the addition of one to the number j in the k-ary notation. We will add the value of $v = f(0, x_z)$ to the partial table $t = (\overline{t}_j)_{z-1}$ thereby obtaining a new table $t \oplus \langle v, 0\rangle$. Should this table become full, it will be reset to the empty table 0 in the list of tables $\overline{t}_{j+1}$ and we will propagate as a 'carry' the value $V(0, x_{z-1}, t \oplus \langle v, 0\rangle) = f(1, x_{z-1})$. If $j+1 < k^z$ then the process of carries rippling through will terminate in some partial table of $\overline{t}_{j+1}$. If $j+1 = k^z$ then $\overline{t}_{j+1}$ will be the empty list of tables $\sigma(z)$ again and the carry out will yield the value of $f(z, x)$.

We indicate no carry by 0 and a carry of the value v by $\langle 0, v\rangle$. The list of tables $\overline{t}_{j+1}$ is be produced by the function *Incr* which yields a pair consisting of a carry and of a list of tables such that $\mathit{Incr}(\overline{t}_j, x, \overline{t}_j) = \langle c, \overline{t}_{j+1}\rangle$. The function has a clausal definition by pair recursion:

$$\begin{aligned}
&\mathit{Incr}(\overline{s}, x, 0) = \langle\langle 0, g\, B(L(\overline{s}), x, \overline{s})\rangle, 0\rangle\\
&\mathit{Incr}(\overline{s}, x, \langle t, \overline{t}\rangle) = \langle 0, t, \overline{t}_1\rangle \leftarrow \mathit{Incr}(\overline{s}, x, \overline{t}) = \langle 0, \overline{t}_1\rangle\\
&\mathit{Incr}(\overline{s}, x, \langle t, \overline{t}\rangle) = \langle 0, t \oplus \langle v, 0\rangle, \overline{t}_1\rangle \leftarrow \mathit{Incr}(\overline{s}, x, \overline{t}) = \langle\langle 0, v\rangle, \overline{t}_1\rangle \wedge L(t) < k-1\\
&\mathit{Incr}(\overline{s}, x, \langle t, \overline{t}\rangle) = \langle\langle 0, V(L(\overline{t}), B(L(\overline{s}) \dot{-} (L(\overline{t})+1), x, \overline{s}), t \oplus \langle v, 0\rangle)\rangle, 0, \overline{t}_1\rangle \leftarrow\\
&\qquad \mathit{Incr}(\overline{s}, x, \overline{t}) = \langle\langle 0, v\rangle, \overline{t}_1\rangle \wedge L(t) = k-1\ .
\end{aligned}$$

The function $\overline{\mathit{Incr}}$ iterates the function *Incr* in such a way that $\overline{\mathit{Incr}}(i, x, \overline{t}_0) = \langle c, \overline{t}_i\rangle$ and it is defined by primitive recursion:

$$\begin{aligned}
&\overline{\mathit{Incr}}(0, x, \overline{t}) = \langle 0, \overline{t}\rangle\\
&\overline{\mathit{Incr}}(i+1, x, \overline{t}) = \mathit{Incr}(\overline{t}_1, x, \overline{t}_1) \leftarrow \overline{\mathit{Incr}}(i, x, \overline{t}) = \langle c, \overline{t}_1\rangle\ .
\end{aligned}$$

The function f defined by nested simple recursion has thus a primitive recursive derivation by an explicit clausal definition:

$$f(z, x) = v \leftarrow \overline{\mathit{Incr}}(k^z, x, \sigma(z)) = \langle\langle 0, v\rangle, \overline{t}\rangle\ . \qquad \square$$

3.8 CL-definable functions. Already the elementary functions [Kal43,Ros82], which are but a tiny subclass of primitive recursive functions, are computationally unfeasible. It is generally accepted that the computationally *feasible* functions are the functions computable in polynomial time. The class P of such functions was characterized by Cobham [Cob65,Ros82] with the help of *recursion on notation* which goes from $2x$ and $2x+1$ to x. We have characterized P in [Vod94] with the help of pair recursion. The class P is a proper subset of elementary functions. We have decided to restrict CL to primitive recursive functions becuase they have simpler closure properties than the elementary functions or the functions in P. Primitive recursive functions have also a simple formal theory (see Sect. 4) for proving the properties of defined functions. The reader might wish to consult [Wai97] for a similar argument in favor of primitive recursive functions.

We say that a function is *CL-definable* if it has a clausal definition from previously CL-defined functions such that the clauses can be converted to an explicit or regular recursion with measure 3.1(1). If the definition is regular then also the measure function must be CL-defined (and so it must be into finite ordinals $\mathbb{N}$).

3.9 Theorem (Characterization of primitive recursive functions) *CL-definable functions are exactly the primitive recursive functions.*

Proof. That CL-definable functions are primitive recursive follows from Péter's theorem and from the reduction of regular definitions to nested simple recursion in Par. 3.6. Vice versa, the *initial* primitive recursive functions, i.e. the projection, successor, and zero functions, have explicit definitions $I_i^n(x_1, \ldots, x_i, \ldots, x_n) = x_i$, $S(x) = S(x)$, and $Z(x) = 0$ respectively. Functions defined by substitution have explicit definitions: $f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_m(\overline{x}))$ and functions defined by primitive recursion have a regular clausal definition with the measure $I_1^{n+1}(x, \overline{y}) = x$:

$$\begin{aligned}
&f(0, \overline{y}) = g(\overline{y})\\
&f(x+1, \overline{y}) = h(x, \overline{y}, f(x, \overline{y}))\ . \quad \square
\end{aligned}$$

3.10 Characterization of recursive functions. By permitting arbitrary ordinal-valued measure functions in regular definitions (even non-computable ones and without ordinal notations in natural numbers) we can define by explicit and regular recursive definitions exactly the *recursive* functions. That every recursive function can be thus defined can be seen from the clausal derivation of *regular minimalization*:

$$f(\overline{x}) = \mu_y[g(y, \overline{x}) = 0]$$

where for arbitrary $\overline{x}$ there is a y such that $g(y, \overline{x}) = 0$. For the clausal derivation of f we define an auxiliary function $\overline{f}$:

$$\begin{aligned}&\overline{f}(y, \overline{x}) = z \leftarrow \mu_{z \leq y}[g(z, \overline{x}) = 0] = z \wedge g(z, \overline{x}) = 0\\ &\overline{f}(y, \overline{x}) = \overline{f}(y+1, \overline{x}) \leftarrow \mu_{z \leq y}[g(z, \overline{x}) = 0] = z \wedge g(z, \overline{x}) > 0 \ .\end{aligned}$$

This is a regular clausal definition using *bounded minimalization* which is reducible to primitive recursion. The measure is: $m(y, \overline{x}) = f(\overline{x}) \dot{-} y$ because when the recursion takes place we must have $g(z, \overline{x}) > 0$ for all $z \leq y$ and so $y < f(\overline{x})$. The function f is then explicitly derived as $f(\overline{x}) = \overline{f}(0, \overline{x})$.

We do not prove here the converse that every regular clausal definition 3.1(1) is recursive in the functions applied in τ. We just suggest the proof by doing it for the Ackermann-Péter function. We redo the definition in Par. 2.6 and add a new argument z:

$$\begin{aligned}&\overline{A}(0, 0, y) = y + 2\\ &\overline{A}(0, x+1, y) = 0\\ &\overline{A}(z+1, 0, y) = y + 2\\ &\overline{A}(z+1, x+1, 0) = \overline{A}(z, x, 1)\\ &\overline{A}(z+1, x+1, y+1) = \overline{A}(z, x, a) \leftarrow \overline{A}(z, x+1, y) = a + 1\\ &\overline{A}(z+1, x+1, y+1) = 0 \leftarrow \overline{A}(z, x+1, y) = 0 \ .\end{aligned}$$

The argument z plays the role of a counter. If z is sufficiently large then $\overline{A}(z, x, y) = A(x, y) + 1$, if z is not large enough to count the number of recursions in A then $\overline{A}(z, x, y) = 0$ (note the second clause for $\overline{A}$). The function $\overline{A}$ is primitive recursive because it is defined by nested simple recursion. Moreover, we have $\exists z\, \overline{A}(z, x, y) = A(x, y) + 1$ as can be proved by transfinite induction with the measure $m(x, y) = \omega \cdot x + y$. Hence, we can define $\overline{m}(x, y) = \mu_z[\overline{A}(z, x, y) > 0]$ by regular minimalization and set $A(x, y) = \overline{A}(\overline{m}(x, y), x, y) \dot{-} 1$. Alternatively, the definition of A in Par. 2.6 has the measure $\overline{m}(x, y) < \omega$, and so A is primitive recursive in $\overline{m}$ which thus cannot be primitive recursive. The last shows the *collapse* of any measure function, no matter how large its ordinal range is, into a measure function $< \omega$ by a single regular minimalization.

4 Formal Arithmetic

4.1 Peano arithmetic. *Peano arithmetic* is obtained by the addition of *axioms of induction*:

$$\phi(0) \wedge \forall x(\phi(x) \rightarrow \phi(x+1)) \rightarrow \forall x \phi(x) \tag{1}$$

to the six axioms of *Robinson arithmetic*:

$$\begin{gathered}S(x) \neq 0\\ S(x) \wedge S(y) \rightarrow x = y\\ 0 + y = y\\ S(x) + y = S(x + y)\\ 0 \cdot y = 0\\ S(x) \cdot y = x \cdot y + y \ .\end{gathered}$$

The language $\mathcal{L}$ of both arithmetics consists of the symbols 0, S, $+$ and $\cdot$. Axioms of both arithmetics have the *standard model* $\mathcal{N}$ whose domain is $\mathbb{N}$ and the symbols of the language have the standard interpretation. We write $T \vdash \phi$ when ϕ is provable from axioms T. The reader is invited to use his favorite (complete) proof system. In the following discussion we state many properties of Peano arithmetic without proofs. The reader is advised to consult [HP93] for more details.

4.2 Fragments of Peano arithmetic. If we restrict the formulas $\phi(x)$ in axioms of induction 4.1(1) to the formulas of certain quantifier complexity we obtain *fragments* of Peano arithmetic.

The class of $\Sigma_0 = \Pi_0$-*formulas* is defined to consist of formulas with at most *bounded quantifiers* $\exists x \leq s\, \phi(x)$ and $\forall x \leq s\, \phi(x)$. Bounded quantifiers are abbreviations for the formulas $\exists x \exists z(x+z = s \wedge \phi(x))$ and $\forall x \forall z(x+z = s \rightarrow \phi(x))$ respectively (z is a new variable). Σ_{n+1}-*formulas* are the formulas of the form $\exists x_1 \cdots \exists x_k \phi$ where $\phi \in \Pi_n$ and Π_{n+1}-*formulas* are the formulas of the form $\forall x_1 \cdots \forall x_k \phi$ where $\phi \in \Sigma_n$ and $k \geq 1$.

We denote by $I\Sigma_{n+1}$ the fragment of Peano arithmetic whose axioms are axioms of Robinson arithmetic and the induction axioms for Σ_{n+1}-formulas.

Let T be a fragment of Peano arithmetic. We say that a formula ϕ is Σ_n (Π_n) *in* T if there is a formula $\psi \in \Sigma_n$ ($\psi \in \Pi_n$) such that $T \vdash \phi \leftrightarrow \psi$.

4.3 Witnessing theorem. Consider a Σ_1-formula $\phi(x_1, \cdots, x_n, y)$ with all of its free variables among the indicated ones and assume that a fragment T is such that

$$T \vdash \exists y\, \phi(x_1, \cdots, x_n, y)\ . \tag{1}$$

Then clearly, there is a *witness* to the existential quantifier, i.e. a function f such that for all numbers k_1, ..., k_n we have

$$\mathcal{N} \models \phi(S^{k_1}(0), \cdots, S^{k_n}(0), S^{f(k_1,\cdots,k_n)}(0))\ .$$

The formula ϕ has a form $\exists z_1 \cdots \exists z_m \phi_1$ where the Σ_0-formula ϕ_1 is constructed by primitive recursive operations. The witness, which for given x_1, ..., x_n looks for the least number v of the form $\langle y, z_1, \cdots, z_m \rangle$ whose projections make the formula ϕ_1 true and then yields $H(v)$, is clearly a recursive function.

The fragment which we have adopted as the formal system of CL is $I\Sigma_1$-arithmetic. We have chosen this fragment because the *witnessing* theorem for $I\Sigma_1$ [Kre52,Par70,Min73] says that every Σ_1-formula ϕ for which (1) holds has a primitive recursive witness. Moreover, a primitive recursive derivation of the witness can be mechanically extracted from the proof.

4.4 Recursive extensions. CL as a programming language allows incremental addition of new definitions of functions and predicates. This has a correspondence in the formal system where a fragment T of Peano arithmetic can be *extended* to the fragment T_1 by a *recursive extension.* For functions this means that when

$$T \vdash \forall \overline{x} \exists! y\, \phi(\overline{x}, y)$$

with $\phi \in \Sigma_1$ the language $\mathcal{L}_1$ of T_1 is obtained by adding a new n-ary function symbol f to the language of T. Axioms of T_1 are those of T plus the *defining* axiom for f:

$$f(\overline{x}) = y \leftrightarrow \phi(\overline{x}, y)\ . \tag{1}$$

With the addition of a new function symbol to T we uniquely *expand* the standard model $\mathcal{N}$.

Classes of formulas $\Sigma_n(f)$ and $\Pi_n(f)$ in the language $\mathcal{L}_1$ are defined similarly as the corresponding classes Σ_n and Π_n (see Par. 4.2).

The *graph* of f, i.e. the left-hand-side of the above equivalence, is given by a Σ_1-formula. T_1 clearly proves that the graph is equivalent to $\forall z(\phi(x_1, \cdots, x_n, z) \rightarrow y = z)$ which is easily shown to be Π_1 in T. This fact permits an *elimination* of all occurrences of f from formulas of $\mathcal{L}_1$ by repeatedly replacing $\psi_y[f(\overline{x})]$ by equivalent in T_1 formulas $\exists y(\phi(\overline{x},y)\wedge\psi)$ or $\forall y(\phi(\overline{x},y)\rightarrow\psi)$. The choice depends on the polarity of the occurrence to be eliminated. As a consequence every $\Sigma_1(f)$ ($\Pi_1(f)$) formula is a Σ_1 (Π_1) formula in T_1. For this reason T_1 is *conservative* over T, i.e. if for a formula ψ of $\mathcal{L}$ we have $T_1 \vdash \psi$ then also $T \vdash \psi$. In other words, recursive extensions do not add anything new to the extended theories except increased expressivity. In situations where there can be no confusion we will use the symbol $I\Sigma_1$ to denote also the axioms of recursive extensions. We can use introduced symbols in induction axioms and in the formulas introducing new symbols because they can be always eliminated without increasing the quantifier complexity.

4.5 Primitive recursive functions in $I\Sigma_1$. For every primitive recursive function f there is a recursive extension of $I\Sigma_1$ which proves the identities used in the derivation of f (i.e. those in the proof of Thm. 3.9) as theorems. Vice versa, if we extend $I\Sigma_1$ with suitable recursive identities for primitive recursive functions as axioms we can derive their defining axioms 4.4(1) as theorems. This equivalence has the advantage that the axioms in recursive extensions can be taken as open formulas (i.e. formulas without quantifiers).

The pairing function $\langle x,y\rangle$ and its projections H and T are primitive recursive. So the functions can be introduced into a recursive extension of $I\Sigma_1$ in such a way that their properties given in Par. 1.1 are theorems.

4.6 Pair induction. The principle of pair induction 1.1(1) for any Σ_1-formula ϕ can be proved in a recursive extension containing the pairing function. We reason in the extension, assume the antecedent of 1.1(1), and continue by induction on z with the formula $\forall x < z\,\phi(x)$ (which is Σ_1 in the extension). For the base case there is nothing to prove. If $x < z+1$ then we distinguish two cases. If $x = 0$ we have $\phi(x)$ from the assumption. Otherwise there are v and w such that $v, w < \langle v,w\rangle = x \leq z$ and so $\phi(v)$ and $\phi(w)$ by IH. Hence, $\phi\langle v,w\rangle$ from the assumption. Now, since $x < x+1$, we have $\phi(x)$ for any x.

4.7 Π_{n+1}-induction in $I\Sigma_{n+1}$. For every $\phi(x) \in \Pi_{n+1}$ the theory $I\Sigma_{n+1}$ proves the Π_{n+1}-induction principle 4.1(1). We reason in $I\Sigma_{n+1}$ and assume the antecedent of 4.1(1) as well as $\neg\forall x\phi(x)$. For some x we then have $\neg\phi(x)$ and we prove

$$z \leq x \rightarrow \neg\phi(x \dot{-} z)\ , \tag{1}$$

which is Σ_{n+1} in $I\Sigma_{n+1}$, by induction on z. For $z = 0$ this follows from $\neg\phi(x)$. If $z+1 \leq x$ then $z < x$ and $\neg\phi(x \dot{-} z)$ from IH. From $x \dot{-} z > 0$ we have $x \dot{-}(z+1)+1 = x \dot{-} z$ and so $\neg\phi(x \dot{-}(z+1))$ from the inductive assumption of 4.1(1). By taking $z = x$ we get from (1) a contradiction $\neg\phi(0)$.

4.8 Clausal definitions in $I\Sigma_1$. Every clausal definition admissible in CL can be mechanically transformed into the form 3.1(1). The demonstration that the two forms are equivalent does not require induction and can be formalized in any fragment of arithmetic.

We wish to prove the property 3.1(1) as a theorem in a suitable recursive extension of $I\Sigma_1$. This is a technically quite demanding proof because we have to make sure that the inductions needed in the reduction of 3.1(1) first to nested simple recursion in Par. 3.6 and then in the proof of Péter's theorem 3.7 are available. This is indeed the case and the formalization of Péter's theorem goes through. Obviously, we cannot do the proof here and so we just state the formalized version of Peter's theorem:

4.9 Formalized Péter's theorem. *If the function f is defined by nested simple recursion from primitive recursive functions then there is a recursive extension of $I\Sigma_1$ which proves the identities 3.6(1)(2) as theorems.* □

4.10 Proof obligations. Before a recursive clausal definition is accepted its regularity must be proved in CL (this is called a *proof obligation*). The proof turns the semantic condition of regularity (in general undecidable) to a syntactic one (decidable). It should be clear that by replacing the semantic condition by a syntactic one we do not lose any primitive recursive function extensionally, we may just lose some definitions intensionally, i.e. as rewriting rules. It can namely happen as a consequence of Gödel's incompleteness theorem that $I\Sigma_1$ is not strong enough to prove a condition of regularity which is true in $\mathcal{N}$.

4.11 Ackermann-Péter function revisited. By the witnessing theorem we have $I\Sigma_1 \nvdash \exists z\,\overline{A}(z,x,y) > 0$ for the function $\overline{A}$ from Par. 3.10 because the primitive recursive witnessing function $\overline{m}(x,y)$ would make also the Ackermann-Péter function A primitive recursive. However, we can introduce A into $I\Sigma_2$ by proving

$$\forall y \exists z\,\overline{A}(z,x,y) > 0 \tag{1}$$

by Π_2-induction on x. For $x = 0$ we take $z = 0$. For $x+1$ we proceed by induction on y with the Σ_1-formula $\exists z \overline{A}(z, x+1, y)$. In the base case we get a z_1 s.t. $\overline{A}(z_1, x, 1) > 0$ from the outer IH and set $z = z_1 + 1$. In the inductive case we get a z_1 s.t. $\overline{A}(z_1, x+1, y) > 0$ from the inner IH and a z_2 s.t. $\overline{A}(z_2, x, \overline{A}(z_1, x+1, y) \dot{-} 1) > 0$ from the outer IH. We then set $z = \max(z_1, z_2) + 1$.

4.12 Theorem of Parsons. Because Π_1-induction is available in $I\Sigma_1$ we have $I\Sigma_1 \vdash \forall y 2.3(1)$ for the second definition of the minimum function in Par. 2.3. On the other hand, Π_2-induction is not available in $I\Sigma_1$ and we cannot directly prove the property 2.4(1) needed for the extraction of the witnessing function *Rta*. Fortunately, a theorem by Parsons [Par70,Bus94,Sie91] helps us here. The theorem says that

> *Robinson arithmetic plus* Π_{n+2}-rules *inferring* $\forall x \phi(x)$ *from* $\phi(0)$ *and* $\forall x(\phi(x) \to \phi(x+1))$ *for any* $\phi \in \Pi_{n+2}$ *are* Π_{n+2}*-conservative over* $I\Sigma_{n+1}$.

In our special case $n = 1$ this means that we can derive the Π_2-formula 2.4(1) by a Π_2-induction rule and then use the conservativity to derive the same formula in $I\Sigma_1$. This is possible because the proof of 2.4(1) is without any side assumptions. Note that this is not the case in the proof of 4.11(1) because the inner induction uses the outer IH as a side assumption and thus cannot be turned into a rule.

4.13 Formal proof system of CL. We will outline the proof of a liberalization of the case $n = 1$ of Parsons theorem in Thm. 4.15. As this is a proof-theoretic argument we now present the proof system of CL which are *signed tableaux* of Smullyan [Smu68]. The only difference is that we write $\phi*$ instead of Smullyan's $F\phi$ and ψ instead of $T\psi$. We can view formulas $\phi*$ as *goals* to be proved under the *assumptions* given by the formulas ψ.

A branch of a signed tableau can be viewed as a *sequent* where the conjunction of assumptions implies the disjunction of goals. This interpretation of signed tableaux gives them a flavor of *natural deduction* which can be readily translated into English description of proofs (as it is the case in CL). We just mention here that this aspect of signed tableaux seems to have escaped the workers in the field and CL is probably the first formal system with such an interpretation.

If we wish to prove a formula ϕ we write it down as a goal $\phi*$ and then proceed to build a tableau as a downwards growing tree with the following expansion rules.

Propositional rules:

$$\frac{\phi \to \psi*}{\begin{array}{c}\phi\\ \psi*\end{array}} \quad \frac{\phi \vee \psi*}{\begin{array}{c}\phi*\\ \psi*\end{array}} \quad \frac{\phi \wedge \psi}{\begin{array}{c}\phi\\ \psi\end{array}} \quad \frac{\phi \wedge \psi*}{\phi* \mid \phi*} \quad \frac{\phi \vee \psi}{\phi \mid \psi} \quad \frac{\phi \to \psi}{\psi \mid \phi*} \quad \frac{\neg\phi*}{\phi} \quad \frac{\neg\phi}{\phi*}\,.$$

Equational rules:

$$\frac{}{s = s} \quad \frac{s = t}{t = s} \quad \frac{s = t \quad t = u}{s = u} \quad \frac{s_1 = t_1 \cdots s_n = t_n}{f(s_1, \cdots, s_n) = f(t_1, \cdots, t_n)} \quad \frac{s_1 = t_1 \cdots s_n = t_n \;\; R(s_1, \cdots, s_n)}{R(t_1, \cdots, t_n)}\,.$$

Quantifier rules ($\dagger$: y is an eigen-variable):

$$\frac{\forall x\phi(x)*}{\phi(y)*}\dagger \quad \frac{\exists x\phi(x)}{\phi(y)}\dagger \quad \frac{\exists x\phi(x)*}{\phi(s)*} \quad \frac{\forall x\phi(x)}{\phi(s)} .$$

Axiom ($\dagger$) and **Cut** ($\ddagger$) **rules**:

$$\frac{}{\phi}\dagger \qquad \frac{}{\phi \mid \phi*}\ddagger .$$

$I\Sigma_1$**-induction rule for** $\exists\overline{x}\phi(z,\overline{x})$ where $\phi(z,\overline{x}) \in \Sigma_0$, z, $\overline{y}$ eigen-variables:

$$\frac{\exists\overline{x}\phi(s,\overline{x})*}{\exists\overline{x}\phi(0,\overline{x})* \;\Big|\; \begin{array}{c}\phi(z,\overline{y})\\ \exists\overline{x}\phi(z+1,\overline{x})*\end{array}} .$$

Axioms are the axioms of Robinson arithmetic as well as the defining axioms for introduced function and predicate symbols. A tableau is *closed* if every of its branches is *closed*, i.e. contains ϕ and $\phi*$ for some formula.

By a straightforward argument, similar to the one discussed in [KV95], we can show that a Σ_1-induction axiom can be *eliminated* by replacing it with a Σ_1-induction rule. Vice versa, every Σ_1 induction rule can be replaced by a Σ_1-induction axiom. This means that the CL proof system is equivalent to any complete proof system with $I\Sigma_1$ axioms, i.e. we have $I\Sigma_1 \vdash \phi$ iff there is a closed tableau for $\phi*$.

By the standard *cut elimination* argument adapted to tableaux (see [KV95]) we can eliminate from a closed tableau all cuts except the ones *anchored* to an induction rule, i.e. such where the cut formula $\phi*$ is the premise of an induction rule.

4.14 Weak Π_2-induction rule. Tableau expansion rules introduced in Par. 4.13 are local in the sense that one expansion rule does not interfere with another one. The following rules are not such. Let $\phi(z,\overline{x},\overline{y}) \in \Sigma_0$. The *weak Π_2-induction rule for the formula* $\forall\overline{x}\exists\overline{y}\phi(z,\overline{x},\overline{y})$ is

$$\frac{\exists\overline{y}\phi(s,\overline{t},\overline{y})*}{\exists\overline{y}\phi(0,\overline{u},\overline{y})* \;\Big|\; \begin{array}{c}\forall\overline{x}\exists\overline{y}\phi(z,\overline{x},\overline{y})\\ \exists\overline{y}\phi(z+1,\overline{u},\overline{y})*\end{array}}$$

where z and $\overline{u}$ are eigen-variables. We attach an additional non-local condition to this rule because otherwise the rule would be equivalent to the corresponding Π_2-induction axiom. The condition is that every tableau expansion by a weak Π_2-induction rule as well as **every** expansion of an induction rule below it (including Σ_1-induction rules) may have at most *weak side formulas*. We say that an induction rule has a *side formula* ϕ if ϕ appears in the branch above the conclusion and it is a premise of a rule used below this conclusion. *Weak* side formulas are Σ_1-formulas in goals, Π_1-formulas in assumptions and Σ_0-formulas anywhere.

4.15 Theorem (Elimination of weak Π_2-induction rules) *Theorems proved by tableaux with weak Π_2-induction rules are provable in $I\Sigma_1$.*

Proof. In the tableau proof p of the following Witnessing lemma we may assume without loss of generality that the bounded quantifiers are eliminated by going into a suitable extension, that adjacent quantifiers of the same kind are contracted by pairing, that the cuts are anchored to induction, i.e. they are with Σ_1-formulas, and that all axioms are open. All formulas named by Greek letters can be thus assumed to be open. The idea of the proof is similar to the very terse outline given in [Bus94] but it is done here with more detail because unlike [Bus94] we have the necessary primitive recursive apparatus developed. In the Witnessing lemma we show just one weak side formula corresponding to a Σ_1-goal, remaining side formulas are indicated by $\cdots$ but

the reader should bear in mind that the witness found for the shown side formula is to be found also for every not shown quantified formula in exactly the same way.
Witnessing lemma: *If p is a closed tableau with weak Π_2-induction rules for assumptions and goals corresponding to the sequent*

$$\forall x \exists y \phi_1(\overline{w}_1, x, y) \rightarrow \cdots \exists v \psi(\overline{w}, v) \cdots \tag{1}$$

where $\overline{w}_1$ is a subsequence of $\overline{w}$ then a suitable recursive extension of $I\Sigma_1$ proves for a new function symbol f_1:

$$\forall x \phi_1(\overline{w}_1, x, f_1(\overline{w}_1, x)) \rightarrow \cdots \psi(\overline{w}, t(f_1(\overline{w}_1, \cdot), \overline{w})) \cdots \ . \tag{2}$$

The proof is similar to the proof of lemma needed for the $I\Sigma_1$ witnessing theorem and it is by induction on the construction of p. We do just two new cases. If the first expansion in p is by the instantiation $\exists y \phi_1(\overline{w}_1, s, y)$ of the Π_2-assumption in (1) then we may assume without loss of generality that the next expansion introduces the eigen-variable y: $\phi(\overline{w}_1, s, y)$. We use IH with the last formula as a new assumption and y as a new parameter and so a suitable extension of $I\Sigma_1$ proves

$$\forall x \phi_1(\overline{w}_1, x, f_1(\overline{w}_1, x)) \wedge \phi_1(\overline{w}_1, s, y) \rightarrow \cdots \psi(\overline{w}, t_1(f_1(\overline{w}_1, \cdot), \overline{w}, y)) \cdots \ .$$

By setting y to $f_1(\overline{w}_1, s)$ we may discharge the second assumption and get (2).

Let the first expansion in p be by a weak Π_2-induction rule for the formula $\forall x \exists y \phi(\overline{w}, z, x, y)$ with the premise $\exists y \phi(\overline{w}, s_1(\overline{w}), s_2(\overline{w}), y)$. Without loss of generality we may assume that the premise is not used in p and so we remove it from the side formulas. We also remove the Π_2-assumption $\forall x \exists y \phi_1$ because it cannot be used in p. For the base-case branch of p we add a new goal $\exists y \phi(\overline{w}, 0, x, y)$ with the eigen-variable x. A suitable extension of $I\Sigma_1$ proves by IH:

$$\phi(\overline{w}, 0, x, s_3(\overline{w}, x)) \vee \cdots \psi(\overline{w}, t_1(\overline{w}, x)) \cdots \ .$$

For the inductive-case branch of p we add a new Π_2-assumption $\forall x \exists y \phi(\overline{w}, z, x, y)$ and a new goal $\exists y \phi(\overline{w}, z+1, x, y)$ for two eigen-variables z and x. A suitable extension of $I\Sigma_1$ proves by IH:

$$\forall x \phi(\overline{w}, z, x, f(\overline{w}, z, x)) \rightarrow \phi(\overline{w}, z+1, x, s_4(f(\overline{w}, z, \cdot), \overline{w}, z, x)) \vee \cdots \psi(\overline{w}, t_2(\overline{w}, z, x)) \cdots \ . \tag{3}$$

We extend the theory unifying both theories from IH by f defined by nested simple recursion:

$$f(\overline{w}, 0, x) = s_3(\overline{w}, x)$$
$$f(\overline{w}, z+1, x) = s_4(f(\overline{w}, z, \cdot), \overline{w}, z, x) \ .$$

This is a slight misuse of notation because we use the same symbol f as the one in (3) but nothing happens because we now substitute in it the new symbol for the old one. Formula (3) is now Σ_1 in the current extension and by the Witnessing theorem for $I\Sigma_1$ we get

$$\phi(\overline{w}, z, h(\overline{w}, z, x), f(\overline{w}, z, h(\overline{w}, z, x))) \rightarrow \phi(\overline{w}, z+1, x, s_4(f(\overline{w}, z, \cdot), \overline{w}, z, x)) \vee \cdots \psi(\overline{w}, t_2(\overline{w}, z, x)) \cdots$$

for a function $h(\overline{w}, z, x)$. We define by (un)nested simple recursion the function g:

$$g(\overline{w}, 0, x) = t_1(\overline{w}, x)$$
$$g(\overline{w}, z+1, x) = t_2(\overline{w}, z, x) \leftarrow \phi(\overline{w}, z, h(\overline{w}, z, x), f(\overline{w}, z, h(\overline{w}, z, x)))$$
$$g(\overline{w}, z+1, x) = g(\overline{w}, z, h(\overline{w}, z, x)) \leftarrow \neg\phi(\overline{w}, z, h(\overline{w}, z, x), f(\overline{w}, z, h(\overline{w}, z, x))) \ .$$

By a not too difficult Π_1-induction on z we prove

$$\forall x \, (\phi(\overline{w}, z, x, f(\overline{w}, z, x)) \vee \cdots \psi(\overline{w}, g(\overline{w}, z, x)) \cdots) \ .$$

We now substitute s_1 for z and s_2 for x and get (2). This ends the proof of the Witnessing lemma.

Now, given a closed tableau p with weak Π_2-induction rules, we replace all top uses of such rules by tableaux without the Π_2-rules. Each such tableau is obtained from the Witnessing lemma without a Π_2 assumption but with all weak side formulas used in the corresponding Π_2-rule. $\square$

4.16 Induction with measure. The schema of recursion with measure 3.1(1) calls for the corresponding schema of *induction with measure*:

$$\forall x\,(\forall y(m(y) < m(x) \rightarrow \phi(y)) \rightarrow \phi(x)) \rightarrow \forall x \phi(x)\ .$$

We just state here without a proof that this can be reduced into a Π_1-induction formula for $\phi \in \Pi_1$ and into a weak Π_2-induction rule for $\phi \in \Pi_2$.

5 Conclusion

We have hopefully demonstrated that CL is a simple language and proof system with natural semantics in $\mathbb{N}$. Our two year's experience with teaching CL shows that the first and second year students have no problems defining functions in CL and seem to enjoy doing the formal proofs of their properties. On the other hand, the characterization of CL requires deep mathematical knowledge and relies on two almost forgotten theorems of logic. Obviously, we do not teach the characterization in the introductory courses.

The interested reader will find in our homepage the executable file of a PC implementation of CL as well as a partially finished text on CL which will eventually contain in detail the material of this paper. We are also preparing lecture notes for the courses we teach with CL.

References

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

[Cob65] A. Cobham. The intristic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science II*, pages 24–30. North-Holland, 1965.

[Col91] L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991.

[Dav85] M. Davis. *Computability and Unsolvability*. McGraw-Hill, second edition, 1985.

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

[Kal43] L. Kalmár. A simple example of an undecidable arithmetical problem (Hungarian with German abstract). *Matematikai és Fizikai Lapok*, 50:1–23, 1943.

[Kre52] G. Kreisel. On the interpretation of non-finitist proofs II. *Journal of Symbolic Logic*, 17:43–58, 1952.

[KV95] J. Komara and P. J. Voda. Syntactic reduction of predicate tableaux to propositional tableaux. In P. Baumgartner, R. Haehnle, and J. Posegga, editors, *Proceedings of TABLEAUX '95*, number 918 in LNAI, pages 231–246. Springer Verlag, 1995.

[Min73] G. Mints. Quantifier-free and one-quantifier systems. *Journal of Soviet Mathematics*, 1:71–84, 1973.

[Par70] C. Parsons. On a number-theoretic choice schema and its relation to induction. In *Intuitionism and Proof Theory: proceedings of the summer conference at Buffalo N.Y. 1968*, pages 459–473. North-Holland, 1970.

[Pét32] R. Péter. Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion. *Mathematische Annalen*, 110:612–632, 1932.

[Pét67] R. Péter. *Recursive Functions*. Academic Press, 1967.

[Ros82] H. E. Rose. *Subrecursion: Functions and Hierarchies*. Number 9 in Oxford Logic Guides. Clarendon Press, Oxford, 1982.

[Sie91] W. Sieg. Herbrand analyses. *Archive for Mathematical Logic*, 30:409–441, 1991.

[Smu68] R. Smullyan. *First Order Logic*. Springer Verlag, 1968.

[Vod94] 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, pages 324–338. Springer Verlag, 1994.

[Wai97] S. S. Wainer. Basic proof theory and applications to computation. In H. Schwichtenberg, editor, *Logic of Computation*, volume 157 of *Series F: Computer and Systems Sciences*, pages 349–394. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 25–August 6, 1995, Springer Verlag, 1997.