

Linear Recursive Functions^{*}

Sandra Alves^{1**}, Maribel Fernández², Mário Florido¹, and Ian Mackie^{3***}

¹ University of Porto, Department of Computer Science & LIACC, R. do Campo Alegre 823, 4150-180, Porto, Portugal

² King's College London, Department of Computer Science, Strand, London WC2R 2LS, U.K.

³ LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

Abstract. With the recent trend of analysing the process of computation through the linear logic looking glass, it is well understood that the ability to copy and erase data is essential in order to obtain a Turing-complete computation model. However, erasing and copying do not need to be explicitly included in Turing-complete computation models: in this paper we show that the class of partial recursive functions that are syntactically linear (that is, partial recursive functions where no argument is erased or copied) is Turing-complete.

Keywords: Recursion theory, linear calculi, iteration, computable functions

1 Introduction

In the definition of recursive functions, together with recursion, a key mechanism in the process of computation is the ability for functions to duplicate and to discard their arguments (i.e., management of resources: erase and copy). In this paper we focus on this aspect of computation, which has attracted a great deal of attention in recent years. We say that a function is *linear* if it uses its arguments exactly once.

Primitive recursive functions, which we shall call PR, are a class of functions which form an important building block on the way to a full formalisation of computability. Intuitively speaking, (partial) recursive functions are those that can be computed by some Turing machine. Primitive recursive functions can be computed by a specific class of Turing machines that always halt. Many of the functions normally studied in number theory, and approximations to real-valued functions, are primitive recursive: addition, division, factorial, exponential, finding the n^{th} prime, and so on [9]. In fact, it is difficult to devise a function that

^{*} Research partially supported by the Treaty of Windsor Grant: “Linearity: Programming Languages and Implementations”, and by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *FEDER/POSI*.

^{**} Programa Gulbenkian de Estímulo à Investigação.

^{***} Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

is not primitive recursive; Ackermann’s function is a well-known example of a non-primitive recursive function.

The class of PR functions is the least set including the zero, successor and projection functions, and closed under the operations of composition and primitive recursion. In the definition of PR, zero and successor give access to the natural numbers and the projection functions are useful for erasing, copying and permuting arguments. Copying and erasing (i.e., the ability for functions to duplicate and to discard their arguments) are key operations in the definition of all interesting functions by primitive recursion, and in particular in the definition of the two operations used to define PR itself: composition and the primitive recursive scheme.

In this context the following question arises: can we define the class of primitive recursive functions without explicitly relying on copying and erasing? In this paper we show that the answer is yes; more precisely, we show that any primitive recursive function can be defined using a syntactically *linear* system. Furthermore, we show that any computable function can be defined using a single minimisation operator and linear functions. This yields an alternative formulation of the theory of recursive functions, where each function is linear; we call this class of functions *linear recursive functions*.

To define linear primitive recursive functions, which we shall call LPR, we start by specifying a set of linear initial functions (projections are not linear, so we will use natural numbers and the identity function), together with composition of linear functions and a linear primitive recursive scheme (i.e., primitive recursion where each function uses its arguments exactly once). Linear primitive recursive functions offer an *implicit* approach to copying and erasing. We can express both the process of copying a number and the process of erasing a number, as linear primitive recursive functions. Thus, the classes PR and LPR coincide.

Summarising, our main contributions are:

- Definition of linear primitive recursive functions (LPR)—a class of functions defined by the initial functions zero, successor, and identity, together with linear composition and *pure* iteration.
- Simulation of erasing and copying in LPR, in particular, projections can be simulated by permutation followed by a linear erasing. Using this result, we show that LPR and PR are exactly the same class of functions.
- Any general recursive function (i.e., any computable function) can be obtained if we add a minimisation operator working on linear primitive recursive functions.

This work exhibits a redundancy in the definition of recursive functions, and shows that a minimalistic definition of primitive recursion, based on linear functions, is sufficient. This is one more indication of the power of linear functions.

Related work: There are several alternative definitions of the primitive recursion scheme [22, 15, 16, 21, 10]. In some of these works, for instance [16, 10], primitive

recursion was replaced by *pure iteration*. Pure iteration is a linear scheme, in the sense that arguments of functions are used exactly once. Gladstone [16] gave a definition of primitive recursion using the standard initial functions and composition, but replaced primitive recursion by pure iteration. Here we refine this definition by replacing also the initial functions (by linear initial functions) and the composition scheme (by a linear composition scheme). We then show that this defines a set of linear functions that corresponds exactly to the primitive recursive functions.

Burroni [10] defined a category of primitive recursive functions with rather intuitive graphic descriptions of its objects. The definition of PR in this category is very close to ours (in particular iteration is also used instead of primitive recursion). The main difference from our work is the underlying approach: Burroni uses a categorical approach, while we use a standard recursion theory approach in the definition of PR. As an example of this, in [10], the construction of natural numbers uses an axiom similar to the Peano-Lawvere [19] axiom (more suitable in a categorical approach), instead of the Peano axioms which build numbers using the successor function.

There are several formalisms based on the notion of linearity that limit the use of copy and erasing. This includes languages based on a version of the λ -calculus with a type system corresponding to intuitionistic linear logic [12]. This calculus (which can be seen as a minimal functional programming language) provides *explicit* syntactical constructs for copying and erasing terms (corresponding to the exponentials in linear logic) [1].

From another perspective there have been a number of calculi, again many based on linear logic, for capturing specific complexity classes ([6, 11, 14, 7, 18, 25, 8]). One of the main examples is that of *bounded linear logic* [14], which captures the class of polynomial time computable functions.

This paper is part of a research project which aims at studying the notion of linearity in computation, and at analysing the computation power of linear systems. The results described here provide the foundations for a series of results, including the definition of a linear version of Gödel's System \mathcal{T} [2] with a decidable typing system for polymorphic iteration [4]. Current research in this area includes the definition of an alternative version of System \mathcal{T} which uses the linear λ -calculus and pure iteration with standard (monomorphic) linear types, without losing any of the computational power of Gödel's original definition. This paper is also a starting point for studying the connection between cartesian closed categories and symmetric monoidal closed categories. This work, which began in [20], studies this question using the internal languages (which are respectively the λ -calculus and the linear λ -calculus).

In the next section we recall the background material. In Section 3 we define the linear primitive recursive functions, and in Section 4 we show how any PR function can be encoded as an LPR function and vice versa. In Section 5 we define linear recursive functions and show that any computable function can be written as a linear recursive function. Section 6 briefly discusses higher-order primitive recursion. Finally we conclude the paper in Section 7.

2 Background

We assume familiarity with recursion theory, and recall some basic notions along the lines presented in [23]. We refer the reader to [23] for more details.

2.1 Primitive recursive functions

Notation: We use x_1, \dots, y_1, \dots to represent natural numbers, f, g, h to represent functions and X_1, \dots to represent sequences of the form x_1, \dots, x_n . We only have tuples on natural numbers, thus we will work modulo associativity for simplicity: $(X_1, (x_1, x_2), X_2) = (X_1, x_1, x_2, X_2)$.

Definition 1 (Primitive recursive functions). *A function $f : \text{Nat}^k \rightarrow \text{Nat}$ is primitive recursive if it can be defined from a set of initial functions using composition and the primitive recursive scheme defined as:*

- *Initial functions:*
 1. *The natural numbers, built from 0 and the successor function S. (We write n or $\text{S}^n 0$ for $\underbrace{\text{S} \dots \text{S}}_n 0$.)*
 2. *Projection functions: $\text{pr}_i^n(x_1, \dots, x_n) = x_i$ ($1 \leq i \leq n$); we omit the superindex when there is no ambiguity.*
- *Composition, which allows us to define a primitive recursive function h using auxiliary functions f, g_1, \dots, g_n where $n \geq 0$: $h(X) = f(g_1(X), \dots, g_n(X))$.*
- *The primitive recursive scheme, which allows us to define a recursive function h using two auxiliary primitive recursive functions f, g :*

$$\begin{aligned} h(X, 0) &= f(X) \\ h(X, \text{S } n) &= g(X, h(X, n), n). \end{aligned}$$

In [16] it was shown that primitive recursion could be replaced by a more restricted recursion scheme, called *pure iteration*:

$$\begin{aligned} h_g(X, 0) &= X \\ h_g(X, \text{S } n) &= g(h_g(X, n)). \end{aligned}$$

The function $h_g(X, n)$, obtained by the last scheme, is the result of applying the function g n times to X . Hence we may write $h_g(X, n)$ to denote the function $g^n(X)$. In the sequel we sometimes use the notation h_f for the operator that iterates f (using the pure iteration scheme).

We do not have constant functions of the form $C(X) = x$ as initial functions. However, we can see 0 as a constant function with no arguments, and every other constant function can be built by composition of 0 and S, and projections. For instance, the constant function $\text{zero}(x, y) = 0$ is defined as an instance of composition (using the initial, 0-ary function 0) and $\text{one}(x, y) = \text{S}(\text{zero}(x, y))$, again as an instance of the composition scheme.

Note also, that functions obtained from primitive recursive functions by introducing “dummy” variables, permuting variables, or repeating variables, are also primitive recursive functions. To keep our definitions simple, we will sometimes omit the definition of those functions. We give some examples below.

Example 1. Consider the standard functions add and mul from Nat^2 to Nat :

$$\text{add}(x, y) = x + y \quad \text{mul}(x, y) = x * y$$

The function add can be defined by primitive recursion as follows:

$$\begin{aligned} \text{add}(x, 0) &= f(x) \\ \text{add}(x, \mathbf{S} n) &= g(x, n, \text{add}(x, n)) \end{aligned}$$

where

$$\begin{aligned} f(x) &= pr_1(x) = x \\ g(x_1, x_2, x_3) &= \mathbf{S}(pr_3(x_1, x_2, x_3)) = \mathbf{S}(x_3) \end{aligned}$$

The primitive recursive function mul is defined by:

$$\begin{aligned} \text{mul}(x, 0) &= f(x) \\ \text{mul}(x, \mathbf{S} n) &= g(x, n, \text{mul}(x, n)) \end{aligned}$$

where

$$\begin{aligned} f(x) &= 0 \\ g(x_1, x_2, x_3) &= \text{add}(pr_1(x_1, x_2, x_3), pr_3(x_1, x_2, x_3)) = \text{add}(x_1, x_3) \end{aligned}$$

In the sequel, we consider primitive recursive functions from Nat^k to Nat^l , since every primitive recursive function from Nat^k to Nat^l can be transformed into a primitive recursive function from Nat^k to Nat , and vice versa (see [24] for details).

2.2 Recursive functions

Definition 2 (Minimisation). Let f be a total function from Nat^{n+1} to Nat . The function g from Nat^n to Nat is called the minimisation of f and is defined as: $g(X) = \min\{y \mid f(X, y) = 0\}$. We denote g as $\mu_y(f)$.

Definition 3 (Recursive functions). The set of (partial) recursive functions is defined as the smallest set of functions containing the natural numbers (built from 0 and the successor function \mathbf{S}) and the projection functions, and closed by composition, primitive recursion and minimisation.

In particular, every primitive recursive function is recursive (since in both definitions we use the same initial functions, composition and primitive recursive scheme). Closure by minimisation implies that for every $n \geq 0$ and every total recursive function $f : \text{Nat}^{n+1} \rightarrow \text{Nat}$, the function $M_f : \text{Nat}^n \rightarrow \text{Nat}$ defined by $M_f = \mu_y(f)$ is a recursive function.

We recall the following result from Kleene [17].

Theorem 1 (The Kleene normal form). Let h be a partial recursive function on Nat^k . Then, a number n can be found such that

$$h(x_1, \dots, x_k) = f(\mu_y(g(n, x_1, \dots, x_k, y)))$$

where f and g are primitive recursive functions.

3 Linear primitive recursive functions

Definition 4. A function $f : \text{Nat}^k \rightarrow \text{Nat}^j$ is linear primitive recursive if it can be defined from a set of linear initial functions using linear composition and the linear primitive recursive scheme defined as follows:

- Initial functions:
 1. The natural numbers, built from 0 and the successor function S . We write n or $S^n 0$ for $\underbrace{S \dots (S 0)}_n$.
 2. The identity function (Id).
- Linear composition, which allows us to define a function h using auxiliary linear primitive recursive functions f, g_1, \dots, g_k ,

$$h(x_1, \dots, x_n) = f(g_1(X_1), \dots, g_k(X_k)),$$

where $(X_1, \dots, X_k) = (x_1, \dots, x_n)$.

- Pure iteration, which allows us to define a recursive function h_g using an auxiliary linear primitive recursive function g :

$$\begin{aligned} h_g(X, 0) &= X \\ h_g(X, S n) &= g(h_g(X, n)). \end{aligned}$$

Note that the condition $(X_1, \dots, X_k) = (x_1, \dots, x_n)$ in the definition of linear composition above is simply a concise way of saying that each argument of the function h must be used exactly once in the composition (it may seem that this condition also restricts the order in which arguments are used, but we will show below that permutations can be defined as primitive recursive functions).

Example 2. A simple example of a linear primitive recursive function is addition. It can be defined as follows, where $g(x) = S(x)$:

$$\begin{aligned} \text{add}_g(x, 0) &= x \\ \text{add}_g(x, S n) &= g(\text{add}_g(x, n)). \end{aligned}$$

Gladstone [16] defined the primitive recursive functions using the standard initial functions and composition, with pure iteration as a recursion scheme (indeed the name comes from [16]). Burroni [10], from a categorical approach, gave a definition of PR that is very close to ours, except that a different construction of natural numbers is given.

3.1 Some useful linear primitive recursive functions

Erasing the last element of a tuple. The function $\mathcal{E}_{\text{last}}$, which erases the last element of a tuple, is defined as:

$$\begin{aligned} \mathcal{E}_{\text{last}}(X, 0) &= X \\ \mathcal{E}_{\text{last}}(X, S n) &= \text{Id}(\mathcal{E}_{\text{last}}(X, n)) \end{aligned}$$

Lemma 1. For any X and number n , $\mathcal{E}_{\text{last}}(X, n) = X$.

Proof. By induction: $\mathcal{E}_{\text{last}}(X, 0) = X$ and $\mathcal{E}_{\text{last}}(X, \mathbb{S} n) = \text{Id}(\mathcal{E}_{\text{last}}(X, n)) = \text{Id}(X) = X$.

We can use the fact that we can erase the last element of a tuple, to erase at any position. We define \mathcal{E}_i , the function that erases element i of a tuple as:

$$\mathcal{E}_i(x_1, \dots, x_n) = \text{Id}(\mathcal{E}_{\text{last}}(x_1, \dots, x_i), \text{Id}(x_{i+1}, \dots, x_n))$$

Lemma 2. $\mathcal{E}_i(x_1, \dots, x_n) = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$.

Proof. Follows from the correctness of $\mathcal{E}_{\text{last}}$.

The zero function. Using the function that erases the last element one can define a function **zero**, such that $\text{zero}(X) = 0$. The function **zero** is defined as:

$$\text{zero}(x_1, \dots, x_n) = \underbrace{\mathcal{E}_{\text{last}} \dots \mathcal{E}_{\text{last}}}_n(0, x_1, \dots, x_n)$$

Lemma 3. For any X , $\text{zero}(X) = 0$.

Proof. By induction on the length of the tuple X .

Linear copying. We now define copying using pure iteration. The function C_k^1 produces k copies of a number:

$$\begin{aligned} f(x_1, \dots, x_k) &= \text{Id}(\mathbb{S}x_1, \dots, \mathbb{S}x_k) \\ C_k^1(n) &= h_f(\underbrace{(0, \dots, 0)}_k, n), \end{aligned}$$

where h_f denotes the function obtained by using the pure iteration scheme with the auxiliary function f .

Lemma 4. For any number n , and any $k > 0$, $C_k^1(n) = \underbrace{(n, \dots, n)}_k$.

Proof. By induction: $C_k^1(0) = h_f(\underbrace{(0, \dots, 0)}_k, 0) = \underbrace{(0, \dots, 0)}_k$, and

$$C_k^1(\mathbb{S} n) = h_f(\underbrace{(0, \dots, 0)}_k, \mathbb{S} n) = f(h_f(\underbrace{(0, \dots, 0)}_k, n)) = f(\underbrace{(n, \dots, n)}_k) = \underbrace{(\mathbb{S} n, \dots, \mathbb{S} n)}_k.$$

This can be generalised to copy tuples. We use C_j^i to denote copying j times a tuple with i elements. We first show how to define C_2^2 , then after encoding permutations we define C_j^i .

$$C_2^2(x_1, x_2) = h(C'(C_2^1(x_1)), x_2)$$

where

$$\begin{aligned} h(x_1, x_2, x_3, x_4, 0) &= (x_1, x_2, x_3, x_4) \\ h(x_1, x_2, x_3, x_4, \mathbf{S} n) &= C''(h(x_1, x_2, x_3, x_4)) \\ C'(x_1, x_2) &= \text{Id}(\text{putZero}(x_1), \text{putZero}(x_2)) \\ \text{putZero}(x_1) &= \text{Id}(x_1, 0) \\ C''(x_1, x_2, x_3, x_4) &= \text{Id}(\text{Id}(x_1), \mathbf{S}(x_2), \text{Id}(x_3), \mathbf{S}(x_4)) \end{aligned}$$

Lemma 5. $C_2^2(x_1, x_2) = (x_1, x_2, x_1, x_2)$.

Proof. Using the definition, $C_2^2(x_1, x_2) = h((x_1, 0, x_1, 0), x_2)$. The result follows by induction on x_2 .

Permutations. We now define the swapping function $\pi(x, y) = (y, x)$, as follows:

$$\pi(x, y) = \mathcal{E}_1(\mathcal{E}_4(C_2^2(x, y)))$$

Permutations on any tuple can be obtained from composition of swappings and the identity. We will use π to denote permutation functions.

Having defined permutations, we define C_j^i as:

$$C_j^i(x_1, \dots, x_i) = \pi(C_j^1(x_1), \dots, C_j^1(x_i))$$

where $\pi(\underbrace{x_1, \dots, x_1}_j, \dots, \underbrace{x_i, \dots, x_i}_j) = (x_1, \dots, x_i, \dots, x_1, \dots, x_i)$.

To simplify notation, we use C to denote C_j^i , when there is no ambiguity.

Example 3. Using the erasing and copying functions, we can define multiplication as a linear primitive recursive function.

$$\begin{aligned} \text{mul}(x, y) &= \mathcal{E}_{\text{last}}(\text{mul}'_g(0, x, y)) \\ \text{mul}'_g(x_1, x_2, 0) &= (x_1, x_2) \\ \text{mul}'_g(x_1, x_2, \mathbf{S} n) &= g(\text{mul}'_g(x_1, x_2, n)) \end{aligned}$$

where $g(x_1, x_2) = f(x_1, C(x_2))$, and $f(x_1, x_2, x_3) = (\text{add}(x_1, x_2), x_3)$.

Using these ideas we will define a systematic translation of primitive recursive definitions into linear primitive recursive functions.

4 From linear primitive to primitive and back

In this section we show that every primitive recursive function is linear primitive recursive. We also show that linear primitive recursive functions do not add any power to primitive recursive functions, i.e., the two classes coincide.

4.1 Primitive recursive functions are linear primitive recursive

A summary of the encoding of primitive recursive functions using linear primitive recursive functions is given as follows:

Primitive recursive	Linear primitive recursive
0 and S	0 and S
projections	permutations + linear erasing
composition	linear composition + linear copying + linear erasing
recursive scheme	pure iteration + linear copying + linear erasing

Projections. There are many alternative definitions of projections using linear primitive recursive functions, for instance:

$$pr_i(x_1, \dots, x_n) = \underbrace{\mathcal{E}_{\text{last}} \cdots \mathcal{E}_{\text{last}}}_{n-1}(x_i, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$$

where $(x_i, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = \pi(x_1, \dots, x_n)$.

Lemma 6. For any (x_1, \dots, x_n) : $pr_i(x_1, \dots, x_n) = x_i$.

Proof. By induction on n .

- Basis: $pr_1(x_1) = x_1$.
- Induction:

$$\begin{aligned} pr_i(x_1, \dots, x_n) &= \underbrace{\mathcal{E}_{\text{last}} \cdots \mathcal{E}_{\text{last}}}_{n-1}(x_i, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) \\ &= \underbrace{\mathcal{E}_{\text{last}} \cdots \mathcal{E}_{\text{last}}}_{n-2}(x_i, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_{n-1}) = x_i. \end{aligned}$$

Multiple projection can be defined as follows:

$$pr_I(X) = \underbrace{\mathcal{E}_{\text{last}} \cdots \mathcal{E}_{\text{last}}}_{n-k}(X_1, X_2)$$

where $I = \{i_1, \dots, i_k\} \subseteq \{1, \dots, n\}$, $(x_{i_1}, \dots, x_{i_k}, X_2) = \pi(X)$.

Lemma 7. $pr_{\{i_1, \dots, i_k\}}(x_1, \dots, x_n) = (x_{i_1}, \dots, x_{i_k})$.

Proof. By induction on $n - k$.

- $pr_{\{1, \dots, n\}}(x_1, \dots, x_n) = (x_1, \dots, x_n)$.
- Induction:

$$\begin{aligned} pr_{\{i_1, \dots, i_k\}}(x_1, \dots, x_n) &= \underbrace{\mathcal{E}_{\text{last}} \cdots \mathcal{E}_{\text{last}}}_{n-k}(x_{i_1}, \dots, x_{i_k}, x_{j_1}, \dots, x_{j_{n-k}}) \\ &= \underbrace{\mathcal{E}_{\text{last}} \cdots \mathcal{E}_{\text{last}}}_{n-k-1}(x_{i_1}, \dots, x_{i_k}, x_{j_1}, \dots, x_{j_{n-k-1}}) \\ &= (x_{i_1}, \dots, x_{i_k}). \end{aligned}$$

Composition. We now define composition (see Definition 1), using linear primitive recursive functions. Let $h(X) = f(g_1(X), \dots, g_k(X))$ where $X = x_1, \dots, x_n$, and assume there are linear primitive recursive functions f^L , and g_1^L, \dots, g_k^L such that

$$\begin{aligned} f^L(Y) &= f(Y) \\ g_i^L(Z) &= g_i(Z), \quad (1 \leq i \leq k). \end{aligned}$$

Then we define h using the linear composition scheme as follows:

$h(X) = f_L^L(\pi(C_k(x_1), \dots, C_k(x_n)))$, where

$$\begin{aligned} f_L^L(X') &= f^L(g_1^L(X_1), \dots, g_k^L(X_k)) \text{ and} \\ X' = (X_1, \dots, X_k) &= (\underbrace{x_1, \dots, x_n}_{X_1}, \dots, \underbrace{x_1, \dots, x_n}_{X_k}) = \pi(\underbrace{x_1, \dots, x_1}_k, \dots, \underbrace{x_n, \dots, x_n}_k). \end{aligned}$$

Primitive recursive scheme. We now define the primitive recursive scheme of Definition 1, using linear primitive recursive functions. Let f^L and g^L be such that for the auxiliary functions f and g in the primitive recursive scheme we have:

$$\begin{aligned} f^L(X) &= f(X) \\ g^L(X, x, n) &= g(X, x, n). \end{aligned}$$

We define h^L in the following way: $h^L(X, n) = pr_1(h_{g_1}(f_1(C(X)), 0, n))$, where

$$\begin{aligned} f_1(X_1, X_2) &= (f^L(X_1), X_2) \\ f_2(X, x, n) &= (X, x, n, X, n) = \pi(C(X), x, C(n)) \\ g_2(X, x, n, X, n) &= (g^L(X, x, n), X, S n) \\ g_1(x, X, n) &= g_2(f_2(X, x, n)) \quad (x, X, n) = \pi(X, x, n). \end{aligned}$$

Lemma 8. For any X , and number n , $h_{g_1}(f_1(C(X)), 0, n) = (h(X, n), X, n)$.

Proof. By induction: $h_{g_1}(f_1(C(X)), 0, 0) = (f^L(X), X, 0) = (h(X, 0), X, 0)$ and

$$\begin{aligned} h_{g_1}(f_1(C(X)), 0, S n) &= g_1(h_{g_1}(f_1(C(X)), 0, n)) \\ &= g_1(h(X, n), X, n) \\ &= g_2(f_2(X, h(X, n), n)) \\ &= (g^L(X, h(X, n), n), X, S n) = (h(X, S n), X, S n). \end{aligned}$$

Lemma 9. For any X , and number n : $h^L(X, n) = h(X, n)$.

Proof. $h^L(X, n) = pr_1(h_{g_1}(f_1(C(X)), 0, n)) = pr_1(h(X, n), X, n) = h(X, n)$.

Example 4. The primitive recursive function `mul` defined in Example 1, can be encoded as the following linear primitive recursive function:

$$\text{mul}^L(x_1, x_2) = pr_1(h_{g_1}(f_1(C(x_1)), 0, x_2))$$

with

$$\begin{aligned} f_1(x_1, x_2) &= (f^L(x_1), x_2) \\ g_1(x_1, x_2, x_3) &= g_2(f_2(x_1, x_2, x_3)) \\ f_2(x_1, x_2, x_3) &= \pi(C(x_1), x_2, C(x_3)) = (x_1, x_2, x_3, x_1, x_3) \\ g_2(x_1, x_2, x_3, x_4, x_5) &= (g^L(x_1, x_2, x_3), x_4, S x_5) \end{aligned}$$

Assuming g^L to be the encoding of $\text{add}(pr_1(x_1, x_2, x_3), pr_3(x_1, x_2, x_3))$ and $f^L(x) = \mathcal{E}(x)$. Notice that a more simple encoding of this function is possible as was shown in Example 3.

4.2 Linear primitive recursive functions are primitive recursive

A summary of the encoding of linear primitive recursive functions using primitive recursive functions is given as follows:

Linear primitive recursive	Primitive recursive
0 and S	0 and S
Id	projection
linear composition	composition + projection
pure iteration	recursive scheme + projection

Identity. This is just a trivial projection $\text{Id}(x) = pr_1^1(x)$.

Linear composition. We now define linear composition, using primitive recursive functions. Let f^P , and g_1^P, \dots, g_k^P be such that

$$\begin{aligned} f^P(X) &= f(X) \\ g_i^P(X_i) &= g_i(X_i), \quad (1 \leq i \leq k). \end{aligned}$$

and $(X_1, \dots, X_k) = \pi(X)$. Then we define linear composition as

$$h(X) = f^P(g_1^P(X), \dots, g_k^P(X))$$

where $g_i^P(X) = g_i^P(pr_{I_i}(X))$, with $I_i = \{i_1, \dots, i_m\} \subseteq \{1, \dots, n\}$, and if $X = x_1, \dots, x_n$, then $X_i = x_{j_{i_1}}, \dots, x_{j_{i_m}}$.

Pure Iteration. Let g^P be a primitive recursive function from Nat^k to Nat^l , such that for the auxiliary function g in the pure iteration scheme we have: $g^P(X) = g(X)$.

We define h^P in the following way: $h^P(X, n) = h_{g_1}(X, n)$, where, if $X = x_1, \dots, x_n$, $Y = y_1, \dots, y_l$

$$\begin{aligned} f^P(X) &= X \\ g_1(X, Y, n) &= g^P(pr_{\{n+1, \dots, n+l\}}(X, Y, n)) = g^P(Y). \end{aligned}$$

Lemma 10. For any $X = x_1, \dots, x_k$, and number n , $h^P(X, n) = h_g(X, n)$.

Proof. By induction on n .

– Basis: $h^P(X, 0) = f(X) = X = h_g(X, 0)$.

– Induction:

$$\begin{aligned}
h^P(X, Sn) &= g_1(X, h(X, n), n) \\
&= g^P(pr_{\{n+1, \dots, n+l\}}(X, h_{g_1}(X, n), n)) \\
&= g^P(pr_{\{n+1, \dots, n+l\}}(X, h^P(X, n), n)) \\
&= g^P(h^P(X, n)) \\
&= g(h_g(X, n)) \\
&= h_g(X, Sn).
\end{aligned}$$

Using the encodings described above, we define the following translation functions:

- Definition 5.** – *Let f be a primitive recursive function. Then $\llbracket f \rrbracket_L$ will denote a linear primitive recursive function such that: $f(X) = \llbracket f \rrbracket_L(X)$.*
- *Let f be a linear primitive recursive function. Then $\llbracket f \rrbracket_P$ will denote a primitive recursive function such that: $f(X) = \llbracket f \rrbracket_P(X)$.*

We thus obtain the main result of this section:

Theorem 2. *Every PR function is LPR, and vice versa. That is $PR = LPR$.*

5 Minimisation of linear functions

5.1 Partial linear recursive functions

The minimisation operator defined in Section 2 (see Definition 2) can also be applied to linear functions.

Definition 6 (Minimisation). *Let f be a linear recursive function from Nat^{n+1} to Nat . The minimisation of f , written $\mu_y(f)$, is the function from Nat^n to Nat defined as: $\mu_y(f)(X) = \min\{y \mid f(X, y) = 0\}$.*

Definition 7 (Linear recursive functions). *The set of linear recursive functions (LRF) is the smallest set containing the initial functions 0 , S and Id (that is, natural numbers and the identity function, as in Definition 4) and closed by linear composition, pure iteration, and minimisation.*

In particular, every linear primitive recursive function is linear recursive, and for every $n \geq 0$ and every total linear recursive function $f : \text{Nat}^{n+1} \rightarrow \text{Nat}$, the function $M_f : \text{Nat}^n \rightarrow \text{Nat}$ defined by $M_f = \mu_y(f)$ is a linear recursive function.

5.2 From recursive to linear recursive

Theorem 3. *Let h be a (partial) recursive function on Nat^k . Then there exists a linear recursive function h^L on Nat^k , such that: $h(x_1, \dots, x_k) = h^L(x_1, \dots, x_k)$.*

Proof. Let h be a recursive function on Nat^k . Then, by Kleene's theorem, there exists f and g primitive recursive, and a number n , such that $h(x_1, \dots, x_k) = f(\mu_y(g(n, x_1, \dots, x_k, y)))$.

Consider then the function $h^L(x_1, \dots, x_k) = \llbracket f \rrbracket_L(\mu_y(\llbracket g \rrbracket_L(n, x_1, \dots, x_k, y)))$. Notice that

$$\begin{aligned} g(n, x_1, \dots, x_k, y) &= \llbracket g \rrbracket_L(n, x_1, \dots, x_k, y) \\ \Rightarrow \mu_y(g(n, x_1, \dots, x_k, y)) &= \mu_y(\llbracket g \rrbracket_L(n, x_1, \dots, x_k, y)) \\ \Rightarrow f(\mu_y(g(n, x_1, \dots, x_k, y))) &= \llbracket f \rrbracket_L(\mu_y(\llbracket g \rrbracket_L(n, x_1, \dots, x_k, y))). \end{aligned}$$

Thus

$$\begin{aligned} h(x_1, \dots, x_k) &= f(\mu_y(g(n, x_1, \dots, x_k, y))) \\ &= \llbracket f \rrbracket_L(\mu_y(\llbracket g \rrbracket_L(n, x_1, \dots, x_k, y))) \\ &= h^L(x_1, \dots, x_k). \end{aligned}$$

The function h^L is linear recursive.

An alternative proof could be written using the fact that closing isomorphic sets of functions with the same minimisation functor gives isomorphic sets.

Corollary 1. *All computable functions are linear recursive.*

6 Primitive recursion at higher types

It is well known that a primitive recursion scheme at higher types permits the representation of all the functions that are provably total in Peano Arithmetic (see e.g., [13]). The most commonly known formalism for this is Gödel's System \mathcal{T} . Using and extending techniques similar to the ones shown in Section 4 we can also prove that:

Theorem 4. *All functions provably total in Peano Arithmetic are linear.*

The essential points in the proof of this theorem are the mechanisms for copying and erasing functions; we refer the reader to [3] for the technical details.

7 Conclusion

The aim of this paper is to demonstrate that linear computations are powerful: linear recursive functions can express copying and erasing, thus all Turing computable functions are linear recursive. Moreover, without minimisation but with the addition of higher-order constructs, any function definable in Gödel's System \mathcal{T} is linear.

Acknowledgements

We thank Yves Lafont for pointing out [10] to us, and Andrew Pitts for comments on a previous version of this paper. A preliminary version of this paper was presented at [5].

References

1. S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.
2. S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In *Proceedings of Computer Science Logic, CSL 2006*, volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2006.
3. S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel’s System T revisited. Technical Report TR-07-02, King’s College London, 2007.
4. S. Alves, M. Fernández, M. Florido, and I. Mackie. Iterator types. In H. Seidl, editor, *Proceedings FOSSACS*, volume 4423 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2007.
5. S. Alves, M. Fernández, M. Florido, and I. Mackie. Very primitive recursive functions. In *Computation and Logic in the Real World, CiE 2007*, Quaderni del Dipartimento di Scienze Matematiche e Informatiche “Roberto Magari”, 2007.
6. A. Asperti. Light affine logic. In *Proc. Logic in Computer Science (LICS’98)*, pages 300–308. IEEE Computer Society, 1998.
7. A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Transactions on Computational Logic*, 3(1):137–175, 2002.
8. P. Baillot and V. Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS’04)*, volume 2987 of *LNCS*, pages 27–41. Springer-Verlag, 2004.
9. W. S. Brainerd and L. H. Landweber. *Theory of Computation*. John Wiley & Sons, Inc., New York, NY, USA, 1974.
10. A. Burroni. Récursivité graphique, I : Catégorie des fonctions récursives primitives formelles. *Cahiers de topologie et géométrie différentielle catégoriques*, XXVII:49, 1986.
11. J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.
12. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
13. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
14. J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992.
15. M. Gladstone. A reduction of the recursion scheme. *J. Symb. Logic*, 32, 1967.
16. M. Gladstone. Simplification of the recursion scheme. *J. Symb. Logic*, 36, 1971.
17. S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.
18. Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.
19. F. Lawvere. An elementary theory of the category of sets. *Proc. Nat. Acad. Sci.*, 52, 1964.
20. I. Mackie, L. Román, and S. Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.
21. P. Odifreddi. *Classical recursion theory*. Elsevier Science, 1999.
22. R. Robinson. Primitive recursive functions. *Bull. Am. Math. Soc.*, 53, 2004.
23. J. Shoenfield. *Recursion Theory*. Springer-Verlag, 1993.
24. J. Stern. *Fondements Mathématiques de L’Informatique*. Ediscience International, Paris, 1994.
25. K. Terui. Light affine calculus and polytime strong normalization. In *Proc. Logic in Computer Science (LICS’01)*. IEEE Computer Society, 2001.