Predicative Functional Recurrence and Poly-space

Daniel Leivant[1] and Jean-Yves Marion[2]

[1] Departments of Computer Science, Indiana University, Bloomington, IN 47405 USA.
[2] Université Nancy 2, CRIN - CNRS & INRIA Lorraine - B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France.

Abstract. We formulate a notion of predicative function types, and define predicative recurrence over functions, both in equational style and as an applicative formalism, pointing out the equivalence between the two approaches. We then show that a function is poly-space iff it is defined using predicative functionals obtained by ramified recurrence over words.

1 Introduction

Recurrence schemas have been used for long as an algebraic method for defining, characterizing, and classifying natural collections of computable functions. Characterizations by recurrence of computational complexity classes that are relevant to computer science originate with Cobham's [5] characterization of the poly-time functions over $\mathbb{N}$ by "bounded recursion on notations." The nature of the correspondence between sub-recursion and computational complexity was greatly clarified by the use of ramified data, by Bellantoni and Cook [3], (with [20] and [11] as independent precursors). Characterizations by ramified recurrence have been provided, for example, for the poly-time functions [3, 12, 14], the extended polynomials [11], the linear space functions [1, 7, 12, 18], NC^1 and polylog space [4], NP and the poly-time hierarchy [2], and the elementary functions [13]. For further background on ramified recurrence see [14].

Recurrence in higher type (i.e. types of higher rank) goes back at least to Hilbert [9], who showed that Ackermann's function can be obtained by recurrence in type $\iota \to \iota$ (where ι is the base type). More generally, Gödel proved [6] that the numeric functions defined using recurrence in all types are precisely the provably recursive functions of first-order arithmetic. In [13] we showed that when data is ramified, recurrence in all finite type generates a much smaller class, namely the functions computable in (Kalmar-) elementary resources.

This remains true even if only two tiers of data are used, since exponentiation can be defined by recurrence over functions of type $\iota_0 \to \iota_0$, as follows. (Here ι_0 is the base tier, and ι_1 is the tier of objects that can drive recurrence over ι_0). Define explicitly the function $\underline{\mathrm{dbl}}$, of type $(\iota_0 \to \iota_0) \to (\iota_0 \to \iota_0)$, by $\underline{\mathrm{dbl}}\,(f)(x) = f(f(x))$; now define by recurrence over $\iota_0 \to \iota_0$ the functional e of type $\iota_1 \to (\iota_0 \to \iota_0)$: $e(\mathbf{0}) = \mathbf{s}$, $e(\mathbf{s}n) = \underline{\mathrm{dbl}}\,(e(n))$. Then $e(n)(x) = 2^n + x$, and so base 2 exponentiation is defined by $\lambda n.\, e(n)(\mathbf{0})$.

However, from a predicative viewpoint higher type recurrence of the kind above is suspect, on grounds similar to Nelson's predicative critique of Peano Arithmetic ([17], see also [14, 15]). Nelson's complaint about Peano Arithmetic is that the natural numbers are conceptualized using induction, which itself depends on assuming a complete understanding of the natural numbers when formulas proved by induction refer to natural numbers via their free or bound variables. The same critique applies to recurrence in lieu of induction. The predicative critique excludes, in fact, the admission of any type α of the form $(\tau \to \sigma) \to \tau$, where τ is a type with an infinite extension and σ is a non-singleton type: such α presupposes the notion of arbitrary functions of type $\tau \to \sigma$, which is possible only if τ is completely delineated. But if we have an object Φ of type α, then new objects of type τ might become definable via the use of Φ; these type τ objects are, therefore, defined in terms of the entire type τ, an anathema to the predicative viewpoint.[3]

Based on the critique above, we formulate here a notion of *predicative types*, and define predicative recurrence over functions, both in an equational-algebraic style and as an applicative formalism, pointing out the equivalence between these two approaches. We then show that a function is poly-space iff it is defined using predicative functionals obtained by ramified recurrence over word algebras. For a survey of other machine independent characterizations of the poly-space functions, see [16], which itself provides such a characterization in terms of ramified recurrence with substitution (this is quoted as Theorem 1 below).

2 Recurrence

2.1 Recurrence over free algebras

Most subrecursive and applicative delineations of complexity classes are based, explicitly or implicitly, on data structures other than the natural numbers. We find it cleaner and clearer to refer directly to recurrence over free algebras.[4] For the rest of the paper $\mathbb{A}$ will be a free *word* algebra generated from $\mathbf{c}_1 \ldots \mathbf{c}_k$ $(k > 0)$, where $arity(\mathbf{c}_i) = r_i$ is 0 or 1. The 0-ary constructors are dubbed *sources* and the unary ones *successors*. If $a \in \mathbb{A}$, we write $|a|$ for the length of a. The algebras with no successor or no source are trivial. The remaining ones fall into two classes with respect to computational behaviors: the ones with one successor, $\mathbf{s}$ say, epitomize by the algebra $\mathbb{N}$ which has also a unique source, $\mathbf{0}$; and the ones with several successors, epitomized by the algebra $\mathbb{W}$ with one source ϵ and two successors, which we choose to denote $\mathbf{0}$ and $\mathbf{1}$. Since $\mathbb{W}$ is isomorphic to $\{0,1\}^*$, the canonical medium of computational complexity theory, it is the

[3] The situation is different for types $\tau \to \tau$, where the domain and range can be generated simultaneously. This critique is akin to the old predicative development of type theory (cf. e.g. [24, 23]), which requires that the arguments of a predicate R of level ℓ be defined without reference to objects of level $\geq \ell$.

[4] These schemas are for the most part well known; see, for example, [22, 21].

most important algebra in relating recurrence to computational complexity. See e.g. [16] for further discussion and examples.

Recurrence over $\mathbb{A}$ is a set of k templates, one for each constructor:

$$\begin{aligned} f(\mathbf{c}_i, \vec{x}) &= g_{c_i}(\vec{x}) && (\mathbf{c}_i \text{ a source}) \\ f(\mathbf{c}_i(a), \vec{x}) &= h_{c_i}(f(a, \vec{x}), a, \vec{x}) && (\mathbf{c}_i \text{ a successor}) \end{aligned}$$

The functions h_{c_i} above are the *recurrence functions*, and the argument of f displayed first is the *recurrence argument*. The argument of h_{c_i} displayed first is its *critical argument*, the second is the *regressive argument*, and the arguments $\vec{x}$ are the *parameters*. An instance of recurrence is *flat* if all critical arguments are missing, and is *monotonic* if all regressive arguments are missing.[5] Typical examples of flat recurrence are the definitions of the conditional and predecessor functions:

$$\underline{\text{cond}}\,(\mathbf{c}_i(\{\text{possibly an argument}\}), x_1, \ldots, x_k) = x_i,$$

$$\begin{aligned} \underline{\text{pred}}\,(\mathbf{c}_i) &= \mathbf{c}_i && (\mathbf{c}_i \text{ a source}) \\ \underline{\text{pred}}\,(\mathbf{c}_i(a)) &= a && (\mathbf{c}_i \text{ a successor}) \end{aligned}$$

2.2 Recurrence with substitution

For a generic word algebra $\mathbb{A}$ *recurrence with (parameter) substitution* [19] is the schema of recurrence, with the clauses for successors generalized to permit modification of the some of the parameters in each recursive call:

$$\begin{aligned} f(\mathbf{c}_i, x_1, \ldots, x_l, \vec{y}) &= g_{c_i}(x_1, \ldots, x_l, \vec{y}) && (\mathbf{c}_i \text{ a source}) \\ f(\mathbf{c}_i(a), x_1, \ldots, x_l, \vec{y}) &= h_{c_i}(b_{i1}, \ldots, b_{im_i}, a, \vec{x}, \vec{y}) && (\mathbf{c}_i \text{ a successor}) \\ &\quad \text{where } b_{ij} = f(a, \varphi_{ij1}(\vec{x}), \ldots, \varphi_{ijl}(\vec{x}), \vec{y}) \end{aligned}$$

The previously-defined functions φ_{ijp} are the *substitution functions* of the definition.[6] The arguments $\vec{x}$ above are the *substitution parameters* of the definition.

For example, we can obtain exponentiation by defining

$$\underline{\text{xp}}\,(\mathbf{0}, u) = u \qquad \underline{\text{xp}}\,(s n, u) = \underline{\text{xp}}\,(n, 2u) \qquad 2^n = \underline{\text{xp}}\,(n, 1)$$

It is well known that recurrence with substitution over $\mathbb{N}$ is reducible to simple recurrence (see [19] §I.3), but the proof uses auxiliary functions that cannot be properly ramified. Moreover, the classical treatment in [19] does not generalize to arbitrary word algebras, for which a more complicated machinery seems to be needed. In the contexts of computational complexity, word algebras, or ramified data, it is therefore appropriate to consider this schema in its own right. For examples and further details see [16].

[5] Monotonic recurrence over $\mathbb{N}$ is often dubbed *iteration with parameters*, but the phrase *iteration* is a misnomer for algebras like $\mathbb{W}$.

[6] The reason for not joining the variables $\vec{y}$ to $\vec{x}$ will become clear below where we define a ramified version of this schema.

2.3 Functionals defined by recurrence

We refer to the usual notion of *types,* formed inductively from the symbol ι (the base type) and the binary operation symbol $\rightarrow$: *Types* $\ni \tau ::= \iota \mid (\tau) \rightarrow (\tau)$. The convention is to drop parentheses around ι, and to associate $\rightarrow$ to the right. Also, we write $\tau_1, \tau_2 \ldots, \tau_r \rightarrow \sigma$ for $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_r \rightarrow \sigma$; and if $\tau_1 \ldots \tau_r$ are all identical to τ, we write $\tau^r \rightarrow \sigma$ for the type above.

The types are interpreted semantically over $\mathbb{A}$ in the obvious way: a type τ determines a space $\mathbb{A}^\tau$, defined by recurrence on τ: $\mathbb{A}^\iota =_{\mathrm{df}} \mathbb{A}$, and $\mathbb{A}^{\tau\rightarrow\sigma}$ is the space of all functions from $\mathbb{A}^\tau$ to $\mathbb{A}^\sigma$. The *rank* of a type is defined by $rnk(\iota) = 1$, $rnk(\sigma \rightarrow \tau) = \max(1+rnk(\sigma), rnk(\tau))$. The types of rank 1, 2 and 3 are the *object, function,* and *functional types,* respectively.

An *explicit definition* of Φ of type $\tau = (\sigma_1, \ldots, \sigma_q \rightarrow \iota)$ takes the form

$$\Phi(X_1) \cdots (X_q) = E$$

where E is an applicative expression of type ι, and $X_1 \ldots X_q$ are variables of types $\sigma_1 \ldots \sigma_q$, respectively. Recurrence (on $\mathbb{A}$) over the type τ above takes the form

$$\begin{aligned} \Phi(\mathbf{c}_i)(X_1) \cdots (X_q) &= \Psi_{c_i}(X_1) \cdots (X_q) && (\mathbf{c}_i \text{ a source}) \\ \Phi(\mathbf{c}_i(a))(X_1) \cdots (X_q) &= \Theta_{c_i}(a)(\Phi(a))(X_1) \cdots (X_q) && (\mathbf{c}_i \text{ a successor}) \end{aligned}$$

2.4 Applicative notation for recurrence

It is well known that computing by recurrence equations can be expressed functionally in the typed lambda calculus augmented with recurrence operators for the corresponding types [6]. A caveat is the presence of regressive arguments (as defined above). Using the full power of recurrence permits sequence coding that bypass this difficulty (see [19] §I.4). We comment below about alternatives.

Let $\boldsymbol{\lambda}Rec(\mathbb{A})$ be the simply typed λ-calculus expanded as follows. Each constructor of $\mathbb{A}$ is a constant of the obvious type, as are the predecessor and conditional. We refer to these as the *basic constants* of the calculus. For each type τ there is, in addition, a constant $\mathbf{R}^\tau$ of type $\rho[\tau] =_{\mathrm{df}} \sigma_1, \ldots \sigma_k, \iota \rightarrow \tau$, where σ_i stands for τ if $\mathbf{c}_i$ is a source and for $\tau \rightarrow \tau$ if it is a successor. The β-reduction rules are augmented with computational rules for the predecessor, conditional, and recursor constants:

$$\begin{aligned} \underline{\mathrm{pred}}\,(\mathbf{c}_i) &\Rightarrow \mathbf{c}_i && \mathbf{c}_i \text{ a source} \\ \underline{\mathrm{pred}}\,(\mathbf{c}_i(E)) &\Rightarrow E && \mathbf{c}_i \text{ a successor} \\ \underline{\mathrm{cond}}\,(\mathbf{c}_i(\cdots), E_1, \ldots, E_k) &\Rightarrow E_i \\ \mathbf{R}^\tau\, E_1 \cdots E_k \mathbf{c}_i &\Rightarrow E_i && \mathbf{c}_i \text{ a source} \\ \mathbf{R}^\tau\, E_1 \cdots E_k(\mathbf{c}_i(a)) &\Rightarrow E_i D_i && \mathbf{c}_i \text{ a successor} \\ & \text{where } D_i = \mathbf{R}^\tau\, E_1 \cdots E_k\, a. \end{aligned}$$

See [13] for details and examples.

3 Ramified recurrence

3.1 Ramified recurrence for base type

Ramified recurrence was discovered independently by Simmons [20], Leivant [11], and Bellantoni and Cook [3]. The latter paper was the first to use tiered data to characterize computational complexity. A systematic development of data tiering and ramified recurrence was introduced in [12, 14]. We recall here the essentials.

Let $\mathcal{S}(\mathbb{A})$ be the many-sorted structure with copies $\mathbb{A}_0, \mathbb{A}_1 \ldots$ of the algebra $\mathbb{A}$ as universes, which are dubbed *tiers*. (We omit the superscript when in no danger of confusion.) *Ramified recurrence* allows the definition of a function $f : \mathbb{A}_i \times \mathcal{A} \to \mathbb{A}_j$ (where $\mathcal{A}$ is the product of some $\mathbb{A}_m$'s), by a recurrence schema as above, provided $i > j$:

$$\begin{aligned} f(\mathrm{c}_i, \vec{x}) &= g_{\mathrm{c}_i}(\vec{x}) && (\mathrm{c}_i \text{ a source}) \\ f(\mathrm{c}_i(a), \vec{x}) &= h_{\mathrm{c}_i}(f(a, \vec{x}), a, \vec{x}) && (\mathrm{c}_i \text{ a successor}) \end{aligned}$$

Note that the recurrence argument is in $\mathbb{A}_i$ and the output is in $\mathbb{A}_j$, $j < i$ by the explicit typing requirement for f. The *ramified PR functions over* $\mathbb{A}$ are the functions over $\mathcal{S}(\mathbb{A})$ that are defined from the basic constants of $\mathbb{A}$ by explicit definitions and ramified recurrence. In [14] we proved that, over word algebras, these are precisely the functions computable in polynomial time.

3.2 Ramified recurrence with substitution

Ramified $\mathbb{A}$-recurrence with substitution is the schema of recurrence with substitution as stated above, but referring to $\mathcal{S}(\mathbb{A})$, and with the provisos that the recurrence argument and substitution arguments all be in the same tier, and that that tier be greater than the tier of the result.[7]

In [16] we introduced a generic notion of alternating register machine over free algebras. Computability in polynomial time over such a machine over $\mathbb{W}$ is equivalent to poly-time computability over an alternating Turing machine, and thus to poly-space. We proved there the following.

Theorem 1. *The functions over a word algebra $\mathbb{A}$ defined by ramified $\mathbb{A}$-recurrence with substitution are precisely the functions computable on an alternating register machine for $\mathbb{A}$. Therefore, the functions over $\mathbb{W}$ defined by ramified recurrence with substitution are precisely the poly-space functions.*

3.3 Predicative recurrence over functions

To keep matters uncluttered, *we restrict attention here to the first three tiers only.*[8] Following the predicative critique of recurrence over functions, outlined

[7] Note that the remaining parameters of the definition, for which we used $\vec{y}$ in the schema above, may be in any tier. Allowing the substitution parameters to have different tiers permits a definition for exponentiation.

[8] Two tiers won't suffice for a predicative recurrence over functions, as will become clear below.

in the introduction, we restrict higher order recurrence operators $\mathbf{R}^\tau$ to function types τ for which the type $\tau \to \tau$, appearing as an argument, is predicatively admissible. We require that no subtype of τ appears in $\tau \to \tau$ as both a domain of a function argument and a co-domain. In addition, we require, in analogy to ramified recurrence for object type, that the recurrence argument be of tier greater than any tier in τ. These conditions imply that recurrence argument be of tier ι_2, and τ be of the form $\iota_1^m \to \iota_0$ or $\iota_0^m \to \iota_1$. Since there are no definable functions of the latter type (see discussion in [14]), we are left with the former.

We call types of the form $\iota_1^m \to \iota_0$ $(m \geq 1)$ *safe function types*, and say that a type is *safe* if it is either an object type (ι_0, ι_1, or ι_2), or a safe function type. A type of the form $\tau_1, \ldots \tau_m \to \iota_0$, where all τ_i's are safe, is a *predicative functional type*. Note that if τ is a predicative functional type then $\sigma \to \tau$ is again a predicative functional type for any safe σ. Observe that $\iota_0 \to \iota_0$ is a (degenerated) predicative functional type, but is not a safe function type: it is admissible on its own, but is not "safe" as an argument.

An applicative formalism $P\lambda Rec^2(\mathbb{A})$ for predicative second order recurrence is now defined as the following extension of the simply typed λ-calculus.

- For each basic constant of $\mathbb{A}$ and each tier we have an identifier for the function operating in that tier. We shall indicate the appropriate tier by a superscript, but often omit it when it is clear from the context or irrelevant.
- For each safe function type τ, and each tier ι_j larger than all tiers in τ, there is a recurrence constant $\mathbf{R}^{\iota_j,\tau}$, whose type is $\rho[\tau] =_{\mathrm{df}} \sigma_1, \ldots, \sigma_k, \iota_j \to \tau$. Here, again, σ_i is τ if $\mathbf{c}_i$ is a source, and $\tau \to \tau$ if it is a successor.
- The β-reduction rule is augmented by the reduction rules for the predecessor, conditional, and recurrence constants, as for the un-ramified version described above.

3.4 Reduction to applicative recurrence

Lemma 2. *If a function f over $\mathbb{A}$ is defined from ramified functions by predicative recurrence, then f has a definition from those functions by monotonic predicative recurrence.*

Proof. Suppose Φ is defined by

$$\begin{aligned} \Phi(\mathbf{c}_i)(X_1)\cdots(X_q) &= \Psi_{c_i}(X_1)\cdots(X_q) && (\mathbf{c}_i \text{ a source}) \\ \Phi(\mathbf{c}_i(a))(X_1)\cdots(X_q) &= \Theta_{c_i}(a)(\Phi(a))(X_1)\cdots(X_q) && (\mathbf{c}_i \text{ a successor}) \end{aligned}$$

Define instead Φ' by

$$\begin{aligned} \Phi'(\mathbf{c}_i)(y)(X_1)\cdots(X_q) &= \Psi_{c_i}(X_1)\cdots(X_q) && (\mathbf{c}_i \text{ a source}) \\ \Phi'(\mathbf{c}_i(a))(y)(X_1)\cdots(X_q) &= \Theta_{c_i}(\underline{\mathrm{pred}}\,(y))(\Phi'(a)(\underline{\mathrm{pred}}\,(y)))(X_1)\cdots(X_q) && (\mathbf{c}_i \text{ a successor}) \end{aligned}$$

Then $\Phi(w) = \Phi'(w)(w)$. □

Corollary 3. *A function f is definable using predicative recurrence over functions iff it is computed by a term of $P\lambda Rec^2(\mathbb{A})$.*

3.5 Predicative recurrence captures recurrence with substitution

Proposition 4. *If a function f over $\mathbb{A}$ is defined by ramified recurrence with substitution, then f is defined by predicative functional recurrence.*

Proof. By induction on the definition of f. The only case of interest is where f is generated by the scheme of ramified recurrence with substitution, as above:

$$\begin{array}{rll} f(\mathbf{c}_i, \vec{x}, \vec{y}) = g_{c_i}(\vec{x}, \vec{y}) & & (\mathbf{c}_i \text{ a source}) \\ f(\mathbf{c}_i(a), \vec{x}, \vec{y}) = h_{c_i}(b_{i1}, \ldots, b_{im_i}, a, \vec{x}, \vec{y}) & & (\mathbf{c}_i \text{ a successor}) \\ \quad \text{where} \quad b_{ij} = f(a, \varphi_{ij1}(\vec{x}), \cdots, \varphi_{ij\ell}(\vec{x}), \vec{y}) & & \end{array}$$

Let σ denote the type $\iota_1^\ell \to \iota_1$ of the substitution functions φ_{ijp}, and $\rho_1 \ldots \rho_q \in \{\iota_0, \iota_1\}$ be the types of $y_1 \ldots y_q$, respectively. Let Z be a variable of type σ. Define by recurrence the functional Φ, with the clauses for successor constructors:

$$\begin{array}{rl} & \Phi(\mathbf{c}_i)(\vec{y})(\vec{x}) = g_{c_i}(\vec{x}, \vec{y}) \\ & \Phi(\mathbf{c}_i(a))(\vec{y})(\vec{x}) = G(a)(\Phi(a)(\vec{y}))(\vec{y})(\vec{x}) \\ \text{where} & G(a)(Z)(\vec{y})(\vec{x}) = h_{c_i}(Z_{i1} \ldots Z_{im_i}, a, \vec{x}, \vec{y}) \\ & \qquad \text{with} \quad Z_{ij} = Z(\varphi_{ij1}(\vec{x}), \cdots, \varphi_{ij\ell}(\vec{x})) \end{array}$$

Note that the first argument of G is in tier ι_2. □

Combining 4 with 3 we obtain:

Proposition 5. *If a function f over $\mathbb{A}$ is defined by ramified recurrence with substitution, then f is defined by a term of $P\lambda Rec^2(\mathbb{A})$.*

4 Predicative functional recurrence is poly-space

We conclude here with a proof of the main technical lemma of this work, showing that every function defined using predicative functional recurrence is computable in poly-space. To keep notations uncluttered, we restrict attention here to the algebra $\mathbb{A} = \mathbb{W}$.

As a computation model for functionals we use multi-tape Turing machines with function oracles. Each such machine refers to a fixed finite list of function oracles, of arities ≥ 1. Oracles are used via query rules, where each such rule specifies the oracle it invokes (i.e. a position in the list of oracles), and, if the arity of that oracle is m, the rules prescribe the m tapes from which the oracle inputs are to be read, and the tape on which the oracle output is overwritten. In measuring the computation time of such a machine we count each oracle call as a single step.[9] Note that a rather tame oracle, such as one for concatenation, can

[9] A similar notion underlies [10]

yield by n iterations an output exponentially larger than the input. However, we will work with machines where unbounded oracle iteration does not occur.

Going back to the formalism $P\boldsymbol{\lambda}Rec^2(\mathbb{A})$, we say that a term E is *normal* if no subterm can be reduced (by β-, recursor-, predecessor-, or conditional-reduction). It is well known that in $\boldsymbol{\lambda}Rec^2(\mathbb{A})$ every term can be converted to normal form, and so the same applies to $P\boldsymbol{\lambda}Rec^2(\mathbb{A})$, which is a more restrictive calculus.

For a term E we write $\bar{E}$ for the λ-closure of E, i.e. $\lambda x_1 \ldots x_r.\, E$, where $x_1 \ldots, x_r$ is a list of all free variables in E, under some canonical order.

In analyzing the structure of normal terms for functionals of predicative types, we have to consider additional types. Call a type *admissible* if it is of the form $\tau_1 \ldots \tau_m \to \iota_j$, $j = 1$ or 2, where the τ_i's are safe types. Call an expression E of $P\boldsymbol{\lambda}Rec^2(\mathbb{A})$ *tame* if it is normal, and its λ-closure has a predicative or admissible type. That is, E has a predicative or admissible type, and all free variables have safe type.

Proposition 6. *Let $E \equiv E[\vec{u}_0, \vec{u}_1, \vec{u}_2, \vec{V}]$ be a tame expression, where $\vec{u}_i$ are the free type-ι_i variables, and $\vec{V}$ all variables with safe function types. Let Φ be the functional over $\mathbb{A}$ defined by $\bar{E}$.*

1. *If the type of E is ι_2, then Φ evaluates in constant time, and its value is dependent only on arguments of type ι_2.*
2. *If the type of $\bar{E}$ is admissible (with ι_1 as co-domain), then Φ evaluates in time polynomial in its type ι_2 arguments, and independent from all its remaining arguments, and its value is dependent only on arguments of type ι_2 and ι_1.*
3. *If the type of $\bar{E}$ is predicative, then Φ evaluates in space polynomial in its type-ι_2 and type-ι_1 arguments and independent from its other arguments, and all queries to function arguments during the computation are for words whose length is within a polynomial in the type-ι_2 arguments from the type-ι_1 arguments.*

Proof. By induction on E. We give here key cases for E. Other cases will be spelled out in the full version of this paper.

If E is a variable x of type τ, then $\bar{E} \equiv \lambda x.\, x$ is of type $\tau \to \tau$. Since $\tau \to \tau$ is predicative or admissible, τ is either an object or a safe function type. In either case the lemma's statement holds trivially.

E cannot be a recurrence constant $\mathbf{R}^\tau$: the type $\rho[\tau]$ of $\mathbf{R}^\tau$ has $\tau \to \tau$, which is never safe, as an argument type. Thus $\mathbf{R}^\tau$ is not tame for any τ.

If E is one of the basic constants, then the lemma is trivial.[10]

If E is a λ-abstraction $\lambda y.\, E_0$, the statement is immediate by induction assumption for E_0.

[10] Note that our using constructors, predecessor, and conditional with input(s) and output of identical base type is essential here.

Suppose that E is an application, say $E \equiv E_0(E_1)\cdots(E_m)$ $(m \geq 1)$, where E_0 is no longer an application. Since E is normal, E_0 cannot be an abstraction. Therefore E is either a variable or a constant.

The type of E_0 must be of the form $(\tau_1 \ldots \tau_m) \to \sigma$, where τ_i is the type of E_i, and σ the type of E_0. If E_0 is a variable, then its type must be a safe function type, since E is tame, and so $\tau_1 = \cdots = \tau_m = \iota_1$, and $\sigma = \iota_0$. By induction assumption, it follows that case (2) of the lemma applies to $E_1, \ldots, E_m$. Therefore, given values for the free variables $\vec{u}$ and $\vec{V}$, we can compute the values of $E_1 \ldots E_m$ in time polynomial in its type ι_2 inputs, without invoking the safe function inputs as oracles. These m values are the inputs to the oracle given as global input for the variable E_0, confirming that case (3) of the lemma holds for E.

Suppose, on the other hand, that E_0 is a basic function. If it is a constructor or predecessor, the lemma holds trivially for E by induction assumption for E_1. If E_0 is $\underline{\text{cond}}^j$ $(j = 0, 1 \text{ or } 2)$, then E is of one of the forms $\underline{\text{cond}}\, E_1$, $\underline{\text{cond}}\, E_1 E_2$, or $\underline{\text{cond}}\, E_1 E_2 E_3$, and the lemma holds trivially for E by induction assumption invoked for the argument expressions E_i.

We are left with the only interesting case of the proof, where E_0 is a recursor. We treat the case where $E_0 = \mathbf{R}^{\iota_2, \iota_1^2 \to \iota_0}$, which typifies recurrence over functions. Other cases are analogous. The full generality of the situation is captured when E is of type ι_0 and of the form $\mathbf{R}^{\iota_2, \iota_1^2 \to \iota_0} E_\epsilon E_0 E_1 W U_1 U_2$.[11]

Let n, m be the maximal lengths of type-ι_2 inputs and of type-ι_1 inputs, respectively. The induction's assumption applies to all subterms F of E. For each such F, let $M(F)$ be a Turing machine that computes the function represented by $\bar{F}$ as prescribed by induction assumption. We then have:

- A constant c_2 such that all subterms of type ι_2 define functionals computed in time $\leq c_2$.
- Constants c_1 and r_1 such that all subterms of admissible type define functionals computed in time $\leq c_1 n^{r_1}$.
- Constants c_0, r_0, d and q such that all subterms of safe type define functionals computed in space $\leq c_0 \cdot \max(m, n)^{r_0}$, and such that function inputs are applied in the computation of the machines $M(E_0)$ and $M(E_1)$ only to values whose length is within $d \cdot n^q$ away from the type ι_1 inputs.

Let M be a function-oracle machine for $\bar{E}$, defined as follows. M's input consists of values assigned to the free variables of E; let us call these the *global* inputs. M starts by computing W, U_1 and U_2, given the global inputs. Let $w \in \mathbb{W}$ be the value obtained for W. By induction assumption, w differs from the type-ι_2 inputs by at most c_2. M proceeds by laying out a sequence of $|w|$ gadgets: if

[11] For instance, if $E \equiv \mathbf{R}^{\iota_2, \iota_1^2 \to \iota_0} E_\epsilon E_0 E_1 W$, of type $\iota_1^2 \to \iota_0$, then we can consider the expression $E' \equiv \mathbf{R}^{\iota_2, \iota_1^2 \to \iota_0} E_\epsilon E_0 E_1 W u_1 u_2$ instead, with u_1, u_2 variables of type ι_1, which is again of the form above; and using induction assumption for the subterms of E is all that one needs to deal with E' instead.

$w = d_k d_{k-1} \cdots d_1 \epsilon$, where $d_i \in \{\mathbf{0}, \mathbf{1}\}$, then the sequence is M_k, $M_{k-1}, \ldots, M_0$, with $M_i = M(E_{d_i})$ for $i = 1 \ldots k$, and $M_0 = M(E_\epsilon)$.

Note that each of the two machines repeated in this sequence, $M(E_0)$ and $M(E_1)$, computes a function of type $(\iota_1^2 \to \iota_0) \to (\iota_1^2 \to \iota_0)$. Therefore, each of these gadgets uses the global inputs as well as a function argument of type $\iota_1^2 \to \iota_0$, and 2 arguments of type ι_1.

M simulates the operation of M_k for the global inputs and the values of U_1, U_2. Since the U_i's are of type ι_1, these values have length $\leq c_1 n^{r_1}$. Whenever the machine M_k would query its oracle, M instructs the gadget M_k to query instead the successor gadget M_{k-1}. The same process is repeated down the sequence of gadgets $M_k \ldots M_1$. Clearly, M computes the value of E (a detailed proof is by induction on $|w|$).

Each M_i queries its successor gadget M_{i-1} for values whose lengths differ by at most $d \cdot n^q$ from the type ι_1-arguments of M_i itself, which include the global type ι_1 inputs, the values of U_1 and U_2, and the queries passed from M_{i+1}, if any. It follows that all gadgets M_i have type-ι_1 inputs that differ from the type-ι_1 global inputs and the values of of U_1 and U_2 by at most $|w| \cdot dn^q$, which is $\leq (n+c_2) \cdot dn^q$. The computation of M is, therefore, performed in some constant space needed for bookkeeping, plus a value

$$
\begin{aligned}
&\leq \text{space used to compute } W, U_1, U_2 \\
&\qquad + \text{ space used by } M_k, \ldots, M_0 \\
&\leq c_2 + 2 \cdot c_1 n^{r_1} \\
&\qquad + (n+c_2) \cdot c_0 \cdot N^{r_0} \\
&\quad \text{where} \qquad N = m + c_1 n^{r_1} + (n+c_2) \cdot dn^q
\end{aligned}
$$

This is bounded by a polynomial in $\max(m, n)$ of degree $r_0 \cdot \max(r_1, q+1) + 1$. Moreover, as noted above, all terms of type ι_1 evaluated during this computation define words whose lengths differ by at most $c_1 n^{r_1} + (n+c_2) \cdot d \cdot n^q$ from the global object inputs, and the computation is independent from the global safe function inputs. This case (3) of the proposition holds for E. □

Theorem 7. *A function over $\mathbb{W}$ is computable in polynomial space iff it is defined by a term of $P\lambda Rec^2(\mathbb{A})$.*

Proof. By Theorem 1 every poly-space function is definable by recurrence with substitution over $\mathbb{W}$, and therefore defined by a term of $P\lambda Rec^2(\mathbb{A})$, by Proposition 5.

Conversely, suppose a function is defined by a (closed) expression of $P\lambda Rec^2(\mathbb{A})$. We can assume that the defining expression E is closed and normal. It is therefore tame, and hence is computable in polynomial space, by lemma 6. □

REMARK. The ramification condition used here cannot be relaxed to allow the recurrence arguments to be of type ι_1. Otherwise we could define, for example, by recurrence over the algebra $\mathbb{N}$ of numerals,

$$E \equiv \mathbf{R}^{\iota_1,\iota_1\to\iota_0}(\lambda f\lambda x.\, f(\mathbf{s}x))$$
$$F \equiv \mathbf{R}^{\iota_1,\iota_1\to\iota_0}(\lambda g\lambda n.\, Egnn)$$

Then

$$E = \lambda g\lambda n\lambda z.\, g(z+n)$$
$$\text{and} \quad F = \lambda h\lambda m.\, h(2^m)$$

So $F\kappa$, where κ is the coercion function from tier ι_1 to tier ι_0 (see [14]), computes exponentiation.

References

1. S. Bellantoni. *Predicative Recursion and Computational Complexity.* PhD thesis, University of Toronto, 1992.
2. S. Bellantoni. Predicative recursion and the polytime hierarchy. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II, Perspectives in Computer Science*, pages 15–29. Birkhäuser, 1994.
3. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
4. S. Bloch. Functional characterizations of uniform log-depth and polylog-depth circuit families. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, pages 193–206. IEEE Computer Society Press, 1992.
5. A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1965.
6. K. Gödel. Über eine bisher noch nicht benüte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. Republished with English translation and explanatory notes by A. S. Troelstra in *Kurt Gödel: Collected Works*, Vol. II. S. Feferman, ed. Oxford University Press, 1990.
7. W.G. Handley. Bellantoni and Cook's characterization of polynomial time functions. Typescript, August 1992.
8. J.van Heijenoort. *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, MA, 1967.
9. D. Hilbert. Über das unendliche. *Mathematische Annalen*, 95:161–190, 1925. English translation in [8], pages 367–392.
10. B.M. Kapron and S.A. Cook. A new characterization of mehlhorn's polynomial time functionals. *SIAM Journal of Computing*, 25(1):117–132, Feb. 1996.
11. D. Leivant. Subrecursion and lambda representation over free algebras. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, Perspectives in Computer Science, pages 281–291. Birkhauser-Boston, New York, 1990.
12. D. Leivant. Stratified functional programs and computational complexity. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 325–333, New York, 1993. ACM.

13. D. Leivant. Predicative recurrence in finite type. In A. Nerode and Yu.V. Matiyasevich, editors, *Logical Foundations of Computer Science*, LNCS 813, pages 227–239, Berlin, 1994. Springer-Verlag. Proceedings of the Third LFCS Symposium, St. Petersburg.
14. D. Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhauser-Boston, New York, 1994.
15. D. Leivant. Intrinsic theories and computational complexity,. In D. Leivant, editor, *Logic and Computational Complexity*, number 960 in LNCS, pages 177–194. Springer-Verlag, 1995.
16. D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL 94*, pages 486–500. LNCS 933, Springer Verlag, 1995.
17. E. Nelson. *Predicative Arithmetic.* Princeton University Press, Princeton, 1986.
18. A. P. Nguyen. *A formal system for linear space reasoning.* PhD thesis, University of Toronto, 1993. Master of Science Thesis.
19. H.E. Rose. *Subrecursion.* Clarendon Press (Oxford University Press), Oxford, 1984.
20. H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.
21. J.V. Tucker and J.I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics.* CWI Monographs No. 6. North-Holland and the Centre for Mathematics and Computer Science, Amsterdam, 1988.
22. K.N. Venkataraman, A. Yasuhara, and F. M. Hawrusik. A view of computability on term algebras. *Journal of Computer and System Sciences*, 26(2):410–471, June 1983.
23. H. Wang. *Some formal details on predicative set theories*, chapter XXIV. Science Press, Peking, 1962. Republished in 1964 by North Holland, Amsterdam. Republished in 1970 under the title *Logic, Computers, and Sets* by Chelsea, New York.
24. A. N. Whitehead and B. Russell. *Principia Mathematicae.* Cambridge University Press, second edition, 1929.