Schwichtenberg-style lambda definability is undecidable.

Jan Maolepszy, Magorzata Moczurad, Marek Zaionc

Computer Science Department, Jagiellonian University,

Nawojki 11, 30-072 Krakow, Poland

E-mail {madry, maloleps, zaionc}@ii.uj.edu.pl

Manuscript version, June 1996

Abstract: We consider lambda definability problem over an arbitrary free algebra. There is a natural notion of primitive recursive function in such algebras and at the same time a natural notion of a λ-definable function. The paper answers the following question: for a given free algebra and a primitive recursive function within this algebra decide whether this function is lambda definable. The paper shows that the question is undecidable if the algebra is infinite. The main part of the paper is dedicated to the algebra of numbers in which lambda definability is described by the Schwichtenberg theorem. The result for an arbitrary infinite free algebra has been obtained by a simple interpretation of numerical functions as recursive functions in this algebra. This result is a counterpart of the distinguished result of Loader [7] in which lambda terms are evaluated in finite domains for which undecidability of λ-definability is proved.

1 Simple typed λ-calculus

We shall consider a simple typed lambda calculus with a single ground type O. The set $TYPES$ is defined as follows: O is a type and if τ and μ are types then $\tau \to \mu$ is a type. By type $\tau^k \to \mu$ we mean the type $\tau \to (... \to (\tau \to \mu)...)$ with exactly k occurrences of τ. For $k = 0$, $\tau^0 \to \mu$ is μ.

For any type τ there is given a denumerable set of variables $V(\tau)$. Any type τ variable is a type τ term. If T is a term of type $\tau \to \mu$ and S is a type τ term, then TS is a term which has type μ. If T is a type μ term and x is a type τ variable, then $\lambda x.T$ is a term of type $\tau \to \mu$. The axioms of equality between terms have the form of $\beta\eta$ conversions and the convertible terms will be written as $T =_{\beta\eta} S$. A closed term is a term without free variables. For more detailed treatment of typed lambda calculus see [12].

2 Free Algebras

Algebra A given by a signature $S_A = (\alpha_1, ..., \alpha_n)$ has n constructors $c_1, ...c_n$ of arities $\alpha_1, ..., \alpha_n$, respectively. Expressions of the algebra A are defined recursively as a minimal set such that if α_i=0 then c_i is an expression and if $t_1, ..., t_{\alpha_i}$ are expressions then $c_i(t_1, ..., t_{\alpha_i})$ is an expression. We may assume that at least one α_i is equal to 0, otherwise the set of expressions is empty. Let $\mathbf{A}$ be the set of all expressions in algebra A. We are going to investigate mappings $f : \mathbf{A}^k \to \mathbf{A}$. Any constructor can be seen as a function $c_i : \mathbf{A}^{\alpha_i} \to \mathbf{A}$. Let $\vec{x}$ be a list of expressions $x_1, ..., x_k$ and $f_1, ..., f_n$ be functions of arity $k + \alpha_i$, respectively. A class $\mathcal{X}$ of mappings is closed under primitive recursion if the $(k + 1)$-ary function h defined by

$$h(c_i(y_1, ..., y_{\alpha_i}), \vec{x}) = f_i(h(y_1, \vec{x}), ..., h(y_{\alpha_i}, \vec{x}), \vec{x}) \text{ for all } i \leq n$$

belongs to $\mathcal{X}$ whenever functions $f_1, ..., f_n$ are in $\mathcal{X}$. In the case when α_i=0 the equation is reduced to $h(c_i, \vec{x}) = f_i(\vec{x})$. Let us distinguish the following operations: p_i^k is the k-ary projection which extracts i-th argument, i.e. $p_i^k(x_1, ..., x_k) = x_i$ and C_i^k is k-ary constant function when $\alpha_i = 0$ which maps constantly into expression c_i, i.e $C_i^k(x_1, ..., x_k) = c_i$. The class of primitive recursive functions over algebra A is defined as a minimal class containing constructors, projections, constant functions, and closed under composition and primitive recursion. The subclass $F_0^{\mathbf{A}}$ of primitive recursive functions is a minimal class containing projections, constructors, and closed under composition. The subclass $F_k^{\mathbf{A}}$ for $k \geq 0$ is a class of all $(k+p)$-ary functions f on $\mathbf{A}$ such that for all expressions $E_1, ..., E_p \in \mathbf{A}$ the function p defined as $p(x_1, ..., x_k) = f(x_1, ..., x_k, E_1, ..., E_p)$ belongs to $F_0^{\mathbf{A}}$. This includes a case when $k = 0$, which means that $F_0^{\mathbf{A}}$ is a set of constant 0-ary constructors. The class $F_\lambda^{\mathbf{A}}$ is a minimal class containing constructors, projections, constant functions, and closed under composition and the following limited version of primitive recursion: if $f_i \in F_\lambda^{\mathbf{A}} \cap F_{\alpha_i}^{\mathbf{A}}$ for $i \leq n$ then the function h obtained from f_i by primitive recursion belongs to F_λ.

We are going to give more attention to the algebra of natural numbers N based on the signature $(0, 1)$. Traditionally two constructors of the algebra N are called 0 (zero) and s (successor). The class of primitive recursive functions over $\mathbf{N}$ is the least one containing the constant zero function, successor, and closed under composition and the following form of primitive recursion:

$$\begin{aligned} h(0, \vec{x}) &= f_1(\vec{x}) \\ h(s(y), \vec{x}) &= f_2(h(y, \vec{x})), \vec{x}) \end{aligned}$$

In addition, we will discuss finite algebras A_n based on the signature $(0, ..., 0)$. Algebra A_n consists of a finite number of expressions $c_1, ...c_n$. It is easy to observe that every function on A_n is primitive recursive.

3 Representability

If A is an algebra given by a signature $S_A = (\alpha_1, ..., \alpha_n)$ then by $\tau^{\mathbf{A}}$ we mean a type $(O^{\alpha_1} \to O), \ldots, (O^{\alpha_n} \to O) \to O$. Assuming that at least one α_i is 0 we get that type $\tau^{\mathbf{A}}$ is not empty. There is a natural 1-1 isomorphism between expressions of the algebra A and closed terms of type $\tau^{\mathbf{A}}$. If c_i is a 0-ary constructor in A then the term $\lambda x_1 ... x_n . x_i$ represents c_i. If $\alpha_i > 0$ and $t_1, ..., t_{\alpha_i}$ are expressions in A represented by closed terms $T_1, ..., T_{\alpha_i}$ of type $\tau^{\mathbf{A}}$,then expression $c_i(t_1, ..., t_{\alpha_i})$ is represented by the term $\lambda x_1 ... x_n . x_i(T_1 x_1 ... x_n) .... (T_{\alpha_i} x_1 ... x_n)$. Thus, we have a 1-1 correspondence between closed terms of the type $\tau^{\mathbf{A}}$ and A. In fact any rank 2 type τ represents some free algebra A. If additionally τ is non-empty (there are some closed terms of type τ) then the algebra A is non-empty. The unique (up to $\beta\eta$-conversion) term which represents an expression t is denoted by $\underline{t}$.

Definition 3.1 *A function $h : \mathbf{A}^n \to \mathbf{A}$ is λ-definable by a closed term H of type $(\tau^{\mathbf{A}})^n \to \tau^{\mathbf{A}}$ iff for all expressions $t_1, ..., t_n$*

$$H \underline{t_1} \ldots \underline{t_n} =_{\beta\eta} \underline{h(t_1, ..., t_n)}.$$

It is easy to observe that any closed term H of the type $(\tau^{\mathbf{A}})^n \to \tau^{\mathbf{A}}$ uniquely defines the function $h : \mathbf{A}^n \to \mathbf{A}$ as follows: if $t_1, ..., t_n$ are expressions then the value is an expression represented by the term $H \underline{t_1}, \ldots, \underline{t_n}$. On the other hand, one function can be represented by many unconvertible terms.

For the algebra of natural numbers $N = (1, 0)$ the type $\tau^{\mathbf{N}} = (O \to O) \to (O \to O)$ is called the Church numerals type. In this case the natural isomorphism identifying closed terms of the type

$\tau^{\mathbf{N}}$ with expressions of the algebra N (non-negative integers) is such that for a number $n \in \mathbf{N}$ the $\underline{n}$ is the closed term $\lambda ux.u(...(ux)...)$ with exactly n occurrences of variable u.

Definition 3.2 *Function $f : \mathbf{N}^k \to \mathbf{N}$ is λ-definable by a closed term F of type $\underline{\mathbf{N}}^k \to \underline{\mathbf{N}}$ iff for all numbers $n_1, ..., n_k$*

$$F\underline{n_1}...\underline{n_k} =_{\beta\eta} \underline{f(n_1, ..., n_k)}$$

The following characteristics of lambda definable functions has been proved.

Theorem 3.3 *(Schwichtenberg [8] and Statman [9]) λ-definable functions on $\mathbf{N}$ are just compositions of 0, 1, addition, multiplication, sq and $\overline{sq}$ where*

$$sq(x) = \begin{cases} 0, & if\ x = 0 \\ 1, & if\ x \neq 0 \end{cases} \qquad \overline{sq}(x) = \begin{cases} 1, & if\ x = 0 \\ 0, & if\ x \neq 0 \end{cases}$$

Theorem 3.4 *(Zaionc [14] and Leivant [6]) λ-definable functions on algebra $\mathbf{A}$ are just functions from the class $F_\lambda^{\mathbf{A}}$*

Theorem 3.5 *(Zaionc [13]) Every function on a finite algebra is λ-definable.*

A much stronger version of this, covering also all functionals of finite type, is proved in [13].

4 Reconstruction of the polynomials

The aim of this section is to show the possibility of reconstructing extended polynomials from the finite set of appropriate examples. This problem has been stated by Bharat Jayraman for the purpose of extracting programs written in the form of lambda terms from the finite number of cases of the behavior of those programs. (for example see [2], [3]). In this technique for program extraction Huet unification procedure is used. See also the programming by examples paradigm in [5], [4]. In the case of reconstructing polynomials some partial solution was obtained by Kannan Govindarajan in [1].

In this section we will demonstrate two theorems. The first one is concern with standard polynomials (composition of addition and multiplication) of k variables. It is proved that only two examples are needed to determine the polynomial. For more general case concerning extended polynomials (composition of addition, multiplication, sq and $\overline{sq}$). it is proved that $2^{k+1} - 1$ examples are necessary and sufficient for extraction.

Definition 4.1 *By the set of polynomials we mean the least set of functions from N^k to N containing all projections, constant functions, addition and multiplication closed for composition.*

By the set of extended polynomials we mean the least set of functions from N^k to N containing all projections, constant functions, addition, multiplication and additionally sq and $\overline{sq}$ (see theorem 3.3 for definition) closed for composition.

Definition 4.2 *An extended polynomial* $f : \mathbf{N}^k \to \mathbf{N}$ *is determined by the finite set of* p *examples* $\{(\vec{x_1}, y_1), ..., (\vec{x_p}, y_p)\}$, *where* $\vec{x_i} \in \mathbf{N}^k$ *and* $y_i \in \mathbf{N}$ *for all* $i \leq p$, *if*

- $f(\vec{x_i}) = y_i$ *for all* $i \leq p$
- *if for some extended polynomial* $g : \mathbf{N}^k \to \mathbf{N}$ $g(\vec{x_i}) = y_i$ *for all* $i \leq p$ *then* $f = g$.

Definition 4.3 *A finite set* $\mathcal{X} \subset \mathbf{N}^k \times \mathbf{N}$ *is called a determinant if some extended polynomial* $f : \mathbf{N}^k \to \mathbf{N}$ *is determined by* $\mathcal{X}$.

Definition 4.4 *We are going to use the standard normal form notation for polynomials with positive integer coefficients. Every polynomial* $f : N^k \to N$ *has a unique representation in the following normal form:*

1. $k = 1$

$$f(x) = \sum_{i=0}^{m} \alpha_i x^i, \text{ where } \alpha_i \in N \text{ — polynomial coefficients}$$

2. $k > 1$

$$f(\vec{x}, x_k) = \sum_{i=0}^{m} \alpha_i(\vec{x}) x_k^i,$$

where $\alpha_i \in (N^{k-1} \to N)$ *— polynomials in normal form.*

Lemma 4.5 *Given polynomial* $f(x_1, \ldots, x_k) = \sum_{i=0}^{m} \alpha_i(x_1, \ldots, x_{k-1}) x_k^i$. *If a point* $(a_1, \ldots, a_k) \in \mathbf{N}^k$ *satisfies the following condition:*

$$\forall 0 \leq i \leq m \quad a_k > \alpha_i(a_1, \ldots, a_{k-1})$$

then there is an algorithm to find out the values $\alpha_i(a_1, \ldots, a_{k-1})$, $i = 1, \ldots, m$ *from the value of the polynomial at the point* $(a_1, \ldots, a_k)$.

Proof. Let $W = f(a_1, \ldots, a_k)$. Because $\forall 0 \leq i \leq m \ a_k > \alpha_i(a_1, \ldots, a_{k-1})$, the values $\alpha_i(a_1, \ldots, a_{k-1})$ can be computed as the digits of W in the number system with base a_k. W is divided by a_k until 0 is reached, and the computed reminders are the values $\alpha_0(a_1, \ldots, a_{k-1})$, $\alpha_1(a_1, \ldots, a_{k-1})$,.... □

Definition 4.6 *For the given total computable function* $f : \mathbf{N}^k \to \mathbf{N}$ *we define numbers* $D_f, E_f \in N$ *and linear function* $M_f : N \to N$ *by:* $D_f \stackrel{\text{def}}{=} f(2, \ldots, 2)$, $E_f \stackrel{\text{def}}{=} \lceil \log_2 D_f \rceil$, $M_f(x) \stackrel{\text{def}}{=} D_f x^{E_f} + D_f$.

Lemma 4.7 *For every polynomial* $f : N^k \to N$ *the following holds:*

1. D_f *is greater or equal to the sum of all the coefficients of the polynomial* f *in the long normal form, and thus greater or equal to each coefficient of* f.
2. E_f *is greater or equal to the degree of* f *(the largest sum of the exponents in the long normal form of the polynomial).*

3. $\forall x_1, \ldots, x_k\ f(x_1, \ldots, x_k) \leq f(x_{max}, \ldots, x_{max}) \leq M_f(x_{max})$, *where* $x_{max} = \max_{1 \leq i \leq k}\{x_i\}$.

Proof.

1. Let α_j be the subsequent coefficients of f (coefficients of the monomials in the long normal form of f). Then
$$D_f = f(2, \ldots, 2) = \sum_j \alpha_j 2^{e_j} \geq \sum \alpha_j \geq \alpha_i\ \forall i$$

2. Assume by contradiction that E_f is smaller than the degree of f. It means that f has (in the form of sum of monomials) the component $\alpha x_1^{e_1} \cdots x_k^{e_k}$, such that $\alpha > 0$, $\sum_{i=1}^k e_i > E_f$. Then $D_f = f(2, \ldots, 2) > 2^{E_f}$ which contradicts $D_f \leq 2^{E_f}$ (from the definition of E_f).

3. Let α_j be the subsequent coefficients of f. The inequality
$$\forall x_1, \ldots, x_k\ f(x_1, \ldots, x_k) \leq f(x_{max}, \ldots, x_{max})$$
follows from the fact that every polynomial is monotonic. The second inequality
$$\forall x\ f(x, \ldots, x) \leq M_f(x)$$
is a consequence of the following :
$$\begin{array}{l}\forall x\ f(x, \ldots, x) = \sum_j \alpha_j x^{e_j} \leq \left(\sum_j \alpha_j\right) x^{E_f} + \left(\sum_j \alpha_j\right) \leq \\ D_f x^{E_f} + D_f = M_f(x)\end{array}$$

□

Lemma 4.8 *Given polynomial $f : N^k \rightarrow N$. If the point $(a_1, \ldots, a_k) \in N^k$ satisfies two conditions:*

1. $a_1 > D_f$
2. $\forall 2 \leq i \leq k\ a_i > M_f(a_{i-1})$

then there is an algorithm to extract values of all the coefficients of the polynomial f from the value of f for $(a_1, \ldots, a_k)$.

Proof. Induction on the number of polynomial arguments.

1. $k = 1$

   From the condition 1. and the lemma 4.7 it comes that f and a_1 satisfy the condition of the lemma 4.5. From the lemma 4.5, the values of all the coefficients of the polynomial f can be computed from the value of f for a_1.

2. $k > 1$

   Assume that the lemma holds for all $(k-1)$-argument polynomials. Represent f in the short normal form:
$$f(x_1, \ldots, x_k) = \sum_{i=0}^{m} \alpha_i(x_1, \ldots, x_{k-1}) x_k^i$$

The condition 2. for $i = k$ implies that f and $(a_1, \ldots, a_k)$ satisfy the condition of the lemma 4.5:

$$\begin{aligned} a_k > M_f(a_{k-1}) \geq f(a_{k-1}, \ldots, a_{k-1}) \geq f(a_1, \ldots, a_{k-1}, a_{k-1}) = \\ = \sum_{i=0}^{m} \alpha_i(a_1, \ldots, a_{k-1}) a_{k-1}^i \geq \alpha_j(a_1, \ldots, a_{k-1}) \ \forall 0 \leq j \leq m \end{aligned}$$

We can see that the point $(a_1, \ldots, a_k) \in N^k$ satisfies the condition of the lemma 4.5: Therefore the values of the polynomial α_j for $(a_1, \ldots, a_{k-1})$ can be obtained from the value of f at $(a_1, \ldots, a_k)$ using the algorithm proposed in lemma 4.5. To compute the values of all the coefficients of the polynomial f it suffices to compute the values of all the coefficients of the polynomials α_j from the values of α_j at $(a_1, \ldots, a_{k-1})$. The polynomials α_j are $(k-1)$-argument polynomials, so the induction assumption can be used. It is now sufficient to show for each $j = 0, \ldots, m$ that α_j and $(a_1, \ldots, a_{k-1})$ satisfy the conditions of the lemma:

(a)

$$a_1 > D_{\alpha_j}$$

comes from:

$$a_1 > D_f = \sum_{i=0}^{m} D_{\alpha_i} 2^i \geq D_{\alpha_j}$$

(b)

$$\forall 2 \leq i \leq k-1 \ a_i > M_{\alpha_j}(a_{i-1})$$

comes from:

$$D_f \geq D_{\alpha_j} \ \Rightarrow \ E_f \geq E_{\alpha_j} \ \Rightarrow \ M_f(x) \geq M_{\alpha_j}(x)$$
$$\forall 2 \leq i \leq k-1 \ a_i > M_f(a_{i-1}) \geq M_{\alpha_j}(a_{i-1})$$

For each $j = 0, \ldots, m$, α_j and $(a_1, \ldots, a_{k-1})$ satisfy the conditions of the lemma 4.5, so the values of all the coefficients of the polynomials α_j (which are in fact coefficients of the polynomial f) can be computed from the values of α_j for $(a_1, \ldots, a_{k-1})$ and thus the values of all the coefficients of the polynomial f can be computed from the value of f for $(a_1, \ldots, a_k)$.

□

Theorem 4.9 *There exists an algorithm that, for an arbitrary k-argument ($k \in N$) polynomial f, extracts the values of all the coefficients of f from the values of f at two vectors of arguments:*

1. $(2, \ldots, 2)$,
2. $(a_1, \ldots, a_k)$, *where $a_1, \ldots, a_k$ are obtained from the value $f(2, \ldots, 2)$ according to the algorithm presented in lemma 4.8, i.e.:*

$$\begin{aligned} a_1 &= D_f + 1 \\ a_i &= M_f(a_{i-1}) + 1 \text{ for } i = 2, \ldots, k. \end{aligned}$$

Proof. The following algorithm can be used to compute the values of all the coefficients of f:

1. Compute the value of f for $(2, \ldots, 2)$.
2. Compute D_f, E_f and M_f.

3. Compute the arguments $a_1, \ldots, a_k$ satisfying the conditions of lemma 4.8.
4. Compute the value of f for $(a_1, \ldots, a_k)$.
5. Using the method from lemma 4.8 compute the values of the coefficients of f.

$\square$

Lemma 4.10 *Let f be a polynomial such that $f \not\equiv 0$. There is no algorithm to extract the values of all the coefficients of f from the single value of the polynomial f.*

Proof. It can be easily seen that for an arbitrary k-argument ($k \geq 1$) polynomial $f \neq 0$ and an arbitrary set of arguments $(a_1, \ldots, a_k)$, there always exists such a polynomial g that: $g(a_1, \ldots, a_k) = f(a_1, \ldots, a_k)$ and $g \neq f$. Thus, the set $(a_1, \ldots, a_k)$ does not determine f, and the coefficients of f cannot be obtained from the value of f for $(a_1, \ldots, a_k)$. $\square$

5 Reconstruction of the extended polynomials

Lemma 5.1 *For every $\alpha, \beta \in N$ the following holds:*

$$\begin{aligned}
sq(sq(\alpha)) &= sq(\alpha) \\
sq(\overline{sq}(\alpha)) &= \overline{sq}(\alpha) \\
sq(\alpha \cdot \beta) &= sq(\alpha) \cdot sq(\beta) \\
sq(\alpha + \beta) &= sq(\alpha) \cdot \overline{sq}(\beta) + sq(\alpha) \cdot sq(\beta) + \overline{sq}(\alpha) \cdot sq(\beta) \\
\overline{sq}(sq(\alpha)) &= \overline{sq}(\alpha) \\
\overline{sq}(\overline{sq}(\alpha)) &= sq(\alpha) \\
\overline{sq}(\alpha \cdot \beta) &= \overline{sq}(\alpha) \cdot sq(\beta) + \overline{sq}(\alpha) \cdot \overline{sq}(\beta) + sq(\alpha) \cdot \overline{sq}(\beta) \\
\overline{sq}(\alpha + \beta) &= \overline{sq}(\alpha) \cdot \overline{sq}(\beta) \\
sq(\alpha) \cdot sq(\alpha) &= sq(\alpha) \\
sq(\alpha) \cdot \overline{sq}(\alpha) &= 0 \\
\overline{sq}(\alpha) \cdot \overline{sq}(\alpha) &= \overline{sq}(\alpha) \\
sq(\alpha) \cdot \alpha &= \alpha \\
\overline{sq}(\alpha) \cdot \alpha &= 0 \\
sq(\alpha) + \overline{sq}(\alpha) &= 1
\end{aligned}$$

Proof. All the properties above can be easily proved from the definition of sq and $\overline{sq}$ (see the theorem 3.3). $\square$

Lemma 5.2 *Every extended polynomial $f : N^k \to N$ has a unique representation in the following normal form:*

$$f(x_1, \ldots, x_k) = \sum_{Z \in \mathcal{P}(\{1,\ldots,k\})} f_Z(x_1, \ldots, x_k) SQ_Z(x_1, \ldots, x_k), \textit{ where}$$

$SQ_Z(x_1, \ldots, x_k) \stackrel{\text{def}}{=} \prod_{i \in Z} sq(x_i) \prod_{j \notin Z} \overline{sq}(x_j)$
f_Z – *polynomials of the arguments x_i for $i \in Z$ (in the normal form).*

Proof. (Existence) It can be seen from the lemma 5.1 that every extended polynomial can be represented as a sum of the products of the form:

$$a \cdot x_{i_1}^{e_1} \cdots x_{i_p}^{e_p} \cdot sq(x_{i_{p+1}}) \cdots sq(x_{i_q}) \cdot \overline{sq}(x_{i_{q+1}}) \cdots sq(x_{i_r}), \text{ where}$$

$$\forall \alpha \; i_\alpha \in \{1, \ldots, k\}, \;\; \forall \alpha \neq \beta \; i_\alpha \neq i_\beta, \;\; r \leq k, \;\; a, e_i \in N, i = 1, \ldots, p$$

Thus every extended polynomial can be represented as a sum of the products of the form:

$$a \cdot x_{i_1}^{e_1} \cdots x_{i_p}^{e_p} \cdot SQ_Z(x_1, \ldots, x_k), \text{ where}$$

$$i_1, \ldots, ip \in Z$$

(Uniqueness) The above representation is unique for the given extended polynomial f, i.e. if there are two representations of that form of the extended polynomial f, the representations are identical (all the coefficients in that representations are the same). Assume that the extended polynomials $g = f$ have the normal forms:

$$f(x_1, \ldots, x_k) = \sum_{Z \in \mathcal{P}(\{1,\ldots,k\})} f_Z(x_1, \ldots, x_k) SQ_Z(x_1, \ldots, x_k)$$

$$g(x_1, \ldots, x_k) = \sum_{Z \in \mathcal{P}(\{1,\ldots,k\})} g_Z(x_1, \ldots, x_k) SQ_Z(x_1, \ldots, x_k)$$

$$\forall x_1, \ldots, x_k \; f(x_1, \ldots, x_k) = g(x_1, \ldots, x_k) \;\; \Rightarrow$$

$$\forall x_1, \ldots, x_k \forall Z \in \mathcal{P}(\{1, \ldots, k\}) \; f_Z(x_1, \ldots, x_k) = g_Z(x_1, \ldots, x_k)$$

Since f_Z and g_Z are standard polynomials over the same variables in normal form, they are identical. Thus there is only one representation of f in the normal form. □

Example 5.3 The extended polynomial

$$f(x, y) = 2(xy + y^2)x + 3x^2(1 + y)\overline{sq}(y) + xy$$

has the normal form:

$$f(x, y) = ((x + 2x^2)y + (2x)y^2)SQ_{\{1,2\}}(x, y) + (3x^2)SQ_{\{1\}}(x, y).$$

Lemma 5.4 *Let f be an extended polynomial in normal form:*

$$f(x_1, \ldots, x_k) = \sum_{Z \in \mathcal{P}(\{1,\ldots,k\})} f_Z(x_1, \ldots, x_k) SQ_Z(x_1, \ldots, x_k)$$

Let $Y \in \mathcal{P}(\{1, \ldots, k\})$ be a set of indices and $(y_1, \ldots, y_k) \in N^k$ be a point such that $y_i \neq 0 \Leftrightarrow i \in Y$. Then:

$$f(y_1, \ldots, y_k) = f_Y(y_1, \ldots, y_k)$$

Proof. Straight from the definition of SQ_Z functions. □

Definition 5.5 *For every computable total function $f : \mathbf{N}^k \to \mathbf{N}$ define the set $Det_f \subset N^{k+1}$ which consists of $2^{k+1} - 1$ tuples:*

- *one tuple* $(0, ..., 0, f(0, ..., 0))$
- *for every non-empty set* $Z \subset \{1, ..., k\}$ *two tuples*

1. $(y_1^Z, \ldots, y_k^Z, f(y_1^Z, \ldots, y_k^Z))$ *where* $y_i^Z = \begin{cases} 0, & \text{if } i \notin Z \\ 2, & \text{if } i \in Z \end{cases}$
2. $(a_1^Z, \ldots, a_k^Z, f(a_1^Z, \ldots, a_k^Z))$ *where* $a_1^Z = D_{f_Z} + 1$ *and* $a_i^Z = M_{f_Z}(a_{i-1}^Z) + 1$

(see the definition of f_Z *in lemma 5.2 and* D_{f_Z} *and* M_{f_Z} *in definition 4.6).*

Theorem 5.6 *There is an algorithm that for an arbitrary k-argument* ($k \in N$) *extended polynomial f computes the values of all the coefficients of f from the values of f at* $(2^{k+1} - 1)$ *points and at the same time computes the deferminant set* Det_f *for* f :

1. 2^k *sets* $(y_1, \ldots, y_k)$*, where* $y_i \in \{0, 2\}$
2. $(2^k - 1)$ *sets* $(a_1^Z, \ldots, a_k^Z)$*, where* $Z \in \mathcal{P}(\{1, \ldots, k\})$*, and* $a_1^Z, \ldots, a_k^Z$ *can be computed from the values of f for the sets of arguments* $(y_1^Z, \ldots, y_k^Z)$*, where*

$$y_i^Z = \begin{cases} 0, & \text{if } i \notin Z \\ 2, & \text{if } i \in Z \end{cases}$$

according to the conditions of lemma 4.8, i.e.:

$$\begin{aligned} a_1^Z &= D_{f_Z} + 1 \\ a_i^Z &= M_{f_Z}(a_{i-1}^Z) + 1 \text{ for } i = 2, \ldots, k. \end{aligned}$$

Proof. The following algorithm can be used to compute the values of all the coefficients of f :

1. For $Z = \phi$
   (a) Compute the value of f for the set $(0, \ldots, 0)$.
   (b) This value is the value of the only coefficient of f_Z.
2. For every $Z \in \mathcal{P}(\{1, \ldots, k\})$, $Z \neq \phi$
   (a) Compute the value of f for the set $(y_1^Z, \ldots, y_k^Z)$, where

   $$y_i^Z = \begin{cases} 0, & \text{if } i \notin Z \\ 2, & \text{if } i \in Z \end{cases}$$

   (b) Compute D_{f_Z}, E_{f_Z} and M_{f_Z}.
   (c) Compute the arguments $a_1^Z, \ldots, a_k^Z$ satisfying the conditions of lemma 4.8.
   (d) Compute the value of f for the set $(a_1^Z, \ldots, a_k^Z)$.
   (e) Using the method from lemma 4.8 compute the values of the coefficients of f_Z.

It is clear from the construction above that the set Det_f is a determinant set for f. □

Lemma 5.7 *There exist the* (k)*-agument extended polynomials whose coefficients cannot be computed from less than* $(2^{k+1} - 1)$ *sets of arguments.*

Proof. Consider an arbitrary (k)-agument extended polynomial f such that:

$$\begin{aligned} &f(x_1, \ldots, x_k) = \textstyle\sum_{Z \in \mathcal{P}(\{1,\ldots,k\})} f_Z(x_1, \ldots, x_k) SQ_Z(x_1, \ldots, x_k), \\ &\text{where } \forall Z \in \mathcal{P}(\{1, \ldots, k\})\ f_Z \neq 0 \end{aligned}$$

Assume that there exists a set E of less than $(2^{k+1} - 1)$ examples determining polynomial f.

1. $(0,\ldots,0) \in E$. Then value of f_ϕ is not known and the values of f_ϕ coefficients cannot be computed.

2. $(0,\ldots,0) \notin E$. Then, for some $Y \in \mathcal{P}(\{1,\ldots,k\})$, there exists in E at most one example with the arguments $(y_1,\ldots,y_k)$ such that $y_i \neq 0 \Leftrightarrow i \in Y$. Thus at most one value of f_Y is known, so by lemma 4.10 the values of f_Y coefficients cannot be computed. $\square$

Theorem 5.8 *There exists an algorithm that takes any computable total function $f : \mathbf{N}^k \to \mathbf{N}$ as its input (in the form of the Turing machine or in the form of the system of recursive equations etc.) and effectively produces the unique extended polynomial f^* such that values of f and f^* are identical on $2^{k+1}-1$ elements of the determinant set Det_f or prints an information that there is no extended polynomial f^* identical with f on the set Det_f.*

Proof. For the purpose of constructing f^* we consider the same algorithm as presented in theorem 5.6. Define the set Det_f by examining the values of f at the set of $2^{k+1}-1$ tuples of arguments. By applying the algorithm described in theorem 5.6 we extract all the coefficients of f^* which crosses the set Det_f. In fact only 2^k points (about half) from Det_f are used to compute the coefficients of the extended polynomial f^*. When extended polynomial f^* is established, determine if values of f^* are the same as f on Det_f. If the answer is YES then extended polynomial f^* is returned. If the answer is NO then our algorithm prints an information that there is no extended polynomial f^* which coincides with f on Det_f. Let us call $\mathcal{POLY}$ the procedure describe above.

Example 5.9 Application of the procedure $\mathcal{POLY}$ to the function $f(x,y) = y^x + x + 1$.

Using construction from 5.8 we construct the following extended polynomial. For the set $Z_1 = \{1,2\}$ we get $f(2,2) = 7$ therefore $D_{Z_1} = 7,\ E_{Z_1} = 3,\ M_{Z_1}(x) = 7x^3 + 7$. Consequently we need to compute $f(8,3591) = 3591^8 + 9$. So the polynomial we get is $f^*_{Z_1}(x,y) = y^8 + x + 1$. The same must be done for sets $Z_2 = \{1\}$ and for $Z_3 = \{2\}$. Finally we obtain that $f^*_{Z_2}(x,y) = x + 1$ and $f^*_{Z_3}(x,y) = 2$ so the total configuration for the function f^* in normal form is

$$f^*(x,y) = (y^8 + x + 1)sq(x)sq(y) + (x+1)sq(x)\overline{sq}(y) + 2\overline{sq}(x)sq(y) + \overline{sq}(x)\overline{sq}(y)$$

and the determinant set Det_f is

$$\{(0,0,1),(2,2,7),(8,3591,3591^8+9),(2,0,3),(4,0,5),(0,2,2),(0,3,2)\}$$

Of course, the extended polynomial f^* and function f must agree on the following subset of Det_f

$$\{(0,0,1),(8,3591,3591^8+9),(4,0,5),(0,3,2)\}$$

Since on some other points from Det_f functions f and f^* do not agree, the result of the procedure $\mathcal{POLY}$ is to print a message that there is no extended polynomial f^* which coincides with f on Det_f.

6 Main result

Theorem 6.1 *Decidability of two following problems is equivalent.*

1. *For a given total primitive recursive function $f : \mathbf{N}^k \to \mathbf{N}$ decide whether or not $\forall \vec{x} \in N^k\ f(x) = 0$.*

2. *For a given primitive recursive function $f : \mathbf{N}^k \to \mathbf{N}$ decide whether or not f is an extended polynomial.*

Proof. $(1 \to 2)$ Suppose the $f \equiv 0$ problem is decidable. Let $\mathcal{Q}$ be a procedure accepting it. We construct an algorithm to recognize if f is an extended polynomial. The algorithm takes the function f given by a set of recursive equations as its input. Function f is sent to the procedure $\mathcal{POLY}$ which generates the set Det_f and extended polynomial f^* in normal form which is identical with f on the set Det_f if such extended polynomial f^* exists. At this stage the extended polynomial f^* is given. Then next procedure writes a system of primitive recursive equations for the extended polynomial f^*, for function $x \dot{-} y = [if\ x > y\ then\ x - y\ else\ 0]$ and accordingly for the function $(f \dot{-} f^*) + (f^* \dot{-} f)$. The system of primitive recursive equations for the function $(f \dot{-} f^*) + (f^* \dot{-} f)$ is forwarded to the procedure $\mathcal{Q}$. If $\mathcal{Q}$ accepts $(f \dot{-} f^*) + (f^* \dot{-} f)$ then it means that $f \equiv f^*$, therefore f is an extended polynomial, otherwise our procedure does not accept f. If $\mathcal{POLY}$ returns an information that there is no extended polynomial f^* which coincides with f on Det_f, then positively f is not an extended polynomial. Otherwise f in normal form should be returned.

$(2 \to 1)$ Suppose the *extended polynomial* problem is decidable. Let $\mathcal{P}$ be a procedure accepting it. We are going to construct an algorithm to recognize the $f \equiv 0$ problem. The algorithm constructed takes a function f given by a set of recursive equations as its input. Function f is sent to the procedure $\mathcal{P}$ which recognizes if f is an extended polynomial. If $\mathcal{P}$ does not accept f then surely $f \not\equiv 0$ since the function constantly equal to 0 is an extended polynomial. Therefore our algorithm does not accept f. If $\mathcal{P}$ accepts f then f is an extended polynomial. In this case we send f to the procedure $\mathcal{POLY}$ to generate f^* as a normal form of f. Then we examine if all coefficients of f^* are zeros or not. Accordingly our algorithm accepts f or not. □

Theorem 6.2 *For a given primitive recursive function f it is undecidable to recognize whether or not the function is constantly equal to* 0.

Proof. The proof is by reducing the Post correspondence problem to the $f \equiv 0$ problem. For a given Post system $S = \{(x_1,\ y_1), ..., (x_k, y_k)\}$ over $\Sigma = \{a,\ b\}$, we construct a primitive recursive function $f_S(n) = [if\ n = 0\ then\ 0\ else\ g_S(n)]$, such that S has no solution if and only if f_S is constantly equal to zero. The function g_S decides if the number n is a code of the solution of S :

$$g_S(n) = con_m(\lambda, \lambda, n),$$

where $m = k + 1$ and

$$con_m(x,\ y,\ n) = \begin{cases} 1, & \text{if } n = 0, \text{ and } x = y \\ 0, & \text{if } n = 0, \text{ and } x \neq y \\ con_m(S_1(n\ mod\ m) \circ x, S_2(n\ mod\ m) \circ y, (n\ div\ m)), & \text{if } n \neq 0 \end{cases}$$

where $\circ$ stands for concatenation of words and S_1, S_2 are selectors defined by:

$$S_1(x) = \begin{cases} \lambda & x = 0 \\ x_1 & x = 1 \\ \vdots & \\ x_k & x = k \\ \lambda & x > k \end{cases} \qquad S_2(x) = \begin{cases} \lambda & x = 0 \\ y_1 & x = 1 \\ \vdots & \\ y_k & x = k \\ \lambda & x > k \end{cases}$$

All functions used $(mod, div, \circ)$ and defined above $(f_S, g_S, con_m, S_1$ and $S_2)$ are primitive recursive ones. The function con_m used here maps the digits d_i of a base-m representation of a natural number $n = \overline{d_1 ... d_p}$ into a series of Post pairs whose numbers are d_i. The base m is chosen as $k+1$, where k is the number of pairs in the Post system. Thus, each digit except zero represents a pair. Zero is simply skipped when scanning the digits. □

Example 6.3 Let us take a Post system $S = \{(aa,\ a),\ (ba,\ ab),\ (b,\ aba)\}$. For this system we define:

$$S_1(x) = \begin{cases} \lambda & x = 0 \\ aa & x = 1 \\ ba & x = 2 \\ b & x = 3 \\ \lambda & x > 3 \end{cases} \qquad S_2(x) = \begin{cases} \lambda & x = 0 \\ a & x = 1 \\ ab & x = 2 \\ aba & x = 3 \\ \lambda & x > 3 \end{cases}$$

The sequence 312 is not a solution of the system S and we have $g_S(312) = con_{10}(\lambda,\ \lambda,\ 312) = con_{10}(ba,\ ab,\ 31) = con_{10}(aaba,\ aab,\ 3) = con_{10}(baaba,\ abaaab,\ 0) = 0 \Rightarrow f_S(312) = 0$. But $g_S(1231) = con_{10}(\lambda,\ \lambda,\ 1231) = con_{10}(aa,\ a,\ 123) = con_{10}(baa,\ abaa,\ 12) = con_{10}(babaa,\ ababaa,\ 1) = con_{10}(aababaa,\ aababaa,\ 0) = 1 \ \Rightarrow f_S(1231) = 1$. Thus Post system S has a solution 1231.

Theorem 6.4 *For a given primitive recursive function f it is undecidable to recognize whether or not the function is lambda definable.*

Proof. By theorem 3.3 λ definable functions are just extended polynomials. By theorem 6.1 decidability of identification of extended polynomials is equivalent with the decidability of identification of total 0 function. But according to theorem 6.2 this is undecidable. □

7 Undecidability of λ-definability in free algebras

In this chapter we are going to extend the result of theorem 6.4 to λ-definability in any infinite free algebra (see the section 3 about λ-representability). The proof of undecidability of the λ-definability is done by a simple reduction of the numerical primitive recursive function to some primitive recursive function on given free algebra in a way that preserves λ-definability.

Let us assume that $\mathbf{A}$ is infinite. Therefore the signature $S_A = (\alpha_1, ..., \alpha_n)$ of A must contain at least one 0 and at least one positive integer. Otherwise $\mathbf{A}$ is either empty or finite. Without loss of generality we may assume that $\alpha_1 = 0$ and $\alpha_2 > 0$. Hence let us take the signature of A as $S_A = (0, \alpha_2, ..., \alpha_n)$ where $\alpha_2 > 0$. Let us name the constructors of A by ϵ of arity $\alpha_1 = 0$ and d of arity $\alpha_2 > 0$ and $c_3, \ldots, c_n$ of arities $\alpha_3, \ldots, \alpha_n$ respectively.

Definition 7.1 *By $lm(t)$ we mean the length of the leftmost path the expresion t in algebra A.*

$$\begin{aligned} lm(\epsilon) &= 0 \\ lm(\ d(t_1, \ldots, t_{\alpha_2})\ ) &= 1 + lm(t_1) \\ lm(\ c_i(t_1, \ldots, t_{\alpha_i})\ ) &= 1 + lm(t_1) \end{aligned}$$

Definition 7.2 *Let us define the translation of numbers to terms of algebra $\mathbf{A}$ by associating with n the full α_2-ary tree $n^{\mathbf{A}}$ of height n.*

$$
\begin{aligned}
0^{\mathbf{A}} &= \epsilon \\
(n+1)^{\mathbf{A}} &= d(n^{\mathbf{A}}, \dots, n^{\mathbf{A}})
\end{aligned}
$$

For any n it is obvious that $lm(n^{\mathbf{A}}) = n$.

Definition 7.3 *Let us define the primitive recursive function "tree" in algebra* $\mathbf{A}$ *by:*

$$
\begin{aligned}
tree(\epsilon) &= \epsilon \\
tree(\ d(t_1, \dots, t_{\alpha_2})\ ) &= d(tree(t_1), \dots, tree(t_1)) \\
tree(\ c_i(t_1, \dots, t_{\alpha_i})\ ) &= d(tree(t_1), \dots, tree(t_1))
\end{aligned}
$$

It is easy to observe that for any term t in $\mathbf{A}$ and for the number n which is the leftmost depth of the term t $tree(t) = n^{\mathbf{A}}$ which means $tree(t) = lm(t)^{\mathbf{A}}$.

Definition 7.4 *Define the translation scheme for all primitive recursive functions on numbers into the primitive recursive functions on algebra* $\mathbf{A}$*. We associate with each primitive recursive numerical function* f *the primitive recursive function* $f^{\mathbf{A}}$ *on algebra* $\mathbf{A}$ *with the same arity. The coding is carried out by structural induction.*

Coding the zero function: for function $f(\vec{x}) = 0$ we define $f^{\mathbf{A}}(\vec{t}) = \epsilon$. Coding succesor: for $f(x) = x + 1$ we define $f^{\mathbf{A}}(t) = d(tree(t), \dots, tree(t))$. Coding projections: for $f(x_1, \dots, x_k) = x_i$ define $f^{\mathbf{A}}(t_1, \dots t_k) = tree(t_i)$. Coding composition: if f is given by: $f(\vec{x}) = g(h_1(\vec{x}), \dots, h_n(\vec{x}))$ and $g^{\mathbf{A}}, h_1{}^{\mathbf{A}}, \dots, h_n{}^{\mathbf{A}}$ are already defined then $f^{\mathbf{A}}(\vec{t}) = g^{\mathbf{A}}(h_1{}^{\mathbf{A}}(\vec{t}), \dots, h_n{}^{\mathbf{A}}(\vec{t}))$. Coding primitive recursion: if f is given by $f(0, \vec{x}) = g(\vec{x})$ and $f(y+1, \vec{x}) = h(f(y, \vec{x}), \vec{x})$ and functions $g^{\mathbf{A}}, h^{\mathbf{A}}$ are already defined then $f^{\mathbf{A}}$ is defined by the recursion scheme in algebra $\mathbf{A}$ as:

$$
\begin{aligned}
f^{\mathbf{A}}(\epsilon, \vec{t}) &= g^{\mathbf{A}}(\vec{t}) \\
f^{\mathbf{A}}(d(t_1, \dots, t_{\alpha_2}), \vec{t}) &= h^{\mathbf{A}}(f^{\mathbf{A}}(t_1, \vec{t}), \vec{t}) \\
f^{\mathbf{A}}(c_i(t_1, \dots, t_{\alpha_i}), \vec{t}) &= h^{\mathbf{A}}(f^{\mathbf{A}}(t_1, \vec{t}), \vec{t})
\end{aligned}
$$

Lemma 7.5 *For any primitive recursive numerical function* f *and for all expresions* $t_1, \dots, t_k$

$$
f^{\mathbf{A}}(t_1, \dots t_k) = (f(lm(t_1), \dots, lm(t_k))^{\mathbf{A}}
$$

Proof. Structural induction on primitive recursive functions. □

Definition 7.6 *Let* $\tau^{\mathbf{A}}$ *be a type representing algebra* $\mathbf{A}$ *in typed* λ*-calculus and let* $\tau^{\mathbf{N}}$ *be the type of Church numerals. We are going to define* λ*-terms* $CODE$ *and* $DECODE$ *of types* $\tau^{\mathbf{A}} \to \tau^{\mathbf{N}}$ *and* $\tau^{\mathbf{N}} \to \tau^{\mathbf{A}}$*, respectively, by:*

$$
\begin{aligned}
CODE &= \lambda tux.tx(\lambda y_1 \dots y_{\alpha_2}.uy_1)(\lambda y_1 \dots y_{\alpha_3}.x) \dots (\lambda y_1 \dots y_{\alpha_n}.x) \\
DECODE &= \lambda nxpy_3 \dots y_n.n(\lambda y.p\underbrace{y \dots y}_{\alpha_2})x
\end{aligned}
$$

It is obvious that for any number $n \in \mathbf{N}$ and for any term t in algebra $\mathbf{A}$:

$$\begin{aligned} CODE\underline{t} &= \underline{lm(t)} \\ DECODE\underline{n} &= \underline{n^{\mathbf{A}}} \end{aligned}$$

Theorem 7.7 *For any primitive recursive numerical function $f : \mathbf{N}^k \to \mathbf{N}$, f is λ-definable if and only if $f^{\mathbf{A}}$ is λ-definable.*

Proof. Let $f : \mathbf{N}^k \to \mathbf{N}$ be primitive recursive function, λ-definable by a term F. Define a term $F^{\mathbf{A}}$ of type $\left(\tau^{\mathbf{A}}\right)^k \to \tau^{\mathbf{A}}$ as:

$$\lambda t_1 \ldots t_k . DECODE(F(CODEt_1) \ldots (CODEt_k))$$

We are going to prove that $f^{\mathbf{A}}$ is λ-definable by $F^{\mathbf{A}}$ which means that $\underline{f^{\mathbf{A}}(t_1, \ldots, t_k)} = F^{\mathbf{A}}\underline{t_1} \ldots \underline{t_k}$ for all expresions $t_1, \ldots, t_k$.

$$\begin{aligned}
F^{\mathbf{A}}\underline{t_1} \ldots \underline{t_k} &= DECODE(F(CODE\underline{t_1}) \ldots (CODE\underline{t_k}) && \text{by the definition} \\
&= DECODE(F\ \underline{lm(t_1)} \ldots \underline{lm(t_k)}) && \text{by equation in 7.6} \\
&= DECODE(\underline{f(lm(t_1), \ldots, lm(t_k)}) && \text{by assumption} \\
&= \underline{f(lm(t_1), \ldots, lm(t_k))^{\mathbf{A}}} && \text{by equation in 7.6} \\
&= \underline{f^{\mathbf{A}}(t_1, \ldots, t_k)} && \text{by lemma 7.5}
\end{aligned}$$

Let us assume now that function $f^{\mathbf{A}} : \mathbf{A}^k \to \mathbf{A}$ is λ-definable by the term $F^{\mathbf{A}}$. Define the term F by:

$$\lambda n_1 \ldots n_k . CODE(F^{\mathbf{A}}(DECODE(n_1) \ldots DECODE(n_k)).$$

In order to prove λ-definability of f we need to show that $F\underline{n_1} \ldots \underline{n_k} = \underline{f(n_1, \ldots, n_k)}$ for all numbers $n_1, \ldots, n_k$.

$$\begin{aligned}
F\underline{n_1} \ldots \underline{n_k} &= CODE(F^{\mathbf{A}}(DECODE\underline{n_1}) \ldots (DECODE\underline{n_k})) && \text{by definition of F} \\
&= CODE(F^{\mathbf{A}}(\underline{n_1{}^{\mathbf{A}}}) \ldots (\underline{n_k{}^{\mathbf{A}}})) && \text{by equation in 7.6} \\
&= CODE(\underline{(f^{\mathbf{A}}(n_1{}^{\mathbf{A}}, \ldots, n_k{}^{\mathbf{A}})}) && \text{by assumption} \\
&= CODE(\underline{f(lm(n_1{}^{\mathbf{A}}), \ldots, lm(n_k{}^{\mathbf{A}}))}^{\mathbf{A}}) && \text{by lemma 7.5} \\
&= CODE(\underline{f(n_1, \ldots, n_k)^{\mathbf{A}}}) && \text{by equation in 7.2} \\
&= \underline{lm(f(n_1, \ldots, n_k)^{\mathbf{A}}} && \text{by equation at 7.6} \\
&= \underline{f(n_1, \ldots, n_k)^{\mathbf{A}}} && \text{by equation in 7.2}
\end{aligned}$$

□

Theorem 7.8 *λ-definability in algebra $\mathbf{A}$ is undecidable if and only if $\mathbf{A}$ is infinite*

Proof. ($\Leftarrow$) Suppose λ-definability in algebra $\mathbf{A}$ is decidable. Prove that λ-definability for primitive recursive numerical functions is decidable. Given primitive recursive function. Using coding from 7.2 one can produce the system of primitive recursive functions in $\mathbf{A}$. Employing the hypothetical procedure one can decide if the coded system of primitive recursive functions on $\mathbf{A}$ is λ-definable. Hence, using the observation from theorem 7.7 determine the λ-definability of the primitive recursive numerical functions we begun with.

($\Rightarrow$) If $\mathbf{A}$ is finite then every function is λ-definable (see theorem 3.3 and [13]), hence the problem is decidable. □

References

[1] K. Govindarajan. A note on Polynomials with Non-negative Integer Coeffitients. Private communication; to appear in the *American Mathematical Monthly*.

[2] J. Haas and B. Jayaraman. Interactive Synthesis of Definite Clause Grammars. In: K.R. Apt (ed.) *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, MIT Press, November 1992, 541-555.

[3] J. Haas and B. Jayaraman. From Context-Free to Definite-Clause Grammars: A Type-Theoretic Approach. *Journal of Logic Programming*, to appear in 1996.

[4] M. Hagiya. From Programming by Example to Proving by Example. In: T. Ito and A. R. Meyer (eds.), *Proc. Intl. Conf. on Theoret. Aspects of Comp. Software*, 387-419, Springer-Verlag LNCS 526.

[5] M. Hagiya. Programming by Example and Proving by Example using Higher-order Unification. In: M. E. Stickel (ed.) *Proc. 10th CADE*, Kaiserslautern, Springer-Verlag LNAI 449, July 1990, 588-602.

[6] D. Leivant. Functions over free algebras definable in the simple typed lambda calculus. *Theoretical Computer Science* **121** (1993), 309-321.

[7] R. Loader. The Undecidability of λ-Definability. Private communication; will be published in forthcoming Alonzo Church Festschrift book.

[8] H. Schwichtenberg. Definierbare Funktionen in λ-Kalkuli mit Typen. *Arch. Math. Logik Grundlagenforsch.* **17** (1975/76), 113–144.

[9] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science* **9** (1979), 73-81.

[10] R. Statman. On the existence of closed terms in the typed λ-calculus. In: R. Hindley and J. Seldin, eds. Combinatory Logic, Lambda Calculus and Formalism (Academic Press, New York, 1980).

[11] R. Statman. Equality of functionals revisited. In: L.A. Harrington et al. (Eds.), *Harvey Friedman's Research on the Foundations of Mathematics*, North-Holland, Amsterdam, (1985), 331-338.

[12] D.A. Wolfram. The Clausual Theory of Types. *Cambridge Tracts in Theoretical Computer Science* **21** Cambridge University Press 1993.

[13] M. Zaionc. On the λ-definable higher order Boolean operations. *Fundamenta Informaticæ* **XII** (1989), 181–190.

[14] M. Zaionc. λ-definability on free algebras. *Annals of Pure and Applied Logic* **51** (1991), 279-300.