Extraction of Efficient Programs in $I\Sigma_1$-arithmetic

Ján Komara and Paul J. Voda

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

Abstract. *C*lausal *L*anguage (CL) is a declarative programming and verifying system used in our teaching of computer science. CL is an implementation of, what we call, $PR{+}I\Sigma_1$ paradigm (primitive recursive functions with $I\Sigma_1$-arithmetic). This paper introduces an extension of $I\Sigma_1$-proofs called *extraction* proofs where one can extract from the proofs of Π_2-specifications primitive recursive programs as efficient as the hand-coded ones. This is achieved by having the programming constructs correspond exactly to the proof rules with the computational content.

1 Introduction

The class of effectively computable functions over natural numbers coincides by the thesis of Church with recursive functions as defined by Herbrand-Gödel style equations. We use Herbrand-Gödel-like recursive equations because they offer the programming comfort with almost unrestricted kinds of recursion and the computation of recursive equations by reductions permits a fine degree of control over the length of reduction sequences. We interpret the recursive equations into natural numbers because the concept of truth in $\mathbb{N}$ is well understood even by beginners and the theory of recursive functions and arithmetic offers a firm natural semantical background.

A possible objection by computer scientists that the domain $\mathbb{N}$ means unpleasant coding (arithmetization) of the rich set of data structures used in programming is answered by coding into $\mathbb{N}$ in the style of LISP with a pairing function (instead of *cons*). We obtain a degree of comfort as it is known from declarative prog. languages (say Haskell). The examples in Sect. 3 should convince the reader. An objection that the coding may prolong the length of reduction sequences is answered by computing in mixed representation (see Sect. 2).

We are interested not only in a programming language but also in the verification of its programs. By restricting ourselves to $\mathbb{N}$ we can use the simplest of formal theories: Peano arithmetic. Since the computationally feasible functions are a proper subset of elementary functions and the latter are but a tiny subset of primitive recursive functions (PR), we restrict the strength of our system to $I\Sigma_1$-arithmetic where the induction formulas are Σ_1 (see [HP93]). The provably recursive functions of $I\Sigma_1$ are exactly the primitive recursive functions. We can thus call our approach the *PR+$I\Sigma_1$ programming/verifying paradigm.* We will briefly discuss our computer implementation of the paradigm in the form of the system CL (*C*lausal *L*anguage) in the conclusion of this paper.

In order to prove properties of functions defined by a rich variety of recursion schemes we need a rich variety of induction schemes. The schemes are needed for the programming/verifying comfort but their power does not exceed mathematical induction with Σ_1-formulas. Our results are of metamathematical (proof-theoretical) character because we do a rigorous development of a usable programming language (usable at least in the teaching if not yet in real life) within the theory of programming languages. This means in our case that we investigate how to express programs as primitive recursive functions and how to do the proofs of their properties in $I\Sigma_1$-arithmetic.

Beside the $I\Sigma_1$-characterization of our language in Sect. 4, the main contribution of this paper is the introduction of a special kind of *extraction proofs* from which efficient programs for functions satisfying Π_2-specifications can be extracted. The main idea is that the user, by deciding which rules have computational content and which not, can control the efficiency of the extracted witnessing function exactly as if he first hand-coded it and then proved that it satisfies the specifications. The goal is achieved by having the programming constructs correspond exactly to the proof rules with the computational content. This contrasts with the approach in systems such as PX [HN88] and MINLOG [BBS+98] where the decision which rules should have computational content is automatic.

2 Primitive Recursive Functions and $I\Sigma_1$-arithmetic

We now give a brief overview of recursion-theoretic semantics so the reader can see how the proofs of Π_2-specifications which will be discussed in Sect. 3 can control the efficiency of extracted programs (for details see [Vod95,KV99,Vod00]). Primitive recursive functions are discussed in detail for instance in [Pét67,Ros82].

2.1 Mixed representation of natural numbers. Every positive number x can be uniquely written in a form $x = \sum_{i<n} d_i{\cdot}2^i$ as a sequence $d_{n-1}d_{n-2}\ldots d_1d_0$ of *dyadic digits* $1 \le d_i \le 2$ and we have $n = \Theta(\log(x))$. Every natural number x can be obtained in the *dyadic representation* by finitely many applications to the constant 0 of *dyadic successor* functions $x\mathbf{1} = 2{\cdot}x{+}1$ and $x\mathbf{2} = 2{\cdot}x{+}2$: $0\mathbf{d}_{n-1}\cdots\mathbf{d}_1\mathbf{d}_0$.

For a suitable *pairing* function (x, y) we have

$$\begin{gathered}(x_1, y_1) = (x_2, y_2) \to x_1 = x_2 \wedge y_1 = y_2\\ x < (x, y) \wedge y < (x, y)\\ x = 0 \vee \exists v \exists w\, x = (v, w)\ .\end{gathered}$$

We require that for the *pair size* function $|x|$ satisfying $|0| = 0$ and $|(x, y)| = |x| + |y| + 1$ we have $|x| = \Theta(\log(x))$. Note that from the second property we get $0 \neq (x,y)$ and so every natural number can be uniquely obtained in the *pair representation* by finitely many applications of (x,y) to the constant 0. Thus $\mathbb{N}$ can be identified with the S-expressions of LISP (with the single atom $nil = 0$). When there can be no confusion of pairing with the comma separating arguments of functions we abbreviate (x,y) to x,y. We also write x,y,z for $x,(y,z)$. To every finite sequence of numbers $x_1 \dots x_n$ there is exactly one natural number, namely $x_1,\dots,x_n,0$, coding the sequence as a *list*.

Mixed numerals are terms obtained from 0 by dyadic successors and pairing. This *mixed representation* of $\mathbb{N}$ is not unique but permits the mixed mode computation of arithmetic and symbolic operations without unnecessary conversions between dyadic and pair representations (provided the definitions are well-typed).

2.2 Recursive definitions. We need three variable binding term operators, respectively called *let*, *dyadic discrimination*, and *pair discrimination* terms. The terms bind the indicated variables in the indicated positions:

$$\begin{aligned}
L_y(x,\tau[y]) &= \tau[x] \\
D_v(x,y,\tau_1[v],\tau_2[v]) = z &\leftrightarrow x = 0 \land z = y \lor \\
&\qquad \exists v(x = v\mathbf{1} \land z = \tau_1[v]) \lor \exists v(x = v\mathbf{2} \land z = \tau_2[v]) \\
P_{v,w}(x,y,\tau[v,w]) = z &\leftrightarrow x = 0 \land z = y \lor \exists v \exists w(x = v,w \land z = \tau[v,w]) \ .
\end{aligned}$$

Recursive terms are constructed from variables and 0 by the three operators and by applications of functions $x\mathbf{1}$, $x\mathbf{2}$, (x,y), as well as of *defined* partial functions. For every recursive term $\tau[f;\vec{x}]$ we let the function symbol $\lambda\vec{x}.\tau$ to denote the least partial function f solving the functional equation $f(\vec{x}) \simeq \tau[f;\vec{x}]$. The equation is a *recursive definition* of f. Note that $\lambda\vec{x}.\tau[f,\vec{x}]$ is not the standard lambda notation because the (meaningless) *recursive* function symbol f can be applied in τ. The function symbols $\lambda\vec{x}.\tau[f;\vec{x}]$ bind the variables $\vec{x}$ as well as applications of f. A recursive term is *closed* if it contains no free variables and no free applications of the recursive symbol f.

The partial functions definable by recursive definitions are exactly the partial recursive functions. The style of the definitions is basically that of Herbrand-Gödel recursive equations. We present the recursive definitions in Sect. 3 in the form of *clausal definitions* which are only *unfolded* recursive definitions. Clausal definitions are used to increase the readability.

2.3 Computation by reductions. The reason for our recursive terms is that one can obtain as efficient reduction (computation) sequences as one wishes. We reduce a closed recursive term τ to a mixed numeral ρ, in writing $\tau \blacktriangleright \rho$, by repeatedly locating in it the leftmost *redex* and rewriting it by its *contractum*. *Let* redexes (and contracta) are $L_y(\rho,\tau[y]) \blacktriangleright \tau[\rho]$, *pair* redexes are $P_{v,w}(0,\tau_1,\tau_2[v,w]) \blacktriangleright \tau_1$ and $P_{v,w}(\rho,\tau_1,\tau_2[v,w]) \blacktriangleright \tau_2[\rho_1,\rho_2]$ where ρ is a mixed numeral and in the last redex we have $\rho = \rho_1,\rho_2$. Note that this may mean a conversion if the outermost application of ρ is not pairing. The *dyadic* redexes involving D_v are similar. *Lambda* redexes are $(\lambda\vec{x}.\tau[f;\vec{x}])(\vec{\rho}) \blacktriangleright \tau[\lambda\vec{x}.\tau;\vec{\rho}]$ where $\vec{\rho}$ is an n-tuple of mixed numerals. Note that the mixed numerals are irreducible.

The denotational semantics is given by recursive definitions and the operational one by reductions. Both semantics coincide because for a partial function $f = \lambda\vec{x}.\tau$ we have $f(\vec{x}) \simeq y$ iff $(\lambda\vec{x}.\tau)(\vec{\rho}) \blacktriangleright \rho_0$ for some mixed numerals $\vec{\rho}$ denoting the corresponding numbers $\vec{x}$ and ρ_0 denoting y. Note that the function symbol $\lambda\vec{x}.\tau$ extensionally denotes a partial function f while intensionally it is a *program* (algorithm) for the computation of f.

2.4 $I\Sigma_1$-arithmetic [HP93]**.** Our proof system for $I\Sigma_1$-arithmetic is based on positive, i.e. non-refutational, tableaux (see [Smu68,KV99]). We mark in the tableaux shown in Sect. 3 the *goal* formulas as $\phi*$ and leave the *assumption* formulas unmarked. We work in recursive extensions of $I\Sigma_1$ (see [Sho67]); also denoted by the same symbol $I\Sigma_1$. We have also a rich set of admissible rules (see Sect. 3) for the proofs of properties of recursively defined functions.

The *graph* $f(\vec{x}) \asymp y$ of every partial recursive function $f = \lambda\vec{x}.\tau$ is Σ_1-definable in $I\Sigma_1$ and we have $I\Sigma_1 \vdash f(\vec{x}) \simeq \tau[f;\vec{x}]$. The last means $I\Sigma_1 \vdash f(\vec{x}) \asymp y \leftrightarrow \tau[f;\vec{x}] \asymp y$ where $\tau \asymp y$ stands for the graph of the partially denoting term τ. Graphs of terms are defined in the obvious way such that for every closed recursive term τ we have: $\tau \asymp y$ iff $\tau \blacktriangleright \rho$ for a mixed numeral ρ denoting y.

A recursive function $f = \lambda\vec{x}.\tau$ is *provably recursive in $I\Sigma_1$ (total, terminating)* if $I\Sigma_1 \vdash \exists y\, f(\vec{x}) \asymp y$. We then have $I\Sigma_1 \vdash f(\vec{x}) = \tau[f;\vec{x}]$ and the graph of f is also Π_1-definable in $I\Sigma_1$. It is well-known [Kre52]) that the provably recursive functions in $I\Sigma_1$ are exactly the primitive recursive functions.

2.5 Satisfying Π_2-specifications. We wish to find a *witness* function $f(\vec{x})$ for a *Π_2-specification* sentence $\forall\vec{x}\,(\psi[\vec{x}] \to \exists y\phi[\vec{x},y])$ which is possibly under assumption $\psi[\vec{x}]$. Instead of proving the specification sentence we will prove by an *extraction* proof its *witnessing* formula:

$$I\Sigma_1 \vdash_e \psi[\vec{x}] \to \phi[\vec{x}, f(\vec{x})] \ . \tag{1}$$

Here $\vdash_e$ stands for *extraction provability* and the symbol $f(\vec{x})$ should be understood as an 'unknown' which obtains a value in the extraction proof by a *definition* $f(\vec{x}) := \tau$. It will be clear from the examples in Sect. 3 that given an extraction proof of (1) we can primitively recursively find a term τ and a (standard) proof of

$$I\Sigma_1 + \forall\vec{x}\, f(\vec{x}) \simeq \tau \vdash \psi[\vec{x}] \to \exists y(\phi[\vec{x},y] \land f(\vec{x}) \asymp y) \ . \tag{2}$$

The proof of (2) decomposes into proofs of the *partial correctness* formula $\psi[\vec{x}] \land f(\vec{x}) \asymp y \to \phi[\vec{x},y]$ and of the *termination* formula $\psi[\vec{x}] \to \exists y\, f(\vec{x}) \asymp y$. That partial functions are needed in order to obtain efficiency will be seen in Par. 3.6.

2.6 Computational content of extraction proof rules. We can use in extraction proofs four kinds of proof rules involving the unknowns which correspond exactly to the kinds of programming constructs allowed in recursive definitions. By suitable applications of rules in an extraction proof of the witnessing formula we can guide the construction of the function to be extracted in exactly the same way as if we did the definition by hand. The rules are:

extraction rule:	corresponding standard proof rule:
recursion	induction
discrimination	cut
assignment $\tau =: x$	use of theorem $\exists x\, \tau = x$
definition $f(\vec{\rho}) := \rho$	consequence of the defining axiom $f(\vec{x}) \simeq \tau$

Standard tableau rules are permitted in a branch above an application of a definition rule only if they are devoid of *computational content* affecting termination of f. Below a definition rule (when a clause of a definition was completed) the branch can closed by unrestricted tableau rules. Rules without computational content affecting termination include the rule of replacement of equals (Leibnitz rule), term simplification rules by using open $I\Sigma_1$-theorems, propositional simplification rules, and quantifier rules (see Par. 3.6). Such rules may introduce no new variables into the definition being constructed.

Standard propositional tableau rules have in general computational content affecting termination and can be used in branches above definition rules only when the $\vdash_e$-proof checker is able to determine that they are used safely, i.e. without affecting termination. In particular, all non-commented uses of *assumption* rules $(\phi_1 \to \phi_2)* \,/\, \phi_1, \phi_2*$ given in the examples of Sect. 3 are safe.

3 Examples of Extraction of Programs in $I\Sigma_1$-arithmetic

3.1 Discrimination on predicates. Assume that $I\Sigma_1$ proves that for every $\vec{x}$ exactly one of $R_1(\vec{x}), \ldots, R_n(\vec{x})$ holds. We can then use the *discrimination* (case analysis) rule as shown in the following on the left:

$$R_1(\vec{x}) \quad R_2(\vec{x}) \quad \cdots \quad R_n(\vec{x})$$

$$\begin{array}{ccccc} & & \phi[\vec{x}, f(\vec{x})]* & & \\ R_1(\vec{x}) & & R_2(\vec{x}) & \cdots & R_n(\vec{x}) \\ \vdots & & \vdots & & \vdots \\ f(\vec{x}) := \tau_1[\vec{x}] & & f(\vec{x}) := \tau_2[\vec{x}] & & f(\vec{x}) := \tau_n[\vec{x}] \end{array}$$

The rule is applied on the right. The extracted witnessing function f has then the following definition:

$f(\vec{x}) = \tau_1[\vec{x}] \leftarrow R_1(\vec{x})$
$f(\vec{x}) = \tau_2[\vec{x}] \leftarrow R_2(\vec{x})$
$\cdots$
$f(\vec{x}) = \tau_n[\vec{x}] \leftarrow R_n(\vec{x})$.

Consider the minimum function $\min(x, y)$ with the specification: $\forall x \forall y \exists m \big(m \leq x \land m \leq y \land (m = x \lor m = y)\big)$. The proof of the witnessing formula (1) starts with the *dichotomy* discrimination on $x \leq y$ and $x > y$ where in each of the branches the obvious definition rules for $\min(x, y)$ close the tableau:

$$\begin{array}{ccc} & \min(x,y) \leq x \land \min(x,y) \leq y \land (\min(x,y) = x \lor \min(x,y) = y)* \quad (1) & \\ x \leq y & & x > y \\ \min(x,y) := x & & \min(x,y) := y \end{array}$$

The extracted definition is:

$\min(x, y) = x \leftarrow x \leq y$
$\min(x, y) = y \leftarrow x > y$.

3.2 Discrimination on patterns. Assume that $I\Sigma_1$ proves that for every x exactly one of $\exists\vec{y}_1\, z = \rho_1[\vec{y}_1]$, … , $\exists\vec{y}_n\, z = \rho_n[\vec{y}_n]$ holds. Here the terms $\rho_i[\vec{y}_i]$ are *patterns* and must be such that when $z = \rho_i[\vec{y}_i]$ holds then the numbers $\vec{y}_i$ are unique and can be primitively recursively obtained from z. For every term τ we can then use a rule *discriminating on patterns*:

$$\tau = \rho_1[\vec{y}_1] \qquad \tau = \rho_2[\vec{y}_2] \qquad \cdots \qquad \tau = \rho_n[\vec{y}_n]$$

where the variables $\vec{y}_i$ are new, i.e. *eigenvariables*.

For instance, the 0-1 valued *signum* function $sg(x)$ has the specification: $\forall x \exists s(x = 0 \wedge s = 0 \vee x > 0 \wedge s = 1)$ and it is extracted with the clauses $sg(0) = 0$ and $sg(y + 1) = 1$ from the following proof:

$$\begin{array}{c} x = 0 \wedge sg(x) = 0 \vee x > 0 \wedge sg(x) = 1\, * \\ \begin{array}{cc} x = 0 & x = y + 1 \\ sg(x) := 0 & sg(x) := 1 \end{array} \end{array}$$

by a rule discriminating on patterns $x = 0$ and $\exists y\, x = y + 1$.

3.3 Monadic representation of natural numbers. Every natural number is obtained in *monadic* representation by finitely many applications of the (monadic) *successor* function $x{+}1$ to the constant 0: $(\cdots((0{+}1){+}1)\cdots){+}1$. Monadic representation is used in the rules of *induction on monadic notation*, better known as *mathematical induction* shown on the left (ϕ is Σ_1):

$$\begin{array}{c} \phi[x]\, * \\ \phi[0]\, * \qquad \phi[x] \to \phi[x+1]\, * \end{array} \qquad\qquad \begin{array}{c} \phi[x, f(x,\vec{z}), \vec{z}]\, * \\ \begin{array}{cc} \phi[0, f(0,\vec{z}), \vec{z}]\, * & \phi[x, f(x,\vec{z}), \vec{z}] \\ f(0,\vec{z}) := \tau_1[\vec{z}] & \phi[x+1, f(x+1,\vec{z}), \vec{z}]\, * \\ & f(x+1,\vec{z}) := \tau_2[f(x,\vec{z}), x, \vec{z}] \end{array} \end{array}$$

The corresponding rule of *recursion on monadic notation* is applied on the right to extract a witness $f(x,\vec{z})$ for the specification $\exists y\, \phi[x, y, \vec{z}]$. Note that in the inductive case we effectively use $f(x,\vec{z})$ as the eigenvariable for the IH formula $\exists y\, \phi[x, y, \vec{z}]$. The 'eigenvariable' is then used in the definition of $f(x + 1, \vec{z})$. The extracted definition is by *primitive recursion*:

$$\begin{aligned} &f(0,\vec{z}) = \tau_1[\vec{z}] \\ &f(x+1,\vec{z}) = \tau_2[f(x,\vec{z}), x, \vec{z}] \quad . \end{aligned}$$

3.4 Square root function. The square root function $[\sqrt{x}]$ satisfies the specification: $\forall x \exists y\, y^2 \leq x < (y + 1)^2$. Its definition:

$$\begin{aligned} &[\sqrt{0}] = 0 \\ &[\sqrt{x+1}] = [\sqrt{x}] \leftarrow x + 1 < ([\sqrt{x}] + 1)^2 \\ &[\sqrt{x+1}] = [\sqrt{x}] + 1 \leftarrow x + 1 = ([\sqrt{x}] + 1)^2 \end{aligned}$$

is extracted from the following proof by induction on x of its witnessing formula. In the base case it suffices to define $[\sqrt{0}] := 0$ and the inductive case is:

$$[\sqrt{x}]^2 \le x < ([\sqrt{x}]+1)^2 \quad \text{(IH)}$$

$$[\sqrt{x+1}]^2 \le x+1 < ([\sqrt{x+1}]+1)^2 * \ (1)$$

$$x+1 < ([\sqrt{x}]+1)^2 \qquad\qquad x+1 = ([\sqrt{x}]+1)^2 \qquad\qquad x+1 > ([\sqrt{x}]+1)^2$$

$$[\sqrt{x+1}] := [\sqrt{x}] \qquad\qquad [\sqrt{x+1}] := [\sqrt{x}]+1$$

We use a rule of *trichotomy* discrimination and in the branch where $x{+}1 < ([\sqrt{x}]{+}1)^2$ holds we satisfy the goal (1) by defining $[\sqrt{x+1}] := [\sqrt{x}]$ because we have from IH: $[\sqrt{x}]^2 \le x < x{+}1 < ([\sqrt{x}]{+}1)^2$. When $x{+}1 = ([\sqrt{x}]{+}1)^2$ holds then the shown definition trivially satisfies (1). The last case $x{+}1 > ([\sqrt{x}]{+}1)^2$ leads to contradiction with IH, i.e. the case cannot happen, and the function value is immaterial. Note that the clause corresponding to the contradictory branch is omitted in the extracted program and yields 0 by *default*.

3.5 Assignment rules. The twofold occurrence of $[\sqrt{x}]$ in the recursive clauses in Par. 3.4 causes the application to be computed twice for each value of x. This leads to the exponential explosion of computation time. The explosion can be prevented by an *assignment*: $[\sqrt{x}] = z$ folding to a let term $L_z([\sqrt{x}], \ldots z \ldots)$:

$$[\sqrt{0}] = 0$$
$$[\sqrt{x+1}] = z \leftarrow [\sqrt{x}] = z \wedge x+1 < (z+1)^2$$
$$[\sqrt{x+1}] = z+1 \leftarrow [\sqrt{x}] = z \wedge x+1 = (z+1)^2 \quad .$$

In the following proof we force the assignment to be extracted by an *assignment* rule $[\sqrt{x}] =: z$ with the eigenvariable z:

$$[\sqrt{x}]^2 \le x < ([\sqrt{x}]+1)^2$$

$$[\sqrt{x+1}]^2 \le x+1 < ([\sqrt{x+1}]+1)^2 *$$

$$[\sqrt{x}] =: z$$

$$x+1 < (z+1)^2 \qquad\qquad x+1 = (z+1)^2 \qquad\qquad x+1 > (z+1)^2$$

$$[\sqrt{x+1}] := z \qquad\qquad [\sqrt{x+1}] := z+1$$

3.6 Measure induction. The rule of *measure* induction:

$$\phi[\vec{x}] *$$

$$\forall \vec{y}(m(\vec{y}) < m(\vec{x}) \rightarrow \phi[\vec{y}]) \rightarrow \phi[\vec{x}] *$$

permits to derive the goal $\phi[\vec{x}]$ under IH that $\phi[\vec{y}]$ holds for all $\vec{y}$ having lesser *measure*: $m(\vec{y}) < m(\vec{x})$. Note that complete induction on x is a special case of measure induction with $m(x) = x$.

We can extract a program for the integer division $x \div y$ by the following proof of the witnessing formula for $y > 0 \rightarrow \exists q \exists r(x = q{\cdot}y + r \wedge r < y)$ where we do not treat $y > 0$ as an assumption. The proof is by complete induction on x:

$$y > 0 \rightarrow \exists r(x = (x \div y){\cdot}y + r \wedge r < y) * \ (1)$$

$$\forall x_1(x_1 < x \rightarrow y > 0 \rightarrow \exists r(x_1 = (x_1 \div y){\cdot}y + r \wedge r < y) \ \text{(IH)}$$

$$y = 0 \qquad\qquad y \neq 0$$

$$\top * \ (2) \qquad\qquad \exists r(x = (x \div y){\cdot}y + r \wedge r < y) * \ (3)$$

$$x < y \qquad\qquad x \ge y$$

$$x \div y := 0 \qquad\qquad x = ((x \dot{-} y) \div y + 1){\cdot}y + r \wedge r < y \ \ (4)$$

$$x \div y := (x \dot{-} y) \div y + 1$$

We do first a *negation* discrimination with $y = 0$. In the case $y = 0$ the goal (1) simplifies to (2) and so any defining rule for $x \div 0$ will do. In the case $y \neq 0$ we do a dichotomy discrimination. In the case $x < y$ we satisfy the simplified goal (3) with the substitution $r := x$ and by defining $x \div y := 0$. In the case $x \geq y$ we have $x \dot{-} y < x$ and we instantiate IH with $x_1 := x \dot{-} y$ and use r as eigenvariable whereby we obtain (4) after a simplification. We now satisfy the goal (3) with the substitution $r := r$ and with a definition as shown. Instantiations and existential substitutions have no computational content provided we do not use in the definition rules eigenvariables (in this case r) other than those coming from assignments and patterns. The extracted definition is

$$
\begin{aligned}
&x \div y = 0 \leftarrow y \neq 0 \land x < y \\
&x \div y = (x \dot{-} y) \div y + 1 \leftarrow y \neq 0 \land x \geq y
\end{aligned}
$$

where the clause $x \div y = 0 \leftarrow y = 0$ is omitted by default. If we decided to apply the assumption rule to (1) without doing the discrimination on $y = 0$ then the definition extracted from the right branch would not contain the tests $y \neq 0$ and would still satisfy the partial correctness but not the termination formula $\exists q\, x \div 0 \asymp q$ for $y = 0$.

The extracted program is less optimal than it should be due to repeated tests $y \neq 0$. We obtain a better one when we do the extraction with the formula: $\exists r(x = (x \div_1 y) \cdot y + r \land r < y)$ under the assumption $y > 0$. The left branch now disappears and we get a definition of a partial function $x \div_1 y$ (diverging when $y = 0$) which can be explicitly completed to $\div$ as follows:

$$
\begin{aligned}
&x \div_1 y = 0 \leftarrow x < y \\
&x \div_1 y = (x \dot{-} y) \div_1 y + 1 \leftarrow x \geq y \\
&x \div y = x \div_1 y \leftarrow y > 0 \quad .
\end{aligned}
$$

3.7 A faster program for $[\sqrt{x}]$ with accumulators. We can save the squaring operation in the test $x + 1 < (z + 1)^2$ of $[\sqrt{x}]$ in Par. 3.5 which is repeatedly done for every recursive call by defining a function $f(z, a, x)$ with *accumulators* z and a such that $f(z, z^2, x) = [\sqrt{x}]$ provided $z^2 \leq x$. We intend to define f by *backward recursion* where a grows towards x, i.e. by recursion with the measure $m(z, a, x) = x \dot{-} a$. As z goes to $z{+}1$ the accumulator $a = z^2$ goes to $a_1 = a{+}2{\cdot}z{+}1 = (z{+}1)^2$. This arrangement reduces the squaring operation to the increments by $2{\cdot}z{+}1$ which are fast in the dyadic notation. The definition:

$$
\begin{aligned}
&f(z, a, x) = z \leftarrow a + 2{\cdot}z + 1 = a_1 \land x < a_1 \\
&f(z, a, x) = f(z + 1, a_1, x) \leftarrow a + 2{\cdot}z + 1 = a_1 \land x \geq a_1 \quad .
\end{aligned}
$$

is extracted from the following proof by induction with measure m of the witnessing formula for $z^2 = a \leq x \rightarrow \exists y\, y^2 \leq x < (y{+}1)^2$:

$$\forall z_1 a_1 x_1 (x_1 \dot{-} a_1 < x \dot{-} a \rightarrow z_1^2 = a_1 \leq x_1 \rightarrow f(z_1, a_1, x_1)^2 \leq x_1 < (f(z_1, a_1, x_1){+}1)^2) \quad \text{(IH)}$$

$$z^2 = a \leq x \quad (1)$$

$$f(z, a, x)^2 \leq x < (f(z, a, x) + 1)^2 * \quad (2)$$

$$a + 2{\cdot}z + 1 =: a_1$$

$$
\begin{array}{ll}
x < a_1 & x \geq a_1 \\
f(z, a, x) := z & f(z + 1, a_1, x)^2 \leq x < (f(z + 1, a_1, x) + 1)^2 \quad (3) \\
 & f(z, a, x) := f(z + 1, a_1, x)
\end{array}
$$

The proof assigns to a_1 so we have $a_1 = (z+1)^2$ and then does a dichotomy discrimination. When $x < a_1$ the goal (2) holds trivially after defining $f(z,a,x) := z$. When $x \geq a_1$ then $x \dot{-} a_1 < x \dot{-} a$ and we instantiate IH to (3) from which the goal (2) is obtained by the shown definition. Note that the conclusion (1) of an assumption rule is without computational effect because when $z^2 \neq a$ or $a > x$ holds then f still terminates.

The square root function can be now explicitly derived by $[\sqrt{x}] = f(0,0,x)$. This is another example when a faster program can be obtained by a detour through a partial function although f happens to be total in this case.

3.8 An optimal program for $[\sqrt{x}]$ by 4-adic recursion. All recursive definitions extracted so far share the same shortcoming: the recursion goes exponentially longer than it should. Recursion on monadic representation is computationally feasible only when initialized with log-sized arguments. Since we cannot restrict the square root function to small arguments, we have to use more economical recursion scheme. Dyadic recursion will not work but recursion in the base 4 will because we have $[\sqrt{4{\cdot}x}] \approx 2{\cdot}[\sqrt{x}]$. For a base $p > 1$ and an offset $m > 0$ we have the following rules of induction on *p-adic notation*:

$$\phi[x]\,*$$

$$\phi[0]\,* \quad \phi[1]\,* \quad \cdots \quad \phi[m-1]\,* \qquad m \leq i \land i < m+p \land \phi[x] \to \phi[p{\cdot}x+i]\,*$$

We can extract the following fast definition of the square root function:

$$[\sqrt{0}] = 0$$
$$[\sqrt{1}] = 1$$
$$[\sqrt{2}] = 1$$
$$[\sqrt{4{\cdot}x+i}] = z \leftarrow 3 \leq i \land i < 7 \land 2{\cdot}[\sqrt{x}] = z \land 4{\cdot}x+i < (z+1)^2$$
$$[\sqrt{4{\cdot}x+i}] = z+1 \leftarrow 3 \leq i \land i < 7 \land 2{\cdot}[\sqrt{x}] = z \land 4{\cdot}x+i \geq (z+1)^2 \land$$
$$\quad 4{\cdot}x+i < (z+2)^2$$
$$[\sqrt{4{\cdot}x+i}] = z+2 \leftarrow 3 \leq i \land i < 7 \land 2{\cdot}[\sqrt{x}] = z \land 4{\cdot}x+i \geq (z+1)^2 \land$$
$$\quad 4{\cdot}x+i \geq (z+2)^2 \quad .$$

from a proof of its witnessing formula by p-adic induction with $p = 4$ and $m = 3$. We leave the base cases to the reader and show here only the inductive case:

$$3 \leq i < 7$$
$$[\sqrt{x}]^2 \leq x < ([\sqrt{x}]+1)^2 \quad \text{(IH)}$$
$$[\sqrt{4{\cdot}x+i}]^2 \leq 4{\cdot}x+i < ([\sqrt{4{\cdot}x+i}]+1)^2\,* \quad (1)$$
$$2{\cdot}[\sqrt{x}] =: z$$

$$4{\cdot}x+i < (z+1)^2 \qquad\qquad 4{\cdot}x+i \geq (z+1)^2$$
$$[\sqrt{4{\cdot}x+i}] := z$$
$$4{\cdot}x+i < (z+2)^2 \qquad\qquad 4{\cdot}x+i \geq (z+2)^2$$
$$[\sqrt{4{\cdot}x+i}] := z+1 \qquad\qquad [\sqrt{4{\cdot}x+i}] := z+2$$

After assignment $2{\cdot}[\sqrt{x}] =: z$ we do a dichotomy discrimination. When $4{\cdot}x{+}i < (z{+}1)^2$ we define $[\sqrt{4{\cdot}x{+}i}] := z$ and prove the goal (1) by:

$$[\sqrt{4{\cdot}x+i}]^2 = z^2 = 4{\cdot}[\sqrt{x}]^2 \overset{IH}{\leq} 4{\cdot}x < 4{\cdot}x+i < (z+1)^2 = ([\sqrt{4{\cdot}x+i}]+1)^2 \ .$$

When $4{\cdot}x{+}i \geq (z{+}1)^2$ then we do another dichotomy discrimination. When $4{\cdot}x{+}i < (z{+}2)^2$ we define $[\sqrt{4{\cdot}x{+}i}] := z{+}1$ and prove the goal trivially. Finally, when $4{\cdot}x{+}i \geq (z{+}2)^2$ then we define $[\sqrt{4{\cdot}x{+}i}] := z{+}2$ and prove the the first half of the goal trivially. For the second half we have $x < ([\sqrt{x}]{+}1)^2$ from IH and thus $x \leq [\sqrt{x}]^2{+}2{\cdot}[\sqrt{x}]$. Therefore we get:

$$4{\cdot}x + i \leq 4{\cdot}[\sqrt{x}]^2 + 8{\cdot}[\sqrt{x}] + i < 4{\cdot}[\sqrt{x}]^2 + 12{\cdot}[\sqrt{x}] + 9 =$$
$$(2{\cdot}[\sqrt{x}] + 3)^2 = (z+3)^2 = ([\sqrt{4{\cdot}x+i}] + 1)^2 \ .$$

3.9 Induction on pair notation. The rules of induction on *pair notation* permit to extraction programs with computationally feasible recursion:

$$\phi[x] *$$
$$\phi[0] * \qquad \phi[v] \wedge \phi[w] \to \phi[v,w] *$$

For instance, the function $x \oplus y$ concatenating two lists is defined primitively recursively by *recursion on pair notation*: $0 \oplus y = y$ and $(v,w) \oplus y = v, w \oplus w$. We can easily prove by pair induction properties of concatenation, such as the associativity: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.

Consider the well-known function $\mathit{Flat}(x)$ *flattening* the number x into a list of the pair size $|x|$ and containing only zeroes as elements. The function is defined primitively recursively by pair recursion:

$$\mathit{Flat}(0) = 0$$
$$\mathit{Flat}(v,w) = 0, \mathit{Flat}(v) \oplus \mathit{Flat}(w) \quad .$$

The program runs in time $O(|x|^2)$ due to repeated concatenations whereas $0(|x|)$ suffices when we extract the accumulator version $f(x,a)$ of *Flat* from the following proof by pair recursion on x with the witnessing formula for the Π_2-*specification formula* $\forall a \exists y\, y = \mathit{Flat}(x) \oplus a$:

$$\forall a\, f(x,a) = \mathit{Flat}(x) \oplus a *$$
$$f(0,a) = \mathit{Flat}(0) \oplus a * (1) \qquad \forall a\, f(v,a) = \mathit{Flat}(v) \oplus a \quad (IH_1)$$
$$f(0,a) := a \qquad \forall a\, f(w,a) = \mathit{Flat}(w) \oplus a \quad (IH_2)$$
$$f((v,w),a) = \mathit{Flat}(v,w) \oplus a * (2)$$
$$f((v,w),a) := 0, f(v, f(w,a))$$

Of course, we can extract f explicitly by defining $f(x,a) := \mathit{Flat}(x) \oplus a$ but that runs in time $O(|x|^2)$. The goal (1) in the base case is trivially satisfied by defining $f(0,a) := a$. In the inductive case we satisfy the goal (2) by the shown definition and by instantiating both induction hypotheses:

$$f((v,w),a) \stackrel{\text{def}}{=} 0, f(v, f(w,a)) \stackrel{IH_2}{=} 0, f(v, \mathit{Flat}(w) \oplus a) \stackrel{IH_1}{=}$$
$$0, \mathit{Flat}(v) \oplus \mathit{Flat}(w) \oplus a = \mathit{Flat}(v,w) \oplus a$$

The reader will note that we have instantiated IH_1 with $a := \mathit{Flat}(w) \oplus a$ and so the induction formula must be Π_2. The extracted program for f is by *simply nested* recursion on pair notation and *Flat* is explicitly defined from f:

$$f(0,a) = a$$
$$f((v,w),a) = 0, f(v, f(w,a))$$
$$\mathit{Flat}(x) = f(x,0) \quad .$$

3.10 Binary trees. We arithmetize binary trees with labels from N as in [KV99] by two constructors: $E = 0, 0$ (the empty tree) and $Nd(x, l, r) = 1, x, l, r$ (a node with label x and two subtrees l and r). The predicate Bt holding of codes of binary trees is defined primitively recursively by *course of values recursion*:

$$
\begin{array}{l}
Bt(E) \\
Bt\, Nd(x, l, r) \leftarrow Bt(l) \wedge Bt(r) \quad .
\end{array}
$$

This is a definition of an *inductive* predicate which affords rules of *Bt-induction*:

$$
\begin{array}{cc}
\multicolumn{2}{c}{Bt(t) \rightarrow \phi[t] *} \\
\phi[E] * & \phi[l] \wedge \phi[r] \rightarrow \phi[Nd(x, l, r)] *
\end{array}
$$

The function $|t|_b$ counting the number of labels and the predicate $x \in_b t$ of binary tree membership are defined primitively recursively by *Bt-recursion*:

$$
\begin{array}{ll}
|E|_b = 0 & x \in_b Nd(y, l, r) \leftarrow x = y \\
|Nd(x, l, r)|_b = |l|_b + |r|_b + 1 & x \in_b Nd(y, l, r) \leftarrow x \neq y \wedge x \in_b l \\
 & x \in_b Nd(y, l, r) \leftarrow x \neq y \wedge x \notin_b l \wedge x \in_b r \quad .
\end{array}
$$

The close relationship between the programming constructs and our tableau proof rules is nicely demonstrated by the proof of the 'boundedness' property of binary trees: $x \in_b t \rightarrow x < t$ which is by complete induction on t and uses the entire clausal definition of $x \in_b t$ for discrimination because under that assumption exactly one of the clauses applies:

$$
\begin{array}{ccc}
\multicolumn{3}{c}{\forall t_1 (t_1 < t \rightarrow x \in_b t_1 \rightarrow x < t_1) \quad \text{(IH)}} \\
\multicolumn{3}{c}{x \in_b t} \\
\multicolumn{3}{c}{x < t *} \\
t = Nd(y, l, r) & t = Nd(y, l, r) & t = Nd(y, l, r) \\
x = y & x \neq y \wedge x \in_b l & x \neq y \wedge x \notin_b l \wedge x \in_b r \\
 & [t_1 := l \text{ in } (IH)] & [t_1 := r \text{ in } (IH)]
\end{array}
$$

The following explicitly defined predicates are primitive recursive because the quantifiers are *bounded*:

$$
t \prec x \leftrightarrow \forall y \in_b t \,.\, y < x \qquad\qquad t \succ x \leftrightarrow \forall y \in_b t \,.\, y > x \;.
$$

3.11 Binary search trees. We arithmetize binary search trees by codes t satisfying the predicate $Bst(t)$ primitively recursively defined by Bt-recursion:

$$
\begin{array}{l}
Bst(E) \\
Bst\, Nd(x, l, r) \leftarrow l \prec x \wedge r \succ x \wedge Bst(l) \wedge Bst(r) \quad .
\end{array}
$$

Bst is an inductive predicate affording the following rules of *Bst-induction*:

$$
\begin{array}{cc}
\multicolumn{2}{c}{Bst(t) \rightarrow \phi[t] *} \\
\phi[E] * & l \prec x \wedge r \succ x \wedge \phi[l] \wedge \phi[r] \rightarrow \phi[Nd(x, l, r)] *
\end{array}
$$

By straightforward Bst-induction we can prove $Bst(t) \rightarrow Bt(t)$ asserting that binary search trees form a *subsort* of binary trees.

We can extract from $I\Sigma_1$-proofs also definitions of predicates because they are introduced through their characteristic functions (0 is false, 1 is true). For instance, the program for the predicate $x \in_s t$ extracted as an 'unknown' predicate from the following proof can speed up the $O(|t|_b)$ search of $x \in_b t$ up to $O(\log |t|_b)$:

$$Bst(t) \to (x \in_s t \leftrightarrow x \in_b t) *$$

$$\begin{array}{ll}
\begin{array}{l} x \in_s E \leftrightarrow x \in_b E * \\ \quad x \in_s E := \bot \end{array} &
\begin{array}{l} \quad\; l \prec y \wedge r \succ y \\ \quad x \in_s l \leftrightarrow x \in_b l \;\; (1) \\ \quad x \in_s r \leftrightarrow x \in_b r \\ x \in_s Nd(y,l,r) \leftrightarrow x \in_b Nd(y,l,r) \;\; * \;\; (2) \end{array}
\end{array}$$

$$\begin{array}{ccc}
x < y & x \doteq y & x > y \\
x \in_s Nd(y,l,r) := x \in_s l & x \in_s Nd(y,l,r) := \top & x \in_s Nd(y,l,r) := x \in_s r
\end{array}$$

The proof is by Bst-induction followed by a trichotomy discrimination. Note the definitions of the unknown truth values of $x \in_s t$. For instance, the goal (2) is proved in the case $x < y$ by:

$$x \in_s Nd(y,l,r) \overset{\text{def}}{\Leftrightarrow} x \in_s l \overset{\text{IH:(1)}}{\Leftrightarrow} x \in_b l \overset{r \succ y}{\Leftrightarrow} x \in_b Nd(y,l,r) \; .$$

The extracted definition with the negative clause omitted by default is:

$$\begin{array}{l}
x \in_s Nd(y,l,r) \leftarrow x < y \wedge x \in_s l \\
x \in_s Nd(y,l,r) \leftarrow x = y \\
x \in_s Nd(y,l,r) \leftarrow x > y \wedge x \in_s r \quad .
\end{array}$$

The function $\{x\} \cup t$ inserting x into the binary search tree t satisfies:

$$Bst(t) \to Bst(\{x\} \cup t) \wedge \forall z(z \in_s \{x\} \cup t \leftrightarrow z = x \vee z \in_s t) \; .$$

The quantifier $\forall z$ is bounded and so we can extract from a proof by Bst-induction on t the following program:

$$\begin{array}{l}
\{x\} \cup E = Nd(x,E,E) \\
\{x\} \cup Nd(y,l,r) = Nd(y,\{x\} \cup l, r) \leftarrow x < y \\
\{x\} \cup Nd(y,l,r) = Nd(y,l,r) \leftarrow x = y \\
\{x\} \cup Nd(y,l,r) = Nd(y,l,\{x\} \cup r) \leftarrow x > y \quad .
\end{array}$$

4 Admissibility of Extraction Rules and of Induction Rules in $I\Sigma_1$-arithmetic

Our tableau proof system uses extraction rules and a rich set of induction rules and we now briefly sketch their admissibility in $I\Sigma_1$-arithmetic.

4.1 Admissibility of extraction rules. Extraction proofs 2.5(1) are translated to the standard $I\Sigma_1$-proofs 2.5(2) as outlined in Par. 2.5. Applications of potentially partial function symbols f in the terms of extraction proofs are only a shorthand to improve the readability and are eliminable by the graph of f. Induction hypotheses $\phi[\vec{\rho}, f(\vec{\rho})]$ are thus of the form $\exists y(\phi[\vec{\rho}, y] \wedge f(\vec{\rho}) \asymp y)$ asserting that $f(\vec{\rho})$ is defined.

4.2 Admissibility of basic induction rules. Our proof system is tableau-based and so it has induction rules with side formulas rather than the induction axioms of $I\Sigma_1$-arithmetic. It should be obvious that the induction formula $\phi[x]$ of the rule of mathematical induction in Par. 3.3 can be any Σ_1-formula with arbitrary side formulas. Π_1-induction rules with arbitrary side formulas reduce to Σ_1-rules the same way as axioms do (see [HP93]). Weak Π_2-induction rules as defined in [KV99] are admissible; their side formulas must be weak, i.e. Π_1 in assumptions and Σ_1 in goals. The same restrictions apply to complete, dyadic, and pair inductions because the first reduces to mathematical induction and the last two reduce directly to complete induction. Π_2-induction rules are necessary when the extracted programs have nested recursion or substitution in parameters (see Par. 3.9).

4.3 Admissibility of rules of measure induction. A rule of measure induction (see Par. 3.6) with the induction formula $\phi[\vec{x}]$ reduces to the rule of mathematical induction on n with the formula: $\forall\vec{x}(m(\vec{x}) < n \to \phi[\vec{x}])$. Hence, if ϕ is Σ_1 or Π_2 the side formulas must be weak.

4.4 Admissibility of R-induction rules. After the elimination of variables introduced into the clauses by assignments and by discrimination on patterns every clausal definition of a primitive recursive predicate R can be brought into the form

$$R(\vec{x}) \leftrightarrow \bigvee_{i=1}^{k} \psi_i[R(\cdot)\,;\vec{x}] \tag{1}$$

with ψ_i conjunctions of literals. For instance, the pattern $\tau = x + 1$ is replaced by $\tau > 0$ and the variable x is eliminated by $\tau \dot{-} 1$.

The predicate R is *inductive* if all recursive applications in $\bigvee_{i=1}^{k} \psi_i$ are positive. The inductive predicate R affords the rules of R-*induction*:

$$\begin{array}{c} R(\vec{x}) \to \phi[\vec{x}]\, * \\ \vdots \\ \bigvee_{i=1}^{k} \psi_i[\phi[\cdot];\vec{x}] \to \phi[\vec{x}]\, * \\ p \end{array} \qquad \begin{array}{l} \phi \text{ is } \Sigma_1,\ \Pi_1, \text{ or } \Pi_2, \text{ where if } \phi \text{ is } \Pi_2 \text{ then} \\ \text{the side formulas must be weak.} \end{array} \tag{2}$$

Boyer and Moore [BM79] were the first to use inductive rules derived from recursive definitions. Similar rules are used in HOL [GM93].

The soundness of the rule (2) follows from the fact that R is the minimal predicate satisfying the ($\leftarrow$)-implication of (1). This is sufficient for the admissibility of (2) in the system [BM79]. The admissibility is proved in HOL in a rather strong Simple Theory of Types (whose strength is similar to ZF). Our task is substantially harder because we have to prove the admissibility in a very weak theory $I\Sigma_1$. For that we use an auxiliary primitive recursive predicate $\bar{R}(i,\vec{x})$ approximating R and satisfying:

$$\neg\bar{R}(0,\vec{x}) \quad \text{and} \quad \bar{R}(i+1,\vec{x}) \leftrightarrow \bigvee_{i=1}^{k} \psi_i[\bar{R}(i,\cdot);\vec{x}]\ .$$

$I\Sigma_1$ clearly proves $R(\vec{x}) \leftrightarrow \exists i\, \bar{R}(i,\vec{x})$ and so the admissibility of (2) reduces to the problem of proving in $I\Sigma_1$

$$\forall\vec{x}(\bar{R}(i,\vec{x}) \rightarrow \phi[\vec{x}]) \tag{3}$$

from the proof p of $\bigvee_{i=1}^{k} \psi_i[\phi[\cdot];\vec{x}] \rightarrow \phi[\vec{x}]$. This is done by bounding in $I\Sigma_1$ the quantifiers $\forall\vec{x}$ of (3) with the help of a primitive recursive *course of values* function $f(i,\vec{z})$ which yields a list containing the pair $(a_1,\ldots,a_n)$ for all computational 'predecessors' $\vec{a}$ of $\vec{z}$ at the i-th level of recursion. The function satisfies:

$$(\vec{x})\ \varepsilon\ f(0,\vec{z}) \leftrightarrow (\vec{x}) = (\vec{z}) \quad \text{and} \quad (\vec{x})\ \varepsilon\ f(i,\vec{z}) \rightarrow (\vec{\rho}[\vec{x}])\ \varepsilon\ f(i+1,\vec{z})$$

for all recursive applications $R(\vec{\rho}[\vec{x}])$ in $\bigvee_{i=1}^{k} \psi_i$. Here $x\ \varepsilon\ y$ is the list membership predicate. $I\Sigma_1$ then proves by induction on i:

$$\forall j \leq m\, \forall x_1{\leq}f(j,\vec{z}) \ldots \forall x_n{\leq}f(j,\vec{z})\, \big((\vec{x})\ \varepsilon\ f(j,\vec{z}) \wedge i+j=m \wedge \bar{R}(i,\vec{x}) \rightarrow \phi[\vec{x}]\,\big)\ .$$

Now (3) follows in $I\Sigma_1$ by setting $m := i$, $\vec{z} := \vec{x}$, and $j := 0$. Note that we have actually proved that R-induction is admissible (with weak side formulas if ϕ is Σ_1) even if R is a recursively enumerable predicate although such an R cannot be used in clausal definitions.

5 Conclusion

We have hopefully demonstrated that the seemingly weak $PR + I\Sigma_1$ paradigm is sufficient to introduce a usable programming language and its verification theory. We have implemented the paradigm in the system CL which we use in four undergraduate courses at our university. The courses are: Introduction to declarative programming, Predicate Logic, Program specification and verification, and Theory of computability. They are taken every year by about 250 students. The main reason why we can teach this is that the intuition about the true properties of CL programs is easily acquired by the students. They do not need to know more than the standard model of $\mathbb{N}$.

We hope that we have convinced the reader that we can arithmetize in $\mathbb{N}$ the basic data structures needed in the computer programming with the level of comfort comparable to that in the declarative programming languages. The objection that our system proves the termination of only primitive recursive functions is answered by pointing out that the computationally feasible functions are a very small subset of PR functions. Besides, by the Incompleteness theorem of Gödel, no formal theory can prove the termination of all recursive functions.

We have also built into CL a mechanism for the definition of abstract data types (ADT) through non-recursive extensions of $I\Sigma_1$-arithmetic (this defines ADT's). The consistency of extensions is proved by primitive recursive interpretations into $I\Sigma_1$ (this constitutes implementations of ADT's).

In the close future we plan to implement the automatic extraction of witnessing functions from Π_2-specifications as discussed in this paper. This can be done without the loss of efficiency when compared with hand-coded programs but with the added advantage that we prove at the same time the total correctness of the function being constructed. We justify our approach of leaving to the user the decision about the computational content of rules by the current trend in the design of theorem provers. Namely, the first theorem provers, for instance the system of Boyer and Moore, tried to construct the proofs automatically. This has proved to be untenable and so the newer systems such as NuPrl, PX, PVS, Coq, Isabelle, MINLOG, HOL, and CL are intelligent proof checkers where the user guides the proofs.

We further plan to add to CL the intensional functionals (via coding into $\mathbb{N}$ in the style of lambdas of LISP). We will then have demonstrated that CL can do most of the things declarative languages can do but with the simple semantics of a rather weak formal theory. We are currently preparing an extension of CL to deal with typed programs and we plan to compile the CL programs with the in-place-modification of data structures.

References

[BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the MINLOG system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction*, volume II. Kluwer, 1998.

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.

[GM93] M. J. C. Gordon and T. F. Melhalm, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[HN88] S. Hayashi and H. Nakano. *PX: A Computational Logic.* The MIT Press, 1988.

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

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

[KV99] J. Komara and P. J. Voda. Theorems of Péter and Parsons in computer programming. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Proceedings of CSL'98*, number 1584 in LNCS, pages 204–223. Springer Verlag, 1999.

[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.

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

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

[Vod95] 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, 1995.

[Vod00] P. J. Voda. Theory of Recursive Functions & Computability (from Computer Programmer's View), 2000. Available through WWW from http://dent.ii.fmph.uniba.sk/~voda/text.ps.gz.