Linear types and non-size-increasing polynomial time computation

Martin Hofmann*

Abstract

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 Introduction

Recent work [7, 1, 4] has shown that predicative recursion [2, 9] 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.

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:

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

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

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, 9]. 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 [6] 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

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

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 show that nevertheless all definable first-order functions are polynomial time computable even if they contain higher-order functions as subexpressions.

The crucial innovation is a special resource type $\diamond$ which has no constructors, hence no closed terms. The application of a constructor functions such as successor or list "cons" always requires an additional argument of type $\diamond$, for example, the binary successor function S_1 gets the type $\diamond \multimap \mathsf{N} \multimap \mathsf{N}$ meaning that in order to construct $2n+1$ from n we need an element of type $\diamond$ in our local context. As already mentioned, such an element cannot be generated "out of nothing", the only place where such elements become available is in the body of step functions of recursive definitions. For example, the operator for iteration on notation[1] takes two step functions $h_0, h_1 : \diamond \multimap \mathsf{N} \multimap \mathsf{N}$. That is to say the body of each step function may contain one constructor function, for example S_0 or S_1.

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 [6] based on predicative recursion. In Section 4.6 below we speculate on the expressivity of the present system alone.

A preliminary version of this work has been presented in [5]. Apart from repairing a few minor shortcomings of *loc. cit.* the present paper presents the following new aspects:

i. An operator for duplication of certain data provided it occurs within the guard of a conditional and hence does not contribute to the size (Section 6).

ii. The definition and justification of a similar system which captures polynomial space. Syntactically, the only difference between the system for polynomial time and the one for polynomial space is that the latter has a typing rule for conditionals which allows variables to be shared between the guard and the branches whereas in the former case the guard and the branches must have disjoint sets of free variables.

iii. The definition and justification of an operator for divide-and-conquer recursion.

[1] Iteration on notation defines a function $f : \mathbb{N} \to X$ on integers from a constant $g \in X$ and functions $h_0, h_1 : \mathbb{N} \times X \to X$ by $f(0) = g, f(2(n+1)) = h_0(f(n)), f(2n+1) = h_1(f(n))$.

2 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

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

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 $(A_1, \dots, A_n)A$ 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.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \tag{T-Var}$$

$$\frac{\Gamma, x{:}\, A \vdash e : B}{\Gamma \vdash \lambda x{:}\, A.e : A \multimap B} \tag{T-Arr-I}$$

$$\frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : B} \tag{T-Arr-E}$$

$$\frac{\Gamma_1 \vdash e_1 : A_1 \quad \Gamma_2 \vdash e_2 : A_2}{\Gamma_1, \Gamma_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \tag{T-Tens-I}$$

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

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

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

$$\frac{\begin{array}{c}\mathrm{op} \text{ has arity } (A_1, \dots, A_n)A \\ \emptyset \vdash e_i : A_i \text{ for } i = 1 \dots n\end{array}}{\Gamma \vdash \mathrm{op}(e_1, \dots, e_n) : A} \qquad \text{(T-Op)}$$

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.

2.1 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 $(A_1, \dots, A_n)A$.

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.

2.2 Size function

We want to assign a *partial* size function $\mathbf{s}_A : [\![A]\!] \rightharpoonup \mathbb{N}$ 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

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

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 $\mathbf{s}$.

Notice that a function $f \in [\![A\multimap B]\!]$ has size 0 precisely if it is non-size-increasing, i.e., when $\mathbf{s}_B(f(a)) \leq \mathbf{s}_A(a)$ for all $a \in [\![A]\!]$ (for which $\mathbf{s}_A(a)$ is defined).

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

Proposition 2.1 *Suppose that for each operator op of arity $(A_1,\dots,A_n)A$ and elements $v_i \in [\![A_i]\!]$ with $\mathbf{s}_{A_i}(v_i) = 0$ we have $\mathbf{s}_A([\![op]\!](v_1,\dots,v_n)) = 0$. 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

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

Notice that the premise stipulates in particular that all constants, i.e., operators without arguments, have size zero, so functional constants are required to be non-size-increasing.

We remark that the role of Prop. 2.1 is largely of motivational nature. It will help us to understand the precise formulation of the signature we are going to introduce next. The proof that all first-order functions are polynomial time computable requires a more sophisticated interpretation which we will give in Section 5.

3 Length spaces

In this section we develop a category-theoretic perspective on the above which will not be used as such, but may be instructive for the reader familiar with categories.

Definition 3.1 *The category $\mathbb{L}$ of length spaces has as objects pairs $X = (|X|, \mathbf{s}_X)$ where $|X|$ is a set and $\mathbf{s}_X : |X| \rightharpoonup \mathbb{N}$ is a partial function. A morphism from X to Y is given by a function $f : |X| \rightharpoonup |Y|$ such that whenever $\mathbf{s}_X(x)$ is defined so is $\mathbf{s}_Y(f(x)$ and $\mathbf{s}_Y(f(x)) \leq \mathbf{s}_X(x)$.*

The category $\mathbb{L}$ is symmetric monoidal closed with tensor product given by $|X \otimes Y| = |X| \times |Y|$ and $\mathbf{s}_{X \otimes Y}(x, y) = \mathbf{s}_X(x) + \mathbf{s}_Y(y)$. The corresponding linear function spaces are given by $|X \multimap Y| = |X| \to |Y|$ and $\mathbf{s}_{X \multimap Y}(f) = \min\{c \mid \forall x \in |X|. \mathbf{s}_Y(f(x)) \leq c + \mathbf{s}_X(x)\}$ where again it is understood that definedness of $\mathbf{s}_X(x)$ implies definedness of $\mathbf{s}_Y(f(x))$.

It has cartesian products given by $|X \times Y| = |X| \times |Y|$ and $\mathbf{s}_{X \times Y}(x, y) = \max(\mathbf{s}_X(x), \mathbf{s}_Y(y))$. It also has co-products given by $|X + Y| = |X| + |Y|$ and $\mathbf{s}_{X+Y}(\mathrm{inl}(x)) = \mathbf{s}_X(x)$, $\mathbf{s}_{X+Y}(\mathrm{inr}(y)) = \mathbf{s}_Y(y)$. It has a terminal object I which coincides with the tensor unit and is given by $|I| = \{0\}$ and $\mathbf{s}_I(0) = 0$.

The category $\mathbb{L}$ is also cartesian-closed; the cartesian function spaces are given by $|X \Rightarrow Y| = |X| \to |Y|$ and $\mathbf{s}_{X \Rightarrow Y}(f) = \min\{c \mid \forall x \in |X|. \mathbf{s}_Y(f(x)) \leq \max(c, \mathbf{s}_X(x))\}$ with the same proviso on definedness as in the case of $X \multimap Y$.

Accordingly, $\mathbb{L}$ forms a model of the type theory BI introduced in [11]; apparently it does not form an instance of the classes of models considered there.

Proposition 2.1 can be proved by interpreting the syntax in $\mathbb{L}$ and using the fact that the forgetful functor $\mathbb{L} \xrightarrow[\mathbf{Sets}]{}$ sending X to $|X|$ is structure preserving.

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 2.1 are satisfied.

4.1 Integers

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

$$\begin{aligned} \mathsf{S}_0 &: \Diamond \multimap \mathsf{N} \multimap \mathsf{N} \\ \mathsf{S}_1 &: \Diamond \multimap \mathsf{N} \multimap \mathsf{N} \end{aligned}$$

with interpretation

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

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

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

Induction on the size of the argument 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. 2.1 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 contain variables) then $\mathsf{it}^{\mathsf{N}}(g, h_0, h_1)$ would multiply the size by that constant thus violating 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

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

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:

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

with semantics

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

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

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

such that $[\![\mathrm{add}]\!](x, y, c) = x + y + (c \pmod 2)$. 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 4.6.

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

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

$$\begin{aligned} &f(0) = 1 \\ &f(x) = 4f(\lfloor \tfrac{x}{2} \rfloor) \end{aligned}$$

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

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

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.

4.2 Lists

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 $[\![\mathsf{L}(A)]\!] = [\![A]\!]^\star$ 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 $\mathsf{it}_X^{\mathsf{L}(A)}$ 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.

The existence of the iterator for lists has a category-theoretic interpretation: for length space A let $\mathsf{L}(A)$ be the length space with $|\mathsf{L}(A)| = |A|^*$ and $\mathbf{s}_{\mathsf{L}(A)}(a_1 \dots a_n) = n + \sum_i \mathbf{s}_A(a_i)$. Let $\Diamond$ be the length space $|\Diamond| = \{\diamond\}$ and $\mathbf{s}_\Diamond(\diamond) = 1$.

Now $\mathsf{L}(A)$ is the initial algebra for the functor $T : \mathbb{L} \to \mathbb{L}$ given by $T(X) = I + \Diamond \otimes A \otimes X$.

4.3 Trees

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. $\mathbf{s}_A$) 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.

4.4 Booleans

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

We may use the more suggestove notation $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ for $\mathsf{if}(e_1, e_2, e_3)$ and we have the following *derived* typing rule for this construct

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{B} \qquad \Delta \vdash_\Sigma e_2 : A \qquad \Delta \vdash_\Sigma e_3 : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A} \tag{IF}$$

It asserts that a free variable may be shared among the branches of a case distinction, but not among the branches and the guard. We will later on study more relaxed versions of this rule.

4.5 Examples

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

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

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

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

The extra $\Diamond$-argument is needed because there is always one more leaf than there are nodes. Similarly, we can define a function $\mathrm{nodes} : \mathsf{T}(A) \multimap \mathsf{L}(A)$ 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 $\mathrm{leq} : (A \otimes A) \multimap \mathsf{B} \otimes A \otimes A$ which besides comparing two elements also gives them back for further processing. Below in Section 6 we discuss a general method for obtaining such a comparison function from an ordinarily-typed one. 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

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

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

If $l : \mathsf{L}(A)$ is sorted in the increasing order w.r.t. leq then so is $\mathrm{insert}_A(x, a, l)$ 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 $\mathrm{sort} = \mathsf{it}^{\mathsf{L}(A)}(\mathsf{nil}_A, \mathrm{insert}_A)$.

Similarly, the usual functional implementations of heap sort (involving a binary search tree as an intermediate data structure), breadth first traversal, 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 we discuss below in Section 8

4.6 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

$$\begin{array}{l} g([])(l') = l' \\ g(a :: l)(l') = f(g(l)(l'@[a])) \end{array}$$

where $l@l'$ denotes the concatenation of l and l' and $\mathit{length}(l)$ 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 $[\![g]\!](l,l') = [\![f]\!]^{\mathit{length}(l)}(l'@l)$.

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

Iterating the #-operation and composition allows us to iterate f any polynomial (in $\mathit{length}(l)$) many times provided f does not shorten its argument, i.e., provided $\mathit{length}(f(l)) = \mathit{length}(l)$.

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 having 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 corresponding 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. $\square$

5 Polynomial-time

Our aim is now to prove that whenever $e : \mathsf{N}\multimap\mathsf{N}$ is a closed term then $[\![e]\!]$ will be polynomial time computable. It seems that in general linear space does not suffice to compute $[\![e]\!]$; however, this is the case for the first-order tail recursive fragment used in the proof of Theorem 4.2. See [8] for some more detail.

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

Definition 5.1 *A partial BCK-algebra (BCK-algebra for short) is given by a set H and a partial function app $: H \times H \longrightarrow H$, written as juxtaposition associating to the left, and constants $B, C, K \in H$ such that*

B1 $Bx, Bxy, Cx, Cxy, Kx, Kxy$ are always defined.

B2 $Bxyz = x(yz)$,

B3 $Cxyz = xzy$,

B4 $Kxy = x$.

Here = denotes "Kleene equality", i.e., the left-hand side is defined iff the right hand side is and both are equal.

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

Lemma 5.2 *Let H be a BCK-algebra and t be a term in the language of BCK-algebras and containing constants from H. If the free variable x appears at most once in t then we can find a term $\lambda x.t$ not containing x such that for each $s \in H$ the equation $(\lambda x.t)s = t[s/x]$ is valid in H in the following sense: if a closed substitution instance of either side is defined so is the other and they are equal.*

Proof. By induction on the structure of t. If t does not contain x then we put $\lambda x.t = Kt$. If $t = x$ then $\lambda x.t = I$. If $t = t_1 t_2$ and x does not appear in t_1 then $\lambda x.t = Bt_1(\lambda x.t_2)$. Notice that this is defined by B1. If $t = t_1 t_2$ and x does not appear in t_2 then $\lambda x.t = C(\lambda x.t_1)t_2$. Again, notice that definedness follows from B1. $\square$

For example, if x, y are variables then $\lambda f.fxy = C(CIx)y$. Further abstraction yields the pairing combinator

$$T = \lambda x.\lambda y.\lambda f.fxy$$

It has the property that Tuv is always defined and $Tuvf = fuv$.

In fact, we can equivalently take $T = BC(CI)$. Another important combinator is $O = CK$ satisfying $Ouv = v$.

We will subsequently use untyped affine linear lambda terms to denote elements of particular *BCK*-algebras. The following is handy in calculations:

Lemma 5.3 *Let t be a term containing variable x at most once and $s \in H$ and $y \neq x$. We have*

$$(\lambda x.t)[s/y] = \lambda x.t[s/y]$$

Proof. Induction on the definition of $\lambda x.t$. $\square$

This means that a *beta*-rule is sound for reasoning with meta-notations invovling λx.

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 extended to all types by the following assignments.

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

Whenever we write here or in the sequel $et \Vdash_B \dots$ then this means in particular that the application et is defined.

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

Definition 5.4 *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.*

Definition 5.5 *Let H_0 be a subalgebra of some BCK-algebra H.*

An operator op of arity $(A_1, \dots, A_n)A$ admits a realisation in H_0 *if there exists a function $t_{op} : H_0^n \longrightarrow H_0$ such that $v_i \Vdash_{A_i} x_i$ for $i = 1 \dots n$ implies $t_{op}(v_1, \dots, v_n) \Vdash_A [\![op]\!](x_1, \dots, x_n)$.*

Note that we do not require the function t_{op} to be "tracked" by an element of H. Induction on typing derivations now yields the following soundness result.

Theorem 5.6 *Let H be a BCK-algebra with subalgebra H_0. If every operator admits a realisation in H_0 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 $H = \mathbb{N}$, $ex = e + x$, $B = C = K = 0$, $H_0 = \{0\}$ and realise basic types B by $e \Vdash_B b$ iff $\mathbf{s}_B(b) \leq e$ where $\mathbf{s}_B$ is some size function. It then turns out that $e \Vdash_A a$ iff $\mathbf{s}_A(a) \leq e$ for arbitrary type A and Proposition 2.1 becomes a corollary of Theorem 5.6.

5.2 Category-theoretic viewpoint

Again, we can view the assignment of realisers to terms as an interpretation in an appropriate category.

Definition 5.7 *Let H be a BCK-algebra and $H_0 \subseteq H$ a subalgebra. The category $\mathbb{H}$ of H-sets has as objects pairs $X = (|X|, \Vdash_X)$ where $|X|$ is a set and $\Vdash_X \subseteq H \times |X|$ is a relation. A morphism from X to Y is a function $f : |X| \to |Y|$ such that there exists $e \in H_0$ with $t \Vdash_X x \Rightarrow et \Vdash_Y f(x)$ for all $x \in |X|$ and $t \in H$.*

The category $\mathbb{H}$ is a symmetric monoidal closed category with respect to the tensor product given by $|X \otimes Y| = |X| \times |Y|$ and $\Vdash_{X \otimes Y} = \{(Tuv, (x, y)) \mid u \Vdash_X x \wedge v \Vdash_Y y\}$.

The corresponding linear function space is given by $|X \multimap Y| = |X| \to |Y|$ and $e \Vdash_{X \multimap Y} f \iff \forall x.\forall t.t \Vdash_X a \Rightarrow et \Vdash_Y f(a)$.

The category $\mathbb{H}$ also has cartesian products given by $|X \times Y| = |X| \times |Y|$ and $e \Vdash_{X \times Y} (x, y) \iff eK \Vdash_X x \wedge eO \Vdash_Y y$.

The terminal object coincides with the unit for the tensor product and is given by $|I| = \{0\}, K \Vdash_I 0$.

The category $\mathbb{H}$ also has co-products whose definition is left to the reader as an easy exercise.

Unlike $\mathbb{L}$ the category $\mathbb{H}$ is not in general cartesian-closed, at least not if *PSPACE* $\neq$ *PTIME*. See Section 7 below.

5.3 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 5.8 *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* $a = |x|$.

Proof. Let $F(x)$ be the function which writes out the binary representation of x using 00 for 0 and 01 for 1, i.e., formally, $F(x) = \sum_{i=0}^{n} 4^i c_i$ when $x = \sum_{i=0}^{n} 2^i c_i$.

We now define $\langle x, y \rangle$ as $x \frown y \frown 1 \frown 1 \frown F(|y|) \frown 0$ where $\frown$ is juxtaposition of bit sequences, i.e., $x \frown y = x \cdot 2^{|y|} + y$. We define $\mathsf{num}(x)$ as $x \frown 1 = 2x + 1$.

In order to decode $z = \langle x, y \rangle$ we strip off the least significant bit (which indicates that we have a pair), then continue reading the binary representation until we encounter the first two consecutive ones (1s). What we have read so far is interpreted as the length of y. Reading this far further ahead gives us the value of y. The remaining bits correspond to x. $\square$

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

Definition 5.9 *The length function $\ell(x)$ is defined recursively by*

$$\begin{aligned}&\ell(\mathsf{num}(x)) = |x| + 1\\&\ell(\langle x, y\rangle) = \ell(x) + \ell(y) + 3\\&\ell(x) = |x|, \text{ otherwise}\end{aligned}$$

Lemma 5.10 *For every $x \in \mathbb{N}$:*

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

Proof. By course-of-values induction on x. If x is not of the form $\langle u, v\rangle$ then the result is direct. So assume the latter and that the inequalities have been established for u and v.

Now, $|\langle u, v\rangle| \geq |u| + |v| + 3 \overset{\mathsf{IH}}{\geq} \ell(u) + \ell(v) + 3 = \ell(\langle u, v\rangle)$ so the first inequality holds. For the second one we calculate as follows.

$$\begin{array}{ll}
 & |\langle u, v\rangle| \\
= & |u| + |v| + 2||v|| + 3 \\
\leq & \ell(u)(1 + ||u||) + \ell(v)(1 + ||v||) + 2||v|| + 3 \qquad \text{IH} \\
\leq & (\ell(u) + \ell(v))(1 + ||\langle u, v\rangle||) + 2||\langle u, v\rangle|| + 3 \\
\leq & (\ell(u) + \ell(v) + 3)(1 + ||\langle u, v\rangle||)
\end{array}$$

□

Proposition 5.11 *For each $\varepsilon > 0$ there is a constant c such that the function $\wp(x) = cx^{1+\varepsilon}$ satisfies*

P1 $x \leq y \Rightarrow \wp(x) \leq \wp(y)$ *(monotonicity)*

P2 $a\wp(x) + b\wp(y) \leq \wp(ax + by)$ *(sublinearity)*

P3 $|x| \leq \wp(\ell(x))$

for $x, y \in \mathbb{N}$.

Proof. P1 is obvious. For P2 we first show $x^{1+\varepsilon} + y^{1+\varepsilon} \leq (x + y)^{1+\varepsilon}$ when $x, y \geq 0$. For $x = 0$ we have equality. For $x, y \geq 0$ the derivative of the difference (w.r.t. x) is positive, so we are done. Now P2 follows since obviously $a\wp(x) \leq \wp(ax)$. P3 obviously holds for $|x| = 0$. If $|x| > 0$ we let $\delta = \varepsilon/(1 + \varepsilon)$, hence $1 + \varepsilon = 1/(1 - \delta)$, and choose d such that $1 + ||x|| \leq d|x|^\delta$. Notice that $||x|| = O(\log(|x|))$. This gives $d|x|/(1 + ||x||) \geq |x|^{1-\delta}$ and hence $d^{1+\varepsilon}(|x|/(1 + ||x||))^{1+\varepsilon} \geq |x|$. The claim now follows from P1 and Lemma 5.10 with $c = d^{1+\varepsilon}$. □

We will henceforth assume that a function $\wp$ with properties P1, P2 P3 has been fixed (not necessarily but most conveniently of the form $cx^{1+\varepsilon}$).

We remark that P2, P3 imply

$$a_1|x_1| + \cdots + a_n|x_n| \leq \wp(a_1\ell(x_1) + \cdots + a_n\ell(x_n))$$

5.4 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 $\mathsf{Time}(\{e\}(x))$, resp. $\mathsf{Space}(\{e\}(x)$ we denote the runtime, resp. space consumption, of this computation.

By *polynomial* we will henceforth understand a unary polynomial with non-negative integer coefficients. If p, q are polynomials and $m \in \mathbb{N}$ we write $p \leq_m q$ to mean that $(q-p)(x)$ is positive and monotone for all $x \geq m$. For example, we have $10x \leq_1 0x^2$.

Definition 5.12 *The set C contains natural numbers of the form $x = \langle p_x, \langle \mathsf{num}(l_x), a_x \rangle \rangle$ 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.*

A partial application function on C is defined as follows: given $e, x \in C$ then ex equals the result y of the computation $\{a_e\}(x)$ provided that

C1 $y \in C$

C2 $l_y \leq l_e + l_x$

C3 $p_y \leq_{l_e+l_x} p_e + p_x$

C4 $\ell(y) \leq \pi + \ell(e) + \ell(x)$

C5 $\mathsf{Time}(\{a_e\}(x)) \leq \wp(d(\pi + \ell(e) + \ell(x)))$

where $\pi = (p_e + p_x - p_y)(l_e + l_x)$ (notice that $\pi \geq 0$) and $d = \pi + \ell(e) + \ell(x) - \ell(y)$.

Otherwise, ex is undefined. This includes the case where $\{a_e\}(x)$ itself is undefined. We call π the polynomial allowance of the application $e\ x$ and d its defect.

Ideally, we would like to allow merely 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 5.13 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 [6]. The verification of composition in the proof of Theorem 5.16 below makes essential use of this.

The preliminary version of this paper [5] had a simpler condition C3, namely $p_y \leq p_e + p_x$ coefficientwise. Unfortunately, this condition is not satisfied for the polynomials constructed in the realisation of tree iteration (Theorem 5.17 below). Another difference between the present approach and the one in *loc. cit.* is that runtime and size bounds were actively monitored during the computation of ex which could therefore be assumed total by cutting off in case of time or size overflow.

Lemma 5.13 *Suppose that cand is an algorithm and D is a subset of C such that whenever $x \in D$ then $\{cand\}(x)$ terminates with a result $y \in C$ and, moreover, there exists a polynomial p and integer constants l, c such that*

$$\begin{array}{l} l_y \leq l + l_x \\ p_y \leq_{l+l_x} p + p_x \\ \ell(y) \leq \pi + \ell(x) + c \\ \mathsf{Time}(\{cand\}(x)) \leq \wp((d+c)(\pi + \ell(x) + c)) \end{array}$$

where $\pi = (p + p_x - p_y)(l + l_x)$ and $d = \pi + \ell(x) - \ell(y)$ then we can find $e \in C$ with $p_e = p$ and $l_e = l$ such that $ex = \{cand\}(x)$ for all $x \in D$.

Proof. We may assume without loss of generality that $\ell(\text{cand}) \geq c$. We can then put $e = \langle p, \langle \mathsf{num}(l), \text{cand}\rangle\rangle$. $\square$

Definition 5.14 *The operation $\otimes : C \times C \longrightarrow C$ is defined by*

$$\begin{array}{l} p_{x \otimes y} = p_x + p_y \\ l_{x \otimes y} = l_x + l_y \\ a_{x \otimes y} = \langle a_x, \langle a_y, \langle p_{\text{diff}}, \mathit{num}(l_{\text{diff}})\rangle\rangle\rangle \end{array}$$

where l_{diff} is such that l_x and l_y can be computed from l_{diff} and $l_x + l_y$ in linear time and moreover $\ell(l_x + l_y) + \ell(l_{\text{diff}}) = \ell(l_x) + \ell(y) + \Theta(1)$. The same should hold for "l" replaced with "p".

To achieve the required properties on l_{diff} we can for example put

$$l_{\text{diff}} = \begin{cases} 2l_x \text{if } l_x \leq l_y \\ 2l_x + 1 \text{if } l_y < l_x \end{cases}$$

and apply this procedure coefficientwise to the polynomials.

This guarantees that $\ell(x \otimes y) = \ell(x) + \ell(y) + \Theta(1)$, i.e., the difference of both sides is bounded by a constant.

Proposition 5.15 (Parametrisation) *For every $e \in C$ there exists $e' \in C$ with $l_{e'} = l_e, p_{e'} = p_e$ such that $e'x$ is always defined and if $e(x \otimes y)$ is defined then $e'xy$ is defined and equal.*

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

$$\mathsf{Time}(e(x \otimes y)) + O(|x| + |y|)$$

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. $\square$

If we would drop the size requirement on $p_{\mathrm{diff}}, l_{\mathrm{diff}}$ then we would need to require

$$\ell(e(x \otimes y)) \leq (p_e + p_x + p_y - p_{e(x \otimes y)})(l_e + l_x + l_y) + \ell(e) + \ell(x) + \ell(y)$$

as extra premise to Proposition 5.15. Notice that this requirement is automatically satisfied with our definition of $x \otimes y$.

Theorem 5.16 *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 $(e \otimes f) \otimes x$. Notice that e, f, x can be computed from $(e \otimes f) \otimes x$.

Assume that $e(fx)$ and hence fx are defined. We introduce the following abbreviations:

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

Notice that $\pi_1 + \pi_2 \leq \pi$ and hence $d_1 + d_2 \leq d$. We now have

$$l_{\{comp\}((e\otimes f)\otimes x)} = l_z \leq l_e + l_y \leq l_e + l_f + l_x = l_{(e\otimes f)\otimes x}$$

Next, from $p_y \leq_{l_f+l_x} p_f + p_x$ and $p_z \leq_{l_e+l_y} p_e + p_y$ it follows that $p_y \leq_{l_e+l_f+l_x} p_f + p_x$ and $p_z \leq_{l_e+l_f+l_x} p_e + p_y$, hence $p_z \leq_{l_e+l_f+l_x} p_e + p_f + p_x$.

Then,

$$\begin{aligned}&\ell(\{comp\}((e\otimes f)\otimes x)) \leq \pi_2 + \ell(e) + \ell(y) \leq \pi_2 + \pi_1 + \ell(e) + \ell(f) + \ell(x) \leq \\ &\quad \pi + \ell(e) + \ell(f) + \ell(x) = \pi + \ell((e\otimes f)\otimes x)\end{aligned}$$

The runtime of $\{comp\}((e\otimes f)\otimes x)$ is $t_{\mathrm{tot}} = t_1 + t_2 + t_a$ where

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

Sublinearity and arithmetic gives us

$$\begin{aligned}&t_1 + t_2 \leq \wp(d_1(\pi_1 + \ell(f) + \ell(x)) + (d_2 + \gamma)(\pi_2 + \ell(e) + \ell(y))) \leq \\ &\quad \wp(d_1(\pi_1 + \ell(f) + \ell(x)) + (d_2 + \gamma)(\pi_2 + \ell(e) + \pi_1 + \ell(f) + \ell(x))) \leq \\ &\quad \wp(d(\pi + \ell(e) + \ell(f) + \ell(x)))\end{aligned}$$

Furthermore, sublinearity allows us to find a constant c such that

$$t_a \leq \wp(c(\pi + \ell(e) + \ell(f) + \ell(x)))$$

So that, finally,

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

Hence, Lemma 5.13 with $D = \{(e\otimes f)\otimes x \mid e(fx) \text{ defined}\}$ then gives us $B_0 \in C$ with $B_0((e\otimes f)\otimes x) = e(fx)$.

The desired combinator B is then obtained by parametrisation.

The other combinators are similar. $\square$

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 $\diamondsuit$ is realised by the element $\diamond$ defined by

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

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

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

$$
\begin{array}{l}
\mathsf{tt}, \mathsf{ff} : \mathsf{B} \\
\mathsf{if} : \mathsf{B} \multimap A \times A \multimap A \\
0 : \mathsf{N} \\
\mathsf{S}_0, \mathsf{S}_1 : \Diamond \multimap \mathsf{N} \multimap \mathsf{N} \\
\mathsf{it}^{\mathsf{N}}_A : (A, \Diamond \multimap A \multimap A, \Diamond \multimap A \multimap A)\mathsf{N} \multimap A \\
\mathsf{nil}_A : \mathsf{L}(A) \\
\mathsf{cons}_A : \Diamond \multimap A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A) \\
\mathsf{it}^{\mathsf{L}}_{A,X} : (X, \Diamond \multimap A \multimap X \multimap X)\mathsf{L}(A) \multimap X \mathsf{leaf}_A : A \multimap \mathsf{T}(A) \\
\mathsf{node}_A : \Diamond \multimap A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \\
\mathsf{it}^{\mathsf{T}}_{A,X} : (A \multimap X, \Diamond \multimap A \multimap X \multimap X \multimap X)\mathsf{T}(A) \multimap X
\end{array}
$$

Figure 1: Signature of operators

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

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

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

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

Theorem 5.17 *All the operators described in Section 2 and summarised in Figure 1 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 a realiser for a tree t be given. We construct from t a an element $B^t \in C$ with the following properties

- $t \Vdash_{\mathsf{T}(A)} t' \Rightarrow tB^t \Vdash_X f(t')$
- $l_{B^t} = 0$
- $p_{B^t} \leq_0 (l_t + 1).p_u + l_t.p_v$
- $\ell(B^t) \leq c((l_t + 1) \cdot \ell(u) + l_t \cdot \ell(v))$ for some constant c.

If $t \Vdash_{\mathsf{T}(A)} \mathsf{leaf}(a')$ 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

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

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. To verify the other claims we observe that B_t is of the form

$$B' \underbrace{uuu \cdots u}_{l_t+1} \underbrace{vvv \cdots v}_{l_t}$$

with B' a pure affine lambda term, i.e., definable from B, C, K alone with $l_{B'} = p_{B'} = 0$, and $\ell(B') = O(\ell(t))$.

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, the term B^t is computable in time $O(|t|)$.

Now notice that

$$(l_t + 1).p_u + l_t.p_v \leq_{l_t} (x + 1)p_u + xp_v$$

This suggests to define $p(x) = c \cdot (x + 1) \cdot (p_u(x) + \ell(u)) + x \cdot (p_v(x) + \ell(v))$ as the resource polynomial for the realiser of f. We have

$$p_{B^t} \leq_{l_t} p$$

and moreover

$$p_{B^t}(l_t) + \ell(B^t) \leq p(l_t)$$

This shows that the algorithm cand defined by

$$\{\text{cand}\}(t) = tB^t$$

provides a realiser e for f via Lemma 5.13. $\square$

Corollary 5.18 *If $e : \mathsf{N} \multimap \mathsf{N}$ is a closed term. Then $[\![e]\!] \in PTIME$.*

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

6 Restricted duplication

Definition 6.1 *An element $e \in C$ with $p_e = 0$ is called* data element. *A* datatype *is a type D such that whenever $e \Vdash_D d$ for some $d \in D$ then D is a data element.*

The types $\mathsf{B}, \diamond, \mathbb{N}$ are datatypes; if D_1, D_2 are datatypes so are $\mathsf{L}(D_1), \mathsf{T}(D_1), D_1 \otimes D_2$. In other words, all types not containing the function space $\multimap$.

Definition 6.2 *A type P is* passive *if there exists a constant C such that whenever $e \Vdash_P p$ then $l_e = 0$ and $\ell(e) \le C$.*

For each datatype D and passive type P we introduce a constant

$$\mathsf{dup}_{D,P} : (D \multimap P) \multimap D \multimap (P \otimes D)$$

with semantics

$$[\![\mathsf{dup}_{D,P}]\!](f)(d) = (f(d), d)$$

Theorem 6.3 *For each datatype D and passive type P there exists an element $dup \in C_0$ such that*

$$dup \Vdash_{(D \multimap P) \multimap D \multimap (P \otimes D)} [\![\mathsf{dup}_{D,P}]\!]$$

Proof. Let C be a bound on the ℓ-length of realisers for P. Define $e^P = e$ if $l_e = 0, \ell(e) \le C$ and $e^P = 0$, otherwise. Define $e^D = e$, if $p_e = 0$ and $e^D = 0$, otherwise.

Let cand be the obvious algorithm defined by

$$\{\text{cand}\}(f \otimes x) = T(fx^D)^P x^D$$

we have

$$\begin{aligned}
&l_{\{\text{cand}\}(f\otimes x)} = l_x\\
&p_{\{\text{cand}\}(f\otimes x)} = p_f\\
&\ell(\{\text{cand}\}(f \otimes x)) \le \ell(T) + C + \ell(x) \text{note that } p_T = 0.\\
&\mathsf{Time}(\{\text{cand}\}(f \otimes x)) \le \wp(d(p_f(l_x) + C + 2\ell(x)))
\end{aligned}$$

where $d = p_f(l_x) + 2\ell(x) + O(1) - \ell(x) = p_f(l_x) + \ell(x) + O(1)$.

The claim now follows from Lemma 5.13. $\square$

We can use this result to construct a comparison function as required in the definition of insertion sort in Section 4.5. Namely, if we have defined $\mathrm{leq} : (A \otimes A) \multimap \mathsf{B}$ where A is a datatype then $\mathsf{dup}_{A\otimes A,P}(\mathrm{leq})$ has the required functionality.

Using $\mathrm{dup}_{D,\mathsf{B}}$ we can justify the following more relaxed typing rule for conditionals:

$$\frac{\Gamma, \Delta \vdash_\Sigma e_1 : \mathsf{B} \qquad \Delta, \Theta \vdash e_2 : A \qquad \Delta, \Theta \vdash e_3 : A}{\Gamma, \Delta, \Theta \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A} \qquad (\textsc{If-Dup})$$

with the side-condition that all the types mentioned in the shared context Δ are datatypes. This might be a more convenient way to present the functionality of restricted duplication to the user.

Let us now study the necessity of the restrictions imposed on duplication as provided by dup.

If we would not require that P be passive then we could define a diagonal map $\delta : D \multimap D \otimes D$ with $[\![\delta]\!](d) = (d, d)$ for all datatypes D as

$$\mathsf{dup}_{D,D}(\lambda x\colon D.x)$$

We have already seen in the Introduction that this leads to exponential growth in the case $D = \mathsf{L}(\mathsf{N})$ or merely $D = \diamond$.

If we relax the restriction that D in $\mathsf{dup}_{\mathsf{D},\mathsf{P}}$ be a datatype it is still the case that all definable functions are non-size-increasing. However, we can evaluate linear boolean valued functions more than once allowing us to encode PSPACE complete algorithms. Namely, we wold then be able to define a map

$$\delta : (X \multimap \mathsf{B}) \multimap (X \multimap X \multimap \mathsf{B})$$

which applies a given predicate to two elements and returns the conjunction of the results. Namely, for arbitrary type X let D be $(X \multimap \mathsf{B}) \otimes X$. Then applying $\mathsf{dup}_{D,\mathsf{B}}$ to the evaluation map $D \multimap \mathsf{B}$ gives an element of $D \multimap \mathsf{B} \otimes D$ from which we obtain the desired map δ by currying and projection:

$$\delta(f) = \lambda x_1 \colon X.\lambda x_2 \colon X.\mathsf{let}\ \mathsf{dup}_{D,\mathsf{B}}(\lambda t\colon X \multimap \mathsf{B} \otimes X.\mathsf{let}\ t{=}f \otimes x\ \mathsf{in}\ fx)(f \otimes x_1){=}b \otimes f \otimes x_3\ \mathsf{in}\ \mathsf{if}\ b\ \mathsf{then}\ fx_2\ \mathrm{else}\ \mathsf{ff}$$

Now consider the following pseudocode (taken from [10]) for a function

$$\mathrm{eval} : \mathrm{Formulas} \multimap \mathsf{B}$$

which evaluates a quantified boolean formula (encoded e.g. as an appropriately labelled binary tree).

$$\begin{array}{l} \mathrm{eval}(\varphi \Rightarrow \psi) = \mathsf{if}\ \mathrm{eval}(\varphi)\ \mathrm{then}\ \mathrm{eval}(\psi)\ \mathrm{else\ wahr} \\ \mathrm{eval}(\forall x.\varphi) = \delta(\lambda v\colon \mathsf{B}.\mathrm{eval}(\varphi[v/x]))(\mathsf{tt}, \mathsf{ff}) \end{array}$$

Here $\varphi[v/x]$ denotes substitution of a variable by a boolean constant; an operation readily definable by tree recursion.

Another possible relaxation would consist of allowing duplication of elements of a datatype whenever this does not lead to a size increase in the result. For example, we could be tempted to provide a constant

$$\mathsf{diag}_{D,P} : (D \multimap D \multimap P) \multimap (D \multimap P)$$

with semantics

$$[\![\mathsf{diag}_{D,P}]\!](f)(d) = f(d)(d)$$

This would allow us to apply a predicate to an exponentially large argument. More concretely, let $f : \mathsf{L}(\mathsf{B}) \multimap X \multimap \mathsf{B}$ be a (closed) function with X arbitrary. The following pseudocode

$$F([], x) = \lambda l{:}\,\mathsf{L}(\mathsf{B}).f(l, x)$$
$$F(\mathsf{cons}(\diamond, b, l_1), x) = \lambda l_2{:}\,\mathsf{L}(\mathsf{B}).F(l_1, x)(l_2 @ l_2)$$

could then be translated into a legal definition and, as is readily seen, we have

$$F(l_1, x)(l_2) = f(l_2^{2^{length(l_1)}}, x)$$

where l^n denotes $l@\dots@l$ with n factors.

7 Polynomial space

By changing "time" into "space" in the definition of C, i.e., replacing C5 by

C5$_{\text{space}}$ $\mathsf{Space}(\{a_e\}(x)) \le \wp(d(\pi + \ell(e) + \ell(x)))$

we obtain another BCK-algebra C_{space} which obviously satisfies the same closure properties as C, i.e., it allows for a realisation of our linear lambda calculus including operators associated to lists and trees. Also Lemma 5.13 holds with "time" replaced by "space".

Theorem 7.1 *There exists $e \in C_{\text{space}}$ such that*

$$e \Vdash_{(\mathsf{B} \times X) \multimap (\mathsf{B} \otimes X)} \mathrm{id}$$

Proof. For $x \in C_{\text{space}}$ let x^{B} be x, if $x \in \{K, O\}$ and 0 otherwise, i.e., if $x \Vdash_{\mathsf{B}} x'$ then $x^{\mathsf{B}} = x$ and $l_{x^{\mathsf{B}}} = p_{x^{\mathsf{B}}} = 0$, $\ell(x^{\mathsf{B}}) \le C$ for $C = \max(\ell(K), \ell(O))$.

Let cand be the algorithm defined by

$$\mathrm{cand}(t) = T(tK)^{\mathsf{B}}(tO)$$

We have

$$l_{\{\mathrm{cand}\}(t)} \le l_t$$
$$p_{\{\mathrm{cand}\}(t)} \le_{l_t} p_t$$
$$\ell(\{\mathrm{cand}\}(t)) = p_t(l_t) + \ell(t) + O(1)$$
$$\mathsf{Space}(\{\mathrm{cand}\}(t)) = \max(\mathsf{Space}(tK), \mathsf{Space}(tO)) + O(|t|)$$

Hence, the space version of Lemma 5.13 provides $e \in C_{\text{space}}$ such that $et = \{\mathrm{cand}\}(t)$ and it is clear from the definition of cand that this has the required property. $\square$

It now follows that we can realise a conditional

$$\mathsf{if} : (\mathsf{B} \times X \times X) \multimap X$$

instead of the usual

$$\mathsf{if} : \mathsf{B} \otimes (X \times X) \multimap X$$

Using this, we can justify with C_{space} the typing rule IF with the side condition on Δ removed. I.e.

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{B} \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A}{\Gamma \vdash_\Sigma \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A} \qquad (\text{IF-PSPACE})$$

This rule allows us to define $\mathsf{dup}_{D,\mathsf{B}}$ for arbitrary D and hence to define truth of quantified boolean formulas as described above so that we obtain a characterisation of polynomial space.

In Section 5.2 we raised the question as to whether the category $\mathbb{H}$ of realisability sets is cartesian closed. We can now show that unless P=PSPACE this is not the case when H is C, i.e., the BCK-algebra for polynomial *time*. Suppose the contrary. In any cartesian-closed category the functor $- \times X$ preserves co-products. Since $\mathsf{B} \simeq I + I$ we would have $\mathsf{B} \times X \simeq X + X$, but $X + X$ is isomorphic to $\mathsf{B} \otimes X$ as is readily seen using the ordinary conditional. This means that if H is cartesian closed then truth of quantified boolean formulas can be defined.

We do not know whether the category of C_{space}-sets is cartesian closed. However, it is easy to see that a function spaces $X \Rightarrow D$ when D has the property that $\ell(x) \leq C(l_x + 1)$ for all realisers of D and a fixed constant C. Examples of such objects D are the interpretations of the datatypes as defined above in Section 6.

8 Divide-and-conquer

As already mentioned in Section 5.2 we can justify co-product types. These allow us to formulate an operator for "divide-and-conquer" with the following arity and semantics:

$$\mathsf{dac}_{X,Y,M} : (X \multimap Y + (X \otimes M \otimes X), Y \otimes M \otimes Y \multimap Y, M \multimap \diamond) X \multimap Y$$
$$[\![\mathsf{dac}_{X,Y,M}]\!](s, j, w)(x) = f(x) \text{where,}$$
$$f(x) = \begin{cases} y, \text{ if } s(x) = \mathsf{inl}(y) \\ j(f(l), m, f(r)), \text{ if } s(x) = \mathsf{inr}(l, m, r) \end{cases}$$

Notice that the third argument $w : M \multimap \diamond$ ensures in view of Prop. 2.1 that whenever $\mathbf{s}_M(m) > 0$ whenever it is defined. This guarantees termination of $f(x)$ after at most $\mathbf{s}_X(x)$ unfoldings. Let us now show that, moreover, dac admits a realisation in C.

We assume the following realisation of co-product types: $TKe \Vdash_{X+Y} \mathsf{inl}(x)$ when $e \Vdash_X x$ and $TOe \Vdash_{X+Y} \mathsf{inr}(y)$ if $e \Vdash_Y y$.

Suppose $s, j, w \in C_0$ are realisers for the three arguments to $\mathsf{dac}_{X,Y,M}$. Towards a realisation of $\mathsf{dac}_{X,Y,M}(s, j, w)$ we let cand be the algorithm recursively defined as follows:

$$\{\text{cand}\}(e) = \begin{cases} y, \text{ if } se = TKy \\ j(T\{\text{cand}\}(l)(Tm(\{\text{cand}\}(r)))), \text{ if } se = TO(Tl(Tmr)) \end{cases}$$

Notice if $e \Vdash_X x$ for some x then in the second case we have $l_l + l_r \leq l_e$ because $wm \Vdash_\diamond \diamond$ hence $l_m \geq 1$. This shows that in this case $\{\text{cand}\}(e)$ terminates after l_e unfoldings. We do not care about the (possibly undefined) result in case e is not a realiser.

Let us introduce the following abbreviations:

$$\begin{aligned} &L(e) = l_{\{\text{cand}\}(e)} \\ &P(e) = p_{\{\text{cand}\}(e)} \\ &\mathcal{L}(e) = \ell(\{\text{cand}\}(e)) \\ &\mathsf{Time}(e) = \mathsf{Time}(\{\text{cand}\}(e)) \\ &p(x) = (2x+1)(p_s(x) + \ell(s)) + x(p_j(x) + \ell(j)) \end{aligned}$$

The following estimates can be established by course-of-values induction on l_e.

$$\begin{aligned} &L(e) \leq e \\ &P(e) \leq_{l_e} (2l_e + 1).p_s + l_e.p_j + p_e \\ &\mathcal{L}(e) \leq \pi + \ell(e) + c \\ &\mathsf{Time}(e) \leq \wp(d(\pi + \ell(e))) + c(|e| + 1) \end{aligned}$$

where

$$\begin{aligned} &\pi = (p + p_e - P(e))(l_e) \\ &d = \pi + \ell(e) - L(e) \end{aligned}$$

and c is some constant.

Another way to obtain the first three inequalities consists of noticing that whenever $e \Vdash_X x$ then there is a term B^e built up from B, C, K alone, hence $l_{B^e} = p_{B^e} = 0$ such that

$$\{\text{cand}\}(e) = B^e e \underbrace{sss\ldots s}_{2l_e+1} \underbrace{jjj\ldots j}_{l_e}$$

Namely, B^e lays out the computation tree associated with e. Unfortunately, unlike in the case of tree iteration this term B^e is not computable from e in linear time as its form depends on the outcome of calls to s. However, one can intuitively argue that B^e can be computed "on the fly" according to the outcomes of the calls to s which are made anyway. We do not see a way for making this argument fully precise so that if in doubt the reader must rely on the watertight but lengthy proof by course-of-values induction. Let us write $z = \{cand\}(e)$.

9 Borrowing does not work

One might be tempted to allow for temporary borrowing of the $\Diamond$-resource, for instance in the form of a constant

$$\mathsf{borrow}_X : (\Diamond \multimap \Diamond \otimes X) \multimap X$$

with semantics

$$[\![\mathsf{borrow}_X]\!](f) = x, \text{ when } [\![f]\!](\diamond) = (\diamond, x)$$

In fact under the reading of $\Diamond$ as a certain amount of memory space proposed in [8] such borrowing would correspond to the region-based memory management introduced in [12].

We will show that at least in the presence of restricted duplication (Section 6) such a borrowing functional leads outside the realm of polynomial space.

First, we observe that with restricted duplication and borrowing we can for each datatype D define a functional

$$A : (\Diamond \otimes D \multimap \mathsf{B}) \multimap (D \multimap \mathsf{B})$$

with semantics

$$[\![A]\!](f)(d) = f(\diamond, d)$$

Namely, we put

$$A(f) = \lambda x\colon D.\mathsf{borrow}_{\mathsf{B}}(\lambda d\colon \Diamond.\mathsf{let}\ \mathsf{dup}_{\Diamond\otimes D,\mathsf{B}}(f)(d \otimes x) = b \otimes d' \otimes x\ \mathsf{in}\ d' \otimes b)$$

Using A we can define a functional

$$B : \mathsf{N} \multimap (\mathsf{N} \multimap \mathsf{B}) \multimap \mathsf{B}$$

with semantics

$$[\![B]\!](x)(f) = f([x]^2)$$

namely,

$$\begin{array}{l} B(0)(f) = \lambda f\colon .f(0) \\ B(\mathsf{S}_0(d, n)) = B(\mathsf{S}_1(d, n)) = \lambda f\colon .B(n)(\lambda n'\colon .A(\lambda d' \otimes n''\colon f.(\mathsf{S}_0(d', \mathsf{S}_0(d, n''))))(n')) \end{array}$$

One further iteration gives us a functional

$$C : \mathsf{N} \multimap \mathsf{N} \multimap (\mathsf{N} \multimap \mathsf{B}) \multimap \mathsf{B}$$

with semantics

$$[\![C]\!](x, y, f) = f(2^[x]y)$$

whose presence leads beyond the realm of polynomial space.

We do not know whether this also happens in the absence of restricted duplication. There is, however, no evident way to evaluate programs involving "borrowing" in polynomial space let alone time.

10 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.

We have also shown that by slightly relaxing the typing rule for conditionals we obtain a characterisation of polynomial space.

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. A nontrivial example is a datatype of trees $\mathsf{T}'(A)$ with alternative access which would be given by

$$
\begin{aligned}
&\mathsf{leaf} : A \multimap \mathsf{T}'(A)\\
&\mathsf{node} : \diamond \multimap A \otimes (\mathsf{T}'(A) \times \mathsf{T}'(A)) \multimap \mathsf{T}(A)\\
&\mathsf{it}_X^{\mathsf{T}'(A)} : (A \multimap X, \diamond \multimap A \multimap (X \times X) \multimap X)\mathsf{T}'(A) \multimap X\\
&TKa \Vdash_{\mathsf{T}'(A)} \mathsf{leaf}(a') \iff a \Vdash_A a'\\
&TO(Taf) \Vdash_{\mathsf{T}'(A)} \mathsf{node}(a', l, r) \iff a \Vdash_A a' \wedge fK \Vdash_{\mathsf{T}'(A)} l \wedge fO \Vdash_{\mathsf{T}'(A)} r
\end{aligned}
$$

Notice that if $e \Vdash_{\mathsf{T}'(A)} t$ then l_e is an upper bound on the *depth* of t rather than on its number of nodes.

Accordingly, it is possible to define a function $f : \mathsf{N} \multimap \mathsf{T}'(\mathsf{N})$ which gives the full binary tree of depth $|n|$:

$$
\begin{aligned}
&f(0) = \mathsf{leaf}(0)\\
&f(\mathsf{S}_0(d, n)) = f(\mathsf{S}_1(d, n)) = \mathsf{node}(0, \langle f(n), f(n)\rangle), \text{ i.e.}\\
&f = \mathsf{it}_{\mathsf{T}'(\mathsf{N})}^{\mathsf{N})}(\mathsf{leaf}(0), \lambda d{:}\, \diamond.\lambda t{:}\, \mathsf{T}'(\mathsf{N}).\mathsf{node}(d, \langle t, t\rangle), \dots)
\end{aligned}
$$

Recall that according to T-Prod-I the function $\lambda d{:}\,.\lambda t{:}\,.\mathsf{node}(d, \langle t, t\rangle)$ is linear although t appears twice in its body.

Of course, it is not possible to compute e.g., the list of leaf labellings of an element of $\mathsf{T}'(A)$.

The constructors and the iteration operator admit a realisation in C; the proof of this is, however, rather involved and will be deferred to a future paper.

We have also experimented with a linear version of Kleene's ordinal notation and preliminary evidence suggests that the highest definable ordinal would be ω^2. We leave a detailed investigation of this and related issues as an interesting and exciting topic for future research.

References

[1] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Ramification, Modality, and Linearity in Higher Type Recursion. to appear, 2000.

[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] J.-Y. Girard. Light Linear Logic. *Information and Computation*, 143, 1998.

[5] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS).* IEEE, Computer Society Press, 1999. to appear.

[6] Martin Hofmann. Typed lambda calculi for polynomial-time computation, 1999. Habilitation thesis, TU Darmstadt, Germany. Edinburgh University LFCS Technical Report, ECS-LFCS-99-406.

[7] Martin Hofmann. Safe recursion with higher types and bck-algebra. *Annals of Pure and Applied Logic*, 2000. to appear.

[8] Martin Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP).* Springer, 2000. to appear.

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

[10] Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity II: Substitution and Poly-space. In Jerzy Tiuryn and Leszek Pacholski, editors, *Proc. CSL '94, Kazimierz, Poland, Springer LNCS, Vol. 933*, pages 4486–500, 1995.

[11] Peter O'Hearn. Doubly closed categories , resource interpretations, and the $\alpha\lambda$-calculus. In *Typed Lambda Calculi and Applications (TLCA '99).* Springer LNCS, 1999.

[12] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.