Linear types and non-size-increasing polynomial time computation

Martin Hofmann*

Synopsis: We propose a linear type system with recursion operators for inductive datatypes which ensures that all definable functions are polynomial time computable. The system improves upon previous such systems in that recursive definitions can be arbitrarily nested, in particular no predicativity or modality restrictions are made.

1 Summary

Recent work has shown that predicative recursion combined with a linear typing discipline gives rise to type systems which guarantee polynomial runtime of well-typed programs while allowing for higher-typed primitive recursion on inductive datatypes.

Although these systems allow one to express all polynomial time functions they reject many natural formulations of obviously polynomial time algorithms. The reason is that under the predicativity regime a recursively defined function is not allowed to serve as step function of a subsequent recursive definition. However, in most functional programs involving inductive data structures such iterated recursion does occur. A typical example is insertion sort which involves iteration of an (already recursively defined) insertion function.

A closer analysis of such examples reveals that the involved functions do not increase the size of their input and that this is why their repeated iteration does not lead beyond polynomial time.

In this work we present a new linear type system based on this intuition. It contains unrestricted recursion operators for inductive datatypes such as integers, lists, and trees, yet ensures polynomial runtime of all first-order programs.

2 Introduction

Suppose we have a type of integers N in binary notation and constructors $0 : \mathsf{N}, \mathsf{S}_0 : \mathsf{N}\to\mathsf{N}, \mathsf{S}_1 : \mathsf{N}\to\mathsf{N}$ with semantics $\mathsf{S}_0(x) = 2x$ and $\mathsf{S}_1(x) = 2x+1$. The following defines a function $f : \mathsf{N}\to\mathsf{N}$ of quadratic growth:

$$\begin{aligned} f(0) &= 1 \\ f(x) &= \mathsf{S}_0(\mathsf{S}_0(f(\lfloor \tfrac{x}{2} \rfloor))), \text{ when } x > 0 \end{aligned}$$

More precisely, $f(x) = [x]^2$ where $[x] = 2^{|x|}$. As usual, $|x| = \lceil \log_2(x+1) \rceil$ denotes the length of x in binary notation. We also write $||x||$ for $|a|$ when $a = |x|$. Iterating f as in

$$\begin{aligned} g(0) &= 2 \\ g(x) &= f(g(\lfloor \tfrac{x}{2} \rfloor)), \text{ when } x > 0 \end{aligned}$$

leads to exponential growth, indeed, $g(x) = 2^{[x]}$.

This example is the motivation behind predicative versions of recursion as used in [2, 5]. In these systems it is forbidden to iterate a function which has itself been recursively defined. More precisely, the step function in a recursive definition is not allowed to recurse on the result of a previous function call (here $g(\lfloor \frac{x}{2} \rfloor)$), but may, however, recurse on other parameters.

If higher-order functions are allowed then a new phenomenon appears: If $h : \mathsf{N}\to\mathsf{N}\to\mathsf{N}$ is defined by

$$\begin{aligned} h(0) &= \mathsf{S}_0 \\ h(x) &= h(\lfloor \tfrac{x}{2} \rfloor) \circ h(\lfloor \tfrac{x}{2} \rfloor) \end{aligned}$$

then $h(x, y) = 2^{[x]} \cdot y$ although no recursion on results of recursive calls takes place. This example suggests to require that the step function in a recursive definition should be affine linear in the sense of linear logic, i.e., use its argument at most once.

In [1] and in [4] it has been shown that this restriction together with predicativity suffices to ensure polynomial runtime of all first-order programs.

Although these systems are very expressive they rule out many naturally occurring and obviously polynomial time algorithms. A typical example is the insertion sort algorithm defined as follows

$$\begin{aligned} &\text{insert}(a, []) = [a] \\ &\text{insert}(a, b :: l) = \mathsf{if}\ a \le b \\ &\qquad \text{then } a :: b :: l \\ &\qquad \text{else } b :: \text{insert}(a, l) \end{aligned}$$

$$\begin{aligned} &\text{sort}([]) = [] \\ &\text{sort}(a :: l) = \text{insert}(a, \text{sort}(l)) \end{aligned}$$

*LFCS Edinburgh, Mayfield Rd, Edinburgh EH9 3JZ, UK. E-mail: mxh@dcs.ed.ac.uk

The definition of insert is perfectly legal under the regime of predicative or safe recursion, but the subsequent definition of sort is not. The reason that nevertheless insertion sort does not lead to an exponential growth and runtime is that the insertion function does not increase the size of its input.

Caseiro [3] has noticed this and developed (under the name "LIN-systems") partly semantic criteria on first-order recursive programs which allow one to detect this situation and which in particular apply to the insertion sort example. The drawback of her criteria is that they are rather complicated, not obviously decidable, and that they do not generalise to higher-order functions in any obvious way.

In this paper we present a type-theoretic approach to this problem. We will develop a fairly natural linear type system which has the property that all definable functions are non-size increasing and which boasts higher-order recursion on datatypes without any predicativity restriction. We will show that nevertheless all definable first-order functions are polynomial time computable even if they contain higher-order functions as subexpressions.

As indicated above the type system will in particular ensure that programs do not increase the size of their input so that iterated recursion does not lead to exponential growth. However, this means that not all polynomial time computable functions can be definable. So, in order to obtain a complete type system we will have to combine the present system with the system in [4] based on predicative recursion. In Section 4.3 below we speculate on the expressivity of the present system alone.

3 Syntax

We use affine linear lambda calculus with products and certain inductive datatypes such as integers, lists, and trees.

The types are given by the following grammar.

$$A, B ::= \mathbf{B} \mid A \multimap B \mid A \otimes B \mid A \times B$$

where $\mathbf{B}$ ranges over a set of *base types* which is left indeterminate as yet. For example, we will introduce a base type N for integers. We will also allow ourselves to extend the above grammar by new *type operators*, notably one for lists, which associates to each type A a type of lists $\mathsf{L}(A)$.

Terms are given by

$$\begin{array}{l} e ::= \mathrm{op}(e_1, \ldots, e_n) \mid x \mid \lambda x{:}\, A.e \mid \\ (e_1\ e_2) \mid e_1 \otimes e_2 \mid \\ \mathsf{let}\, e_1{=}x \otimes y\, \mathsf{in}\, e_2 \mid \langle e_1, e_2 \rangle \mid e.1 \mid e.2 \end{array}$$

Here op ranges over a set of *operators* to be determined later and x ranges over a countable set of variables.

As usual, terms are understood as equivalence classes modulo renaming of bound variables, i.e., $\lambda x{:}\, A.x$ and $\lambda y{:}\, A.y$ are considered identical.

A *context* is a partial function from variables to types. Two contexts Γ_1, Γ_2 are called *disjoint* if $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. In this case we write Γ_1, Γ_2 for the union of Γ_1 and Γ_2. If $x \notin \mathrm{dom}(\Gamma)$ and A is a type then we write $\Gamma, x{:}\, A$ for the context $\Gamma \cup \{(x, A)\}$.

An *arity* is an expression of the form $(A_1, \ldots, A_n)A$ where $n \geq 0$ and A_i, A are types.

We fix an assignment of arities to operators.

An operator of arity $()A$ is called a constant and we write $c : A$ to mean that c is a constant of arity $()A$.

The typing judgement $\Gamma \vdash e : A$ read "e has type A in context Γ" is defined inductively by the following rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}\, A \vdash e : B}{\Gamma \vdash \lambda x{:}\, A.e : A \multimap B} \qquad \text{(T-ARR-I)}$$

$$\frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash (e_1\ e_2) : B} \qquad \text{(T-ARR-E)}$$

$$\frac{\Gamma_1 \vdash e_1 : A_1 \quad \Gamma_2 \vdash e_2 : A_2}{\Gamma_1, \Gamma_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \qquad \text{(T-TENS-I)}$$

$$\frac{\Gamma_1 \vdash e_1 : A_1 \otimes A_2 \quad \Gamma_2, x{:}\, A_1, y{:}\, A_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\, e_1{=}x \otimes y\, \mathsf{in}\, e_2 : B} \qquad \text{(T-TENS-E)}$$

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \qquad \text{(T-PROD-I)}$$

$$\frac{\Gamma \vdash e : A_1 \times A_2 \quad i \in \{1, 2\}}{\Gamma \vdash e.i : A_i} \qquad \text{(T-PROD-E)}$$

$$\frac{\mathrm{op\ has\ arity}\ (A_1, \ldots, A_n)A \quad \emptyset \vdash e_i : A_i\ \text{for}\ i = 1 \ldots n}{\Gamma \vdash \mathrm{op}(e_1, \ldots, e_n) : A} \qquad \text{(T-OP)}$$

The rules are set up in such a way that when $\Gamma \vdash e : A$ then all the free variables of e are mentioned in Γ and they are used at most once in e. To be used at most once is slightly more generous than to occur at most once. Namely, by rule T-PROD-I, a variable may occur in both components of a cartesian product ($A \times B$). For example, we have

$\lambda x{:}A.\langle x,x\rangle : A \multimap A \times A$, but not $\lambda x{:}A.x \otimes x : A \multimap A \otimes A$. There is a coercion from tensor product ($A \otimes B$) to cartesian product ($\times$), namely $\lambda z{:}A \otimes B.\mathsf{let}\ t{=}x \otimes y\ \mathsf{in}\ \langle x,y\rangle$, but not vice versa. The only "candidate" $\lambda z{:}A \times B.z.1 \otimes z.2$ is not well typed because rule TENS-I requires both components to have disjoint sets of variables.

Notice that an operator is applicable to closed terms only. This is the reason why it is not possible to encode operators by constants of functional type.

3.1 Set-theoretic interpretation

We assume for every base type A a set $[\![A]\!]$, for example $[\![\mathsf{N}]\!] = \mathbb{N}$ and extend this inductively to all types by the clauses $[\![A \multimap B]\!] = [\![A]\!] \to [\![B]\!]$, $[\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!]$, $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$. An environment for a context Γ is a function η mapping each variable $x \in \mathrm{dom}(\Gamma)$ to an element $\eta(x) \in [\![\Gamma(x)]\!]$.

We also assume an assignment of functions

$$[\![\mathrm{op}]\!] \in [\![A]\!]_1 \times \cdots \times [\![A_n]\!] \to [\![A]\!]$$

for each operator op of arity $(A_1,\ldots,A_n)A$.

Relative to such interpretation of operators we can interpret a term $\Gamma \vdash e : A$ as a function $[\![e]\!]$ mapping environments for Γ to elements of $[\![A]\!]$ in the usual way.

3.2 Size function

We want to assign a *partial* size function $\mathbf{s}_A : [\![A]\!] \to \mathbb{N}$ to every type A. To do this we assume such size function for every basic type, for example $\mathbf{s}_{\mathsf{N}}(x) = |x|$, and extend this to all types by the following inductive definition

$$\begin{array}{l}\mathbf{s}_{A\otimes B}((u,v)) = \mathbf{s}_A(u) + \mathbf{s}_B(v)\\ \mathbf{s}_{A\times B}((u,v)) = \max(\mathbf{s}_A(u),\mathbf{s}_B(v))\\ \mathbf{s}_{A\multimap B}(f) = \min\{c \mid \forall a{\in}[\![A]\!].\\ \qquad \mathbf{s}_B(f(a)) \le c + \mathbf{s}_A(a)\}\end{array}$$

In the last clause a ranges over those elements of $[\![A]\!]$ for which $\mathbf{s}_A(a)$ is defined. It is assumed that in this case $\mathbf{s}_B(f(a))$ is also defined; otherwise $\mathbf{s}_{A\multimap B}(f)$ will be undefined. It will likewise be undefined if no c with the required property exists. This is the primary source for undefinedness of $\mathbf{s}$.

Now denotations of terms are non-size-increasing in the following sense.

Proposition 3.1 *Suppose that for each operator op of arity $(A_1,\ldots,A_n)A$ and elements $v_i \in [\![A_i]\!]$ with $\mathbf{s}_{A_i}(v_i) = 0$ we have $\mathbf{s}_A([\![\mathrm{op}]\!](v_1,\ldots,v_n)) = 0$. In particular $\mathbf{s}_A([\![c]\!]) = 0$ for each constant $c : A$.*

If η is an environment for Γ such that $\mathbf{s}_{\Gamma(x)}(\eta(x))$ is defined for each $x \in \mathrm{dom}(\Gamma)$ then $\mathbf{s}_A([\![e]\!]\eta)$ is also defined and moreover

$$\mathbf{s}_A([\![e]\!]\eta) \le \sum_{x\in\mathrm{dom}(\Gamma)} \mathbf{s}_{\Gamma(x)}(\eta(x))$$

We remark that this result will not in itself be used later on; it merely serves as a motivation for the signature we are going to introduce next. The proof that all first-order functions are polynomial time computable requires a more sophisticated interpretation given in Section 5.

4 Inductive types and iteration

We will now introduce base types and operators (in particular those for recursion). This will happen in such a way that the premises to Proposition 3.1 are satisfied.

We start by introducing a type of integers N with $[\![\mathsf{N}]\!] = \mathbb{N}$ and $\mathbf{s}_{\mathsf{N}}(x) = |x|$. We further introduce a constant $0 : \mathsf{N}$ with $[\![0]\!] = 0$. Clearly, $\mathbf{s}_{\mathsf{N}}(0) = 0$.

In order to construct numerals we would like to introduce constants for the binary successor functions $\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N} \multimap \mathsf{N}$ with meaning $[\![\mathsf{S}_0]\!](x) = 2x$ and $[\![\mathsf{S}_1]\!](x) = 2x+1$. However, these functions increase the size of their argument by one and so we would have $\mathbf{s}_{\mathsf{N}\multimap\mathsf{N}}([\![\mathsf{S}_0]\!]) = 1$ rather than 0.

In order to fix this problem we introduce a new base type $\Diamond$ with interpretation $[\![\Diamond]\!] = \{\diamond\}$ and size function $\mathbf{s}_\Diamond(\diamond) = 1$. Now we can use the following typing for the successor functions

$$\begin{array}{l}\mathsf{S}_0 : \Diamond \multimap \mathsf{N} \multimap \mathsf{N}\\ \mathsf{S}_1 : \Diamond \multimap \mathsf{N} \multimap \mathsf{N}\end{array}$$

with interpretation

$$\begin{array}{l}[\![\mathsf{S}_0]\!](\diamond,x) = 2x\\ [\![\mathsf{S}_1]\!](\diamond,x) = 2x+1\end{array}$$

Now, indeed, $\mathbf{s}([\![\mathsf{S}_0]\!]) = \mathbf{s}([\![\mathsf{S}_1]\!]) = 0$ as required.

Next, for each type A we introduce an operator $\mathsf{it}^{\mathsf{N}}_A$ of arity

$$(A, \Diamond \multimap A \multimap A, \Diamond \multimap A \multimap A)\mathsf{N} \multimap A$$

for recursion on notation. The semantics of this operator is given by $[\![\mathsf{it}^{\mathsf{N}}(g,h_0,h_1)]\!] = f$ where

$$\begin{array}{l}f(0) = [\![g]\!]\\ f(2(x+1)) = [\![h_0]\!](\diamond, f(x+1))\\ f(2x+1) = [\![h_1]\!](\diamond, f(x))\end{array}$$

Induction on x shows that $\mathsf{it}^{\mathsf{N}}_A(g,h_0,h_1)$ is non-size-increasing if g,h_0,h_1 are so that Prop. 3.1 continues to hold in the presence of $\mathsf{it}^{\mathsf{N}}_A$.

Notice that the typing of $\mathsf{it}^{\mathsf{N}}_A$ as an operator rather than a higher-order constant, hence the fact that the functions

$[\![g]\!], [\![h_0]\!], [\![h_1]\!]$ do not increase the size, is crucial here. If h_0 or h_1 increase the size by a constant (as would be the case if they were allowed to be variables) then $\mathsf{it}^{\mathsf{N}}(g, h_0, h_1)$ would multiply the size by that constant thusviolating the intended invariant.

From it^{N} we can *define* an operator for primitive recursion: If $g : X$, $h_0, h_1 : \Diamond \multimap (X \times \mathsf{N}) \multimap X$ are closed terms as indicated then we can obtain

$$\mathsf{rec}^{\mathsf{N}}(g, h_0, h_1) : \mathsf{N} \multimap X$$

with semantics $[\![\mathsf{rec}^{\mathsf{N}}(g, h_0, h_1)]\!] = f$ where

$$\begin{array}{l} f(0) = [\![g]\!] \\ f(2x) = [\![h_0]\!](\star)(f(x), x) \text{ when } x > 0 \\ f(2x+1) = [\![h_1]\!](\star)(f(x), x) \end{array}$$

by invoking it^{N} with result type $A = X \times \mathsf{N}$ and parameters constructed from g, h_0, h_1 in the obvious way. This gives a function $\mathsf{N} \multimap X \times \mathsf{N}$ from which we obtain the desired function by projection.

Notice that due to the cartesian product $\times$ as opposed to $\otimes$ in a primitive recursion using $\mathsf{rec}^{\mathsf{N}}$ we can access either the recursion variable or make a recursive function call but are not allowed to do both. It is not possible to define $\mathsf{rec}^{\mathsf{N}}$ with $\otimes$ instead of $\times$.

From $\mathsf{rec}^{\mathsf{N}}$ we can in turn define a constant for case distinction:

$$\begin{array}{c} \mathsf{case}^{\mathsf{N}} : (X \times (\Diamond \multimap \mathsf{N} \multimap X) \times \\ (\Diamond \multimap \mathsf{N} \multimap X)) \multimap \mathsf{N} \multimap X \end{array}$$

with semantics

$$\begin{array}{l} [\![\mathsf{case}^{\mathsf{N}}]\!](g, h_0, h_1)(0) = g \\ [\![\mathsf{case}^{\mathsf{N}}]\!](g, h_0, h_1)(2(x+1)) = h_0(x+1) \\ [\![\mathsf{case}^{\mathsf{N}}]\!](g, h_0, h_1)(2x+1) = h_1(x) \end{array}$$

where this time the arguments g, h_0, h_1 may contain parameters. To do this, we invoke $\mathsf{rec}^{\mathsf{N}}$ with the higher-order result type

$$(X \times ((\mathsf{N} \multimap X) \times (\mathsf{N} \multimap X))) \multimap X$$

and the obvious arguments.

The cartesian product (as opposed to $\otimes$) in the type of $\mathsf{case}^{\mathsf{N}}$ is a "feature"; it means that a variable can be used in each branch and still count as linear. Notice that we only need to introduce $\mathsf{it}^{\mathsf{N}}_A$ as syntactic primitive; $\mathsf{rec}^{\mathsf{N}}$ and $\mathsf{case}^{\mathsf{N}}$ are then definable using just affine linear lambda calculus.

Example Using these building blocks it is easy to define an addition function

$$\text{add} : \mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$$

such that $[\![\text{add}]\!](x, y, c) = x + y + (c \pmod 2)$. All we need to do is to translate the obvious recursive equations into a formal definition involving $\mathsf{rec}^{\mathsf{N}}$ and $\mathsf{case}^{\mathsf{N}}$.

The obvious definition of multiplication in terms of addition is not possible since it is nonlinear; nevertheless multiplication is definable by the expressivity result below in Section 4.3.

However, it is easy to define the function $\text{pad}(x, y) = x[y] + y$.

Notice that we can *not* define the function $f : \mathsf{N} \longrightarrow \mathsf{N}$ from the introduction given by

$$\begin{array}{l} f(0) = 1 \\ f(x) = 4f(\lfloor \frac{x}{2} \rfloor) \end{array}$$

since it exhibits quadratic growth. Defining it by diagonalising pad violates linearity. The obvious formalisation of the recursive definition would use $\mathsf{it}^{\mathsf{N}}_A$ with result type $A = \mathsf{N}$ and arguments

$$\begin{array}{l} g = 0 \\ h_0 = h_1 = \lambda c{:}\Diamond.\lambda z{:}\mathsf{N}.\mathsf{S}_0(c)(\mathsf{S}_0(c)(z)) \end{array}$$

However the last definition is not type correct because the variable $c{:}\Diamond$ is used twice.

This illustrates the restricting effect of the $\Diamond$-resource. We can only apply as many constructor symbols $(\mathsf{S}_0, \mathsf{S}_1)$ as we have variables of type $\Diamond$ in our local context.

4.1 Lists and trees

Similarly, we can introduce a type of lists $\mathsf{L}(A)$ for each type A (formally by extending the grammar for the types with the clause $\ldots \mid \mathsf{L}(A) \mid$). The set-theoretic semantics of the new type former is given by $[\![\mathsf{L}]\!](X) = X^\star$ and the size function is

$$\mathbf{s}_{\mathsf{L}(A)}([a_1, \ldots, a_n]) = n + \sum_{i=1}^{n} \mathbf{s}_A(a_i)$$

The usual constructor functions for lists give rise to constants

$$\begin{array}{l} \mathsf{nil}_A : \mathsf{L}(A) \\ \mathsf{cons}_A : \Diamond \multimap A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A) \end{array}$$

Taking the length of a list to be merely the sum of the sizes of its entries would be unreasonable as the entries might all have zero size, e.g. we could have $A = \mathsf{N} \multimap \mathsf{N}$ and $a_i = \lambda x{:}\mathsf{N}.x$.

For each type X we introduce an operator $\mathsf{it}^{\mathsf{L}(A)}_X$ of arity

$$(X, \Diamond \multimap A \multimap X \multimap X)\mathsf{L}(A) \multimap X$$

with semantics $[\![\mathsf{it}^{\mathsf{L}(A)}(g, h)]\!] = f$ whenever

$$\begin{array}{l} f([]) = [\![g]\!] \\ f(a :: l) = [\![h]\!](\diamond)(a)(f(l)) \end{array}$$

As in the case of integers we can define an operator $\mathsf{rec}^{\mathsf{L}(A)}$ which from $g : X$ and $h : \Diamond \multimap A \multimap (X \times \mathsf{L}(A)) \multimap X$ constructs $\mathsf{rec}^{\mathsf{L}(A)}(g,h) : \mathsf{L}(A) \multimap X$ with the obvious semantics and also a case construct.

Likewise we can define binary labelled trees $\mathsf{T}(A)$ with constructors

$$\begin{array}{l}\mathsf{leaf} : A \multimap \mathsf{T}(A)\\ \mathsf{node} : \Diamond \multimap (\mathsf{T}(A) \otimes \mathsf{T}(A)) \multimap \mathsf{T}(A)\end{array}$$

The set $[\![\mathsf{T}(A)]\!]$ then consists of binary trees with both leaves and nodes labelled with elements of $[\![A]\!]$. The size of such a tree is given by the number of its nodes plus the sizes (w.r.t. s_A) of all its labels.

We can then justify an iteration construct $\mathsf{it}_X^{\mathsf{T}(A)}$ of arity

$$(A \multimap X, \Diamond \multimap A \multimap X \multimap X \multimap X)\mathsf{T}(A) \multimap X$$

with semantics given by $f = [\![\mathsf{it}_X^{\mathsf{T}(A)}]\!](g,h)$ iff

$$\begin{array}{l}f(\mathsf{leaf}(a)) = g(a)\\ f(\mathsf{node}(a,l,r)) = h(a, f(l), f(r))\end{array}$$

By following this pattern other inductively defined datatypes can be introduced as well.

We remark that in the definition of $f(\mathsf{node}(a,l,r))$ two recursive calls to f are made which indicates that it is not easily possible to encode trees in terms of natural numbers or lists. We also remark that the type of entries in a tree type need not be basic; it can be functional, a list type or a tree type itself. The same goes for the type of entries in a list.

We also introduce a base type of Boolean values B with $[\![\mathsf{B}]\!] = \{\mathtt{tt}, \mathtt{ff}\}$ and size function $\mathrm{s}_{\mathsf{B}}(x) = 0$. We can justify constants $\mathtt{tt} : \mathsf{B}, \mathtt{ff} : \mathsf{B}$ and a construct for case distinction

$$\mathsf{if} : \mathsf{B} \multimap A \times A \multimap A$$

The cartesian product as opposed to a tensor product signifies that a variable may occur in both branches of a case distinction without violating linearity.

4.2 Examples

Concatenation of lists $@ : \mathsf{L}(A) \multimap \mathsf{L}(A) \multimap \mathsf{L}(A)$ is definable as

$$\begin{array}{l}@ =_{\mathrm{def}} \quad \mathsf{it}^{\mathsf{L}(A)}_{\mathsf{L}(A)\multimap\mathsf{L}(A)}(\\ \qquad \lambda l{:}\mathsf{L}(A).l,\\ \qquad \lambda c{:}\Diamond.\lambda a{:}A.\lambda p{:}\mathsf{L}(A)\multimap\mathsf{L}(A).\lambda l'{:}\mathsf{L}(A).\\ \qquad\quad \mathsf{cons}(c,a,p(l)))\end{array}$$

This readily allows us to produce the list of leaf labellings of a tree (disregarding the labels in the nodes) as a function $\mathsf{leaves} : \mathsf{T}(A) \multimap \Diamond \multimap \mathsf{L}(A)$ by

$$\begin{array}{l}\mathsf{leaves} =_{\mathrm{def}} \quad \mathsf{it}^{\mathsf{T}(A)}_{\Diamond\multimap\mathsf{L}(A)}(\\ \qquad \lambda a{:}A.\lambda c{:}\Diamond.\mathsf{cons}(c,a,\mathsf{nil}),\\ \qquad \lambda c_1{:}\Diamond.\lambda a{:}A.\lambda l,r{:}\Diamond\multimap\mathsf{L}(A).\\ \qquad\quad \lambda c_2{:}\Diamond.(l\ c_1)@(r\ c_2))\end{array}$$

The extra $\Diamond$-argument is needed because there is always one more leaf than there are nodes. Similarly, we can define a function $\mathsf{nodes} : \mathsf{T}(A) \multimap \mathsf{L}(A)$ giving the list of node labellings. If we want to get the list of all labels we need another tree type in which leaves also require a $\Diamond$-argument. For our trees this function increases the size hence cannot be representable.

For a more ambitious example we will now turn to the insertion sort algorithm mentioned in the introduction. We assume a closed comparison function $\mathsf{leq} : (A \otimes A) \multimap \mathsf{B} \otimes A \otimes A$ which besides comparing two elements also gives them back for further processing. The full paper contains a more general account of this seemingly ad-hoc modification. Now, we use $\mathsf{rec}^{\mathsf{L}(A)}$ with result type $X = \Diamond \multimap A \multimap \mathsf{L}(A)$. We define $g : X$ by

$$g = \lambda x{:}\Diamond.\lambda a{:}A.\mathsf{cons}_A(x,a,\mathsf{nil}_A)$$

and $h : \Diamond \multimap A \multimap (X \times \mathsf{L}(A)) \multimap X$ by

$$\begin{array}{l}h = \lambda x{:}\Diamond.\lambda a{:}A.\lambda p{:}X \times \mathsf{L}(A).\lambda y{:}\Diamond.\lambda b{:}A.\\ \quad \mathsf{let}\ \mathsf{leq}(a\otimes b){=}t \otimes a\otimes b\ \mathsf{in} \qquad \mathsf{if}\ t\\ \qquad\quad \mathsf{cons}_A(x,a,p.1(y,b))\\ \qquad\quad \mathsf{cons}_A(x,b,\mathsf{cons}_A(y,a,p.2)))\end{array}$$

We put $\mathsf{insert}_A =_{\mathrm{def}} \mathsf{rec}_X^{\mathsf{L}(A)}(g,h)$.

If $l : \mathsf{L}(A)$ is sorted in the increasing order w.r.t. leq then so is $\mathsf{insert}_A(x,a,l)$ and its elements agree with $a :: l$.

Notice here how the use of a functional result type X in the definition of insert_A allows us to subsume its definition under the $\mathsf{rec}^{\mathsf{L}(A)}$ construct.

Now we obtain insertion sort as $\mathsf{sort} = \mathsf{it}^{\mathsf{L}(A)}(\mathsf{nil}_A, \mathsf{insert}_A)$.

Similarly, the usual functional implementations of heap sort (involving a binary search tree as an intermediate data structure) or the function of type $\mathsf{L}(\mathsf{T}(A)) \multimap \mathsf{L}(\mathsf{T}(A))$ describing one step in Huffman's algorithm are directly representable in the system. In order to represent divide-and-conquer algorithms such as quicksort one needs another recursion pattern which can also be justified semantically. The full paper will give details.

4.3 Expressivity

At present we are not in a position to characterise the functions definable in affine linear lambda calculus with the above iteration principles. The best we can offer is that all functions computable in polynomial time and simultaneously in linear space are representable.

Proposition 4.1 *Let $f : \mathsf{L}(A) \multimap \mathsf{L}(A)$ be a closed term. We can define a closed term $f^{\#} : \mathsf{L}(A) \multimap \mathsf{L}(A)$ such that*

$$[\![f^{\#}]\!](l) = f^{length(l)}(l)$$

Proof. Define $g : \mathsf{L}(A) \multimap \mathsf{L}(A) \multimap \mathsf{L}(A)$ by

$$g([])(l') = l'$$
$$g(a :: l)(l') = f(g(l)(l'@[a]))$$

where $l@l'$ denotes the concatenation of l and l' and $length(l)$ is the number of entries of l.

It is clear that this can be translated into a legal definition of g using $\mathsf{it}^{\mathsf{L}(A)}_{\mathsf{L}(A)\multimap\mathsf{L}(A)}$. Induction readily shows that $[\![g]\!](l, l') = [\![f]\!]^{length(l)}(l'@l)$.

Hence, we can put $f^{\#}(l) = g(l)(\mathsf{nil})$. □

Iterating the $\#$-operation and composition allows us to iterate f any polynomial (in $length(l)$) many times provided f does not shorten its argument. Therefore, we can represent linear space, polynomial time computable functions in the following sense:

Theorem 4.2 *Let $f : \mathbb{N} \to \mathbb{N}$ be computable in polynomial time and linear space such that moreover $|f(x)| \leq |x|$. Then f is the denotation of a closed term of type $\mathsf{N} \multimap \mathsf{N}$.*

Proof. We may assume that f is computed by a polynomially time-bounded Turing machine M which has one I/O tape and k worktapes, which is initialised by writing the input on the I/O tape and on all the worktapes and which never writes beyond the space occupied by this initialisation. The one step function of this machine can be represented as a closed term of type $W \multimap W$ where $W = \mathsf{L}(\mathsf{B} \otimes \ldots \otimes \mathsf{B})$ with $k + 1$ factors correpsonding to the $k + 1$ tapes. Iterating this function the required (polynomial) number of times and composing with initialisation and output extracting functions gives the result. □

5 Polynomial-time

Our aim for the rest of this paper will be to prove that whenever $e : \mathsf{N} \multimap \mathsf{N}$ is a closed term then $[\![e]\!]$ will be polynomial time computable. We are at present not able to show that $[\![e]\!]$ can also be computed in linear space, but are confident that this is in fact the case. Work in this direction is underway.

Our strategy is to assign certain untyped *PTIME*-algorithms to terms in such a way that realisers for first-order terms are algorithms for their set-theoretic denotations. It simplifies the presentation if we first define this realisation abstractly for an arbitrary *BCK*-algebra (the affine-linear analogue of combinatory or SK-algebra) and then show how untyped *PTIME*-algorithms can be organised as such an algebra and how the iteration constructs can be interpreted.

Definition 5.1 *A BCK-algebra is given by a set H and a function app $: H \times H \longrightarrow H$, written as juxtaposition associating to the left, and constants $B, C, K \in H$ such that $Bxyz = x(yz)$, $Cxyz = xzy$, $Kxy = x$.*

An identity combinator I with $Ix = x$ can be defined as $I = CKK$.

If H is a *BCK*-algebra and t a term in the language of *BCK*-algebras and containing constants from H then if the free variable x appears at most once in t we can find a term $\lambda x.t$ not containing x such that for every other term s the equation $(\lambda x.t)s = [s/x]t$ is valid in H, i.e., all ground instances of the equation hold in H.

For example, if x, y are variables then $\lambda f.fxy = C(CIx)y$. Further abstraction yields the pairing combinator $T = \lambda x \lambda y \lambda f.fxy$. In fact, we have $T = BC(CI)$. Another important combinator is $O = CK$. We will subsequently use untyped affine linear lambda terms to denote elements of particular *BCK*-algebras.

5.1 Realisation of affine linear lambda calculus

Fix a *BCK*-algebra H and an assignment of a relation $\Vdash_A \subseteq H \times [\![A]\!]$ for every basic type A.

Such relation can then be defined for all types by the following assignments.

$$\begin{array}{l} e \Vdash_{A\multimap B} f \iff \\ \forall a.\forall t.\, t \Vdash_A a \Rightarrow et \Vdash_B f(a) \\ e \Vdash_{A\otimes B} (a, b) \iff \\ \exists u.\exists v.\, e = Tuv \wedge u \Vdash_A a \wedge v \Vdash_B b \\ e \Vdash_{A\times B} (a, b) \iff \\ eK \Vdash_A a \wedge eO \Vdash_B b \end{array}$$

If η is an environment for context Γ and $t \in H$ then we write $t \Vdash_\Gamma \eta$ to mean that $t = Tt_1(Tt_2(T \ldots Tt_nK) \ldots)$ where $x_1, \ldots, x_n$ is an enumeration of $\mathrm{dom}(\Gamma)$ and $t_i \Vdash_{\Gamma(x_i)} \eta(x_i)$.

Definition 5.2 *Let H be a BCK-algebra. A subalgebra of H is given by a set $H_0 \subseteq H$ which contains B, C, K and is closed under application.*

Proposition 5.3 *Let H_0 be a subalgebra of some BCK-algebra H.*

Suppose that for each operator op of arity $(A_1, \ldots, A_n)A$ we are given a function $t_{op} : H_0^n \longrightarrow H_0$ such that $v_i \Vdash_{A_i} x_i$ for $i = 1 \ldots n$ implies $t_{op}(v_1, \ldots, v_n) \Vdash_A [\![op]\!](x_1, \ldots, x_n)$.

Then for each term $\Gamma \vdash e : A$ there exists an element $t_e \in H_0$ such that whenever $t \Vdash_\Gamma \eta$ then $t_e\, t \Vdash_A [\![e]\!]\eta$.

Example We can take $H = \mathbb{N}$, $ex = e + x$, $B = C = K = 0$, $H_0 = \{0\}$. It then turns out that $e \Vdash_A a$ if and only if $\mathrm{s}_A(a) \leq e$ and Proposition 3.1 becomes a corollary of Proposition 5.3.

5.2 Pairing function and length

Usually, complexity of number-theoretic functions is measured in terms of the binary length $|-|$. This length measure has the disadvantage that there does not exist an injective function $\langle -,- \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ such that $|\langle x,y \rangle| = |x| + |y| + O(1)$ (Thanks to John Longley for a short proof of this fact.) The best we can achieve is a logarithmic overhead:

Lemma 5.4 *There exist injections* $\mathsf{num} : \mathbb{N} \longrightarrow \mathbb{N}$, $\langle -,- \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ *with disjoint images such that* $\mathsf{num}(x)$, $\langle x,y \rangle$ *as well as their inverses are computable in linear time and such that moreover we have*

$$|\langle x,y \rangle| = |x| + |y| + 2||y|| + 3$$
$$|\mathsf{num}(x)| = |x| + 1$$

Recall that $||x|| = |a|$ when $a = |x|$.

Now we define a length measure in such a way that the above pairing function produces constant overhead:

Definition 5.5 *The length function* $\ell(x)$ *is defined recursively by*

$$\ell(\mathsf{num}(x)) = |x| + 1$$
$$\ell(\langle x,y \rangle) = \ell(x) + \ell(y) + 3$$
$$\ell(x) = |x|, \textit{ otherwise}$$

The following estimates are proved by course-of-values induction.

Lemma 5.6 *For every* $x \in \mathbb{N}$*:*

$$|x| \geq \ell(x) \geq |x|/(1 + ||x||)$$

It follows that if a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is computable in time $O(\ell(x)^n)$ then it is all the more computable in time $O(|x|^n)$. Conversely, if $f : \mathbb{N} \longrightarrow \mathbb{N}$ is computable in time $O(|x|^n)$ then the function $\lambda x.f(\mathsf{num}(x))$ is computable in time $O(\ell(x)^n)$.

More generally, in this case f itself is computable in time $O(\ell(x)^{n+1})$ as $|x|/(1 + ||x||) \geq |x|^{1-1/n}$ for large x.

This means that by moving from $|-|$ to ℓ we do not essentially change complexity.

5.3 The *BCK*-algebra

The idea is that an element of the algebra to be constructed is an algorithm together with a polynomial and a size value which together will determine the (maximum) runtime of applications involving it. Unfortunately, using arbitrary length measures rather than ℓ or $|-|$ is delicate as administrative intermediate computations are linear in $|-|$, but may be exponential or worse in some arbitrarily assigned size measure. So the run time bounds will depend both on the pair (size value, polynomial) and on the actual length $\ell(x)$.

An *algorithm* will be formalised as a natural number using Gödelisation of some universal machine model, e.g., Turing machines or LISP expressions.

If e is such an algorithm then by $\{e\}(x)$ we denote the computation of e on input x and also the result of this computation if it terminates. By $\mathsf{Time}(\{e\}(x))$ we denote the runtime of this computation.

By *polynomial* we will henceforth understand a unary polynomial with nonnegative integer coefficients. Let us write $p[d]$ for the coefficient of x^d in p. We define sum and cut-off subtraction of polynomials coefficient-wise by

$$(p_1 + p_2)[d] = p_1[d] + p_2[d]$$
$$(p_1 \dot{-} p_2)[d] = p_1[d] \dot{-} p_2[d]$$

We write $p_1 \leq p_2$ if $p_1[d] \leq p_2[d]$ for all d. Note that if $p_1 \leq p_2$ then $p_2 \dot{-} p_1 + p_1 = p_2$. We assume an encoding of polynomials as integers in such a way that these operations can be performed in time $O(|p_1| + |p_2|)$.

Let $\wp$ be a monotone function such that $|x| \leq \wp(\ell(x))$, e.g., $\wp(u) = cu^{1+\varepsilon}$ for arbitrarily small ε and appropriately chosen constant c.

By monotonicity, we have $\wp(u+v) \geq \max(\wp(u), \wp(v))$, hence $2\wp(u+v) \geq \wp(u) + \wp(v)$ and accordingly $|x_1| + \cdots + |x_n| = O(\wp(\ell(x_1) + \cdots + \ell(x_n)))$

Definition 5.7 *The set* C *contains natural numbers of the form* $x = \langle p_x, \langle l_x, a_x \rangle \rangle$ *where* p_x *is (an encoding of) a polynomial,* l_x *is a natural number thought of as "abstract size", and* a_x *is (an encoding of) an algorithm.*

In addition to this C *contains* 0 *and we define* $p_0 = a_0 = l_0 = 0$.

An application function on C *is defined as follows: given* $e, x \in C$ *then whenever* $y = \{a_e\}(x)$ *is defined then* $ex =_{\mathrm{def}} y$ *provided that* $y \in C$ *and* $p_y \leq p_e + p_x$ *and* $l_y \leq l_e + l_x$ *and* $\ell(y) \leq \pi + \ell(e) + \ell(x)$ *and* $\mathsf{Time}(\{a_e\}(x)) \leq d(\pi + \wp(\ell(e) + \ell(x)))$ *where* $\pi = (p_e + p_x \dot{-} p_y)(l_e + l_x)$ *and* $d = \pi + \ell(e) + \ell(x) - \ell(y)$.

In all other cases $ex =_{\mathrm{def}} 0$.

We call π *the polynomial allowance of the application* $e\ x$ *and* d *its defect.*

Ideally, we would like to allow just time π for application but as said above we also have to allow time for administrative computations which are linear in $|e| + |x|$ hence linear in $\wp(\ell(e) + \ell(x))$. By padding the ℓ-length we can always blow up the defect so as to account for an arbitrary linear factor, see Lemma 5.9 below.

The reason for the use of subtraction in the definition of polynomial allowance and defect has to do with the definability of composition and is explained in more detail in [4]. The verification of composition makes essential use of this.

Proposition 5.8 *There exists a constant γ such that the application ex is computable in time*

$$T = d \cdot (\pi + \wp(\ell(e) + \ell(x) + \gamma))$$

where $\pi = (p_e + p_x \dot{-} p_{ex})(l_e + l_x)$ and $d = \pi + \ell(e) + \ell(x) \dot{-} \ell(ex)$.

Proof. Simulate $\{a_e\}(x)$ for at most $d_0(\pi_0 + \wp(\ell(e) + \ell(x)))$ steps where $\pi_0 = (p_e + p_x)(l_e + l_x)$ and $d_0 = \pi_0 + \ell(e) + \ell(x)$. If the computation has terminated by then check whether the used-up time as well as the result itself fulfill the requirements. If yes then output the result. Otherwise or if the computation has failed to terminate just output 0. The administrative overhead involved with simulation and checking the conditions can be made up for by an appropriate choice of the constant γ. □

Lemma 5.9 *Suppose that e is an algorithm such that whenever $x \in C$ then $\{e\}(x)$ terminates with a result $y \in C$ and, moreover, there exists a polynomial p and an integer l such that whenever $\{e\}(x) = y$ then*

$$\begin{aligned} &l_y \leq l + l_x \\ &p_y \leq p + p_x \\ &\ell(y) = \pi + \ell(x) + O(1) \\ &\mathsf{Time}(\{e\}(x)) = O(\pi + |x|) \end{aligned}$$

where $\pi = (p + p_x \dot{-} p_y)(l + l_x)$ then we can find $e' \in C$ with $p_{e'} = p$ and $l_{e'} = l$ such that $e'x = \{e\}(x)$ for all $x \in C$.

Proof. We let e_1 be an algorithm with the same behaviour as e, but with ℓ-length padded out so as to make up for the O-terms in the assumptions. We can then put $e' = \langle p, \langle l, e_1 \rangle \rangle$. □

Definition 5.10 *The operation $\otimes : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ is defined by*

$$\begin{aligned} p_{x \otimes y} &= p_x + p_y \\ l_{x \otimes y} &= l_x + l_y \\ a_{x \otimes y} &= \langle a_x, a_y \rangle \end{aligned}$$

Proposition 5.11 (Parametrisation) *For every $e \in \mathbb{N}$ there exists $e' \in \mathbb{N}$ with $l_{e'} = l_e, p_{e'} = p_e$ such that for each $x, y \in \mathbb{N}$ we have*

$$e(x \otimes y) = e'xy$$

Proof. We define $p_{e'} = p_e$ and $l_{e'} = l_e$ as required. We define $z = a_{e'}$ in such a way that

$$\begin{aligned} p_{\{z\}(x)} &= p_e + p_x \\ l_{\{z\}(x)} &= l_e + l_x \\ \{a_{\{z\}(x)}\}(y) &= e(x \otimes y) \end{aligned}$$

If this is done reasonably then the time needed to compute $\{z\}(x)$ is linear in $|x|$ and the time needed to compute $\{a_{\{z\}(x)}\}(y)$ equals

$$T(e(x \otimes y)) + O(|x| + |y|)$$

Moreover, we have $\ell(\{a\}_z) \leq \ell(e) + \ell(x) + O(1)$. Here we use the property of the ℓ-length to allow pairing with constant overhead.

Thus, by choosing $\ell(a_{e'})$ sufficiently large we can achieve that

$$e'x = \{a_{e'}\}(x)$$

and also

$$\{a_{\{z\}(x)}\}(y) = a_{\{z\}(x)}(y)$$

and hence the result. □

Theorem 5.12 *There exist constants $B, C, K \in C$ such that the above application function defines a BCK-algebra structure in such a way that $l_B = l_C = l_K = 0$ and $p_B = p_C = p_K = 0$.*

Proof. Let $comp$ be the obvious algorithm which computes $e(fx)$ from $(e \otimes f) \otimes x$. The runtime of $\{comp\}((e \otimes f) \otimes x)$ is $t_{\mathrm{tot}} = t_1 + t_2 + t_a$ where

$$\begin{aligned} t_1 &\leq d_1(\pi_1 + \wp(\ell(f) + \ell(x)) + \gamma) \\ t_2 &\leq d_2(\pi_2 + \wp(\ell(e) + \ell(y)) + \gamma) \\ t_a &= O(\wp(\ell(e) + \ell(f) + \ell(x) + \ell(y) + \ell(z))) \end{aligned}$$

where

$$\begin{aligned} y &= fx \\ z &= ey \\ \pi_1 &= (p_f + p_x \dot{-} p_y)(l_f + l_x) \\ \pi_2 &= (p_e + p_y \dot{-} p_z)(l_e + l_y) \\ d_1 &= \pi_1 + \ell(f) + \ell(x) - \ell(y) \\ d_2 &= \pi_2 + \ell(e) + \ell(y) - \ell(z) \end{aligned}$$

Let

$$\begin{aligned} \pi &= (p_e + p_f + p_x \dot{-} p_z)(l_e + l_f + l_x) \\ d &= \pi + \ell(e) + \ell(f) + \ell(x) - \ell(z) \end{aligned}$$

Now,

$$\pi_1 + \pi_2 \leq \pi$$

by monotonicity and cancellation of p_y. As a consequence we obtain

$$d_1 + d_2 \leq d$$

Therefore,

$$t_1 + t_2 \leq d(\pi + \wp(\ell(f) + \ell(x)) + \wp(\ell(e) + \ell(y)))$$

and we can find a constant c such that

$$t_{\mathrm{tot}} \leq (d + c)(\pi + \wp(\ell(e) + \ell(f) + \ell(x)))$$

This allows us to define $B_0 \in C$ with by $B_0((e \otimes f) \otimes x) = e(fx)$. Namely, we put $l_{B_0} = p_{B_0} = 0$ and obtain a_{B_0} by padding $comp$ so as to make up for the constant c. The combinator B is then obtained by parametrisation.

The other combinators are similar. □

It follows immediately that $C_0 := \{x \in C \mid l_x = 0\}$ forms a subalgebra of C.

We will now describe a realisation of the base types and operators of our linear lambda calculus enabling us to prove the main result.

Booleans are realised by K and O, respectively.

The sole element $\diamond$ of $\Diamond$ is realised by the element $\diamond$ defined by

$$l_\diamond = 1 \wedge p_\diamond = 0 \wedge a_\diamond = 0$$

The relation $\Vdash_{\mathsf{N}}$ is defined inductively by

$$\begin{array}{l} TKK \Vdash_{\mathsf{N}} 0 \\ t \Vdash_{\mathsf{N}} x + 1 \Rightarrow \\ \quad TO(TK(T \diamond t)) \Vdash_{\mathsf{N}} 2(x+1) \\ t \Vdash_{\mathsf{N}} x \Rightarrow \\ \quad TO(TO(T \diamond t)) \Vdash_{\mathsf{N}} 2x + 1 \end{array}$$

The relation $\Vdash_{\mathsf{L}(A)}$ is defined inductively by

$$\begin{array}{l} TKK \Vdash_{\mathsf{L}(A)} [] \\ a \Vdash_A a' \wedge l \Vdash_{\mathsf{L}(A)} l' \Longrightarrow \\ \quad TO(T \diamond (Tal)) \Vdash_{\mathsf{L}(A)} a' :: l' \end{array}$$

The relation $\Vdash_{\mathsf{T}(A)}$ is defined inductively by

$$\begin{array}{l} a \Vdash_A a' \Longrightarrow TKa \Vdash_{\mathsf{T}(A)} \mathsf{leaf}(a') \\ a \Vdash_A a' \wedge l \Vdash_{\mathsf{T}(A)} l' \wedge r \Vdash_{\mathsf{T}(A)} r' \Longrightarrow \\ \quad TO(T \diamond (Ta(Tlr)))) \Vdash_{\mathsf{T}(A)} \mathsf{node}(a, l, r) \end{array}$$

Theorem 5.13 *All the operators described in Section 3 admit a realisation in C_0.*

Proof. The realisation of the constants is direct, for example $\mathsf{cons}_A : \Diamond \multimap A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A)$ may be realised by

$$\lambda c.\lambda a.\lambda l.TO(Tc(Tal))$$

Next, we consider the operator $\mathsf{it}^{\mathsf{T}(A)}$ the other ones being similar.

Suppose we are given $u \in C_0$ and $v \in C_0$ such that $u \Vdash_{A \multimap X} g$ and $v \Vdash_{\Diamond \multimap A \multimap X \multimap X \multimap X} h$ for appropriately typed set-theoretic functions g, h. Our task is to exhibit $w \in C_0$ such that $t \Vdash_{\mathsf{T}(A)} t'$ implies $wt \Vdash_X f(t')$ where $f : [\![\mathsf{T}(A)]\!] \to [\![X]\!]$ is the function defined recursively by

$$\begin{array}{l} f(\mathsf{leaf}(a)) = g(a) \\ f(\mathsf{node}(a, l, r)) = h(\diamond, a, f(l), f(r)) \end{array}$$

If p is a polynomial and $n \in \mathbb{N}$ we write $n.p$ for the polynomial $p + \cdots + p$ with n summands.

Let $t \in C$ be given. We construct a term $B^t \in C$ such that

- $t \Vdash_{\mathsf{T}(A)} t' \Rightarrow tB^t \Vdash_X f(t')$
- $l_{B^t} = 0$
- $p_{B^t} \leq (l_t + 1).p_u + l_t.p_v$
- $\ell(B^t) \leq c((l_t + 1) \cdot \ell(u) + l_t \cdot \ell(v))$ for some constant c.

If $t \not\Vdash_{\mathsf{T}(A)} t'$ for all trees t' then $B^t = 0$. If $t \Vdash_{\mathsf{T}(A)} \mathsf{leaf}(a')$ for some $a' \in [\![A]\!]$ then $B^t = \lambda x.\lambda\bar{a}.u\bar{a}$ (recall that in this case $t = TKa$, so indeed $tB^t = ua \Vdash_X f(t')$. Also $l_{B^t} = 0$ since $u \in C_0$ and $p_{B^t} \leq p_u$ since $p_B = p_C = p_K = 0$.

If $t \Vdash_{\mathsf{T}(A)} \mathsf{node}(a', l', r')$, hence $t = TO(T \diamond (Ta(Tlr)))$ and B^l, B^r have already been defined, then we put

$$\begin{array}{l} B^t = \lambda x\lambda m_1.m_1(\lambda c\lambda m_2.m_2(\lambda\bar{a}\lambda m_3.m_3(\lambda\bar{l}\lambda\bar{r}. \\ \quad vc\bar{a}(\bar{l}B^l)(\bar{r}B^r)))) \end{array}$$

In evaluating tB^t the variables $c, \bar{a}, \bar{l}, \bar{r}$ will be "bound" to a, l, r and the claim $tB^t \Vdash_X f(t')$ follows inductively. The same goes for the other claims.

Finally, we observe that no actual computation takes place in the definition of B^t. It merely consists of arranging an appropriate number of copies of u and v in a pattern prescribed by the structure of t. Therefore, under any reasonable implementation of the combinators B, C, K the term B^t is computable in time $O(|t|)$. This, together with Proposition 5.8 shows that the function sending t to tB^t satisfies the premises of Lemma 5.9 with $l = 0$ and $p(x) = c \cdot (x + 1) \cdot (p_u(x) + \ell(u)) + x \cdot (p_v(x) + \ell(v))$. □

Corollary 5.14 *If* $e : \mathsf{N} \multimap \mathsf{N}$ *is a closed term. Then* $[\![e]\!] \in$ *PTIME.*

Proof. Immediate from Prop. 5.8 using the fact that if $d \Vdash_{\mathsf{N}} x$ then $p_d = 0$. □

6 Conclusion and further work

We have shown that the functions definable in affine linear lambda calculus with a certain iteration principle for inductive datatypes are polynomial time computable. Apart from linearity and the counting of constructor symbols using the $\Diamond$ base type the type system makes no further restrictions and in particular offers full-blown recursion principles for inductive datatypes with arbitrary even higher-order result type.

Of course, rather than introducing $\Diamond$ as a type one could introduce a more complex syntax with a family of judgements $\Gamma \vdash_n e : A$ and function spaces $A \multimap_n B$ which would provide access to n constructor symbols. Even better would be some kind of type inference system which would start from an ordinary functional program and try to annotate it with $\Diamond$-resources so that it would become typable in the present system. That, however, falls beyond the scope of this paper.

The semantic framework used in the proof is certainly not limited to natural numbers, lists, and trees with the indicated operators. New datatypes and operations can be introduced as long as they admit a realisation in C_0. An example is an appropriately typed operator for divide-and-conquer recursion. Another example is a general construct which allows one to turn a function of type $A \multimap \mathsf{B}$ into a function of type $A \multimap \mathsf{B} \otimes A$ which gives its argument back for further processing as was needed in the insertion sort example. It turns out that this is soundly possible iff the type A has the property that $d \Vdash_A x$ implies $p_d = 0$.

Several people have suggested that the type system could also be used to detect space bounded computation and, in a similar vein, to avoid dynamic memory allocation. Preliminary experiments in this direction which interpret $\Diamond$ as a pointer type in the C programming language are promising.

References

[1] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Ramification, Modality, and Linearity in Higher Type Recursion. in preparation, 1998.

[2] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[3] Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions*. PhD thesis, University of Oslo, 1997. Available by ftp from ftp.ifi.uio.no/pub/vuokko/0adm.ps.

[4] Martin Hofmann. Typed lambda calculi for polynomial-time computation, 1998. Habilitation thesis, TU Darmstadt, Germany. To appear as Edinburgh University Technical Report.

[5] Daniel Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, 1993.