Safe recursion with higher types and *BCK*-algebra

Martin Hofmann*

Abstract

In previous work the author has introduced a lambda calculus SLR with modal and linear types which serves as an extension of Bellantoni-Cook's function algebra BC to higher types. It is a step towards a functional programming language in which all programs run in polynomial time.

In this paper we develop a semantics of SLR using *BCK*-algebras consisting of certain polynomial-time algorithms. It will follow from this semantics that safe recursion with arbitrary result type built up from N and $\multimap$ as well as recursion over trees and other data structures remains within polynomial time.

In its original formulation SLR supported only natural numbers and recursion on notation with first order functional result type.

1 Introduction

In [10] and [11] we have introduced a lambda calculus SLR which generalises the Bellantoni-Cook characterisation of *PTIME* [2] to higher-order functions. The separation between normal and safe variables which is crucial to the Bellantoni-Cook system has been achieved by way of an S_4-modality $\Box$ on types. So $\Box\mathsf{N}$ is the type of normal natural numbers over which primitive recursion is allowed and N is the type of safe natural numbers to which only basic primitive functions may be applied.

While in op. cit. only natural numbers and recursion with first-order functional result type was allowed, we generalise in this paper to recursion with functional result type of arbitrary order and also consider inductive types such as lists and binary trees.

Let us briefly recall Bellantoni-Cook's system BC. Its purpose is to define exactly the *PTIME*-functions on integers using composition and a certain form of primitive recursion. Unlike Cobham's system [5] where every primitive recursive definition must be annotated with an a priori bound on the growth rate of the function to be defined, in BC no explicit mention is made of resource bounds. The restriction to *PTIME* is achieved by separating the variables, i.e. argument positions, into two zones: the normal ones over which primitive recursion is allowed and the safe ones which can only serve as input to basic primitive functions such as case distinction modulo 2. It is customary to note such a function as $f(\vec{x};\vec{y})$ with the normal variables before the semicolon and the safe variables after the semicolon.

The crucial point which prevents us from reverting to ordinary primitive recursion by using normal variables and ignoring the safe variables is that in a primitive recursion a recursive call to the function being defined may only be performed via a safe variable. This ensures in particular that one is not allowed to recur over the result yielded by a recursive call.

It is this restriction which ensures that the time complexity of the definable functions does not explode as is the case with unrestricted primitive recursion. Applying this pattern to the familiar scheme of primitive recursion under which $f(x)$ may be defined in terms of $f(x-1)$ yields the elementary functions. In order to get *PTIME* one must use the following scheme of *recursion on notation* which is a slight variant of the original one[1] used by Bellantoni-Cook:

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

[1] In that scheme one has two recurrence functions h_0 and h_1 employed according to whether the recursion variable is even or odd. This scheme can be defined in terms of the present one and a conditional construct which we define later on.

From $g(\vec{x};\vec{y})$ and $h(\vec{x},x;\vec{y},y)$ define $f(\vec{x},x;\vec{y})$ by

$$f(\vec{x},0;\vec{y}) = g(\vec{x};\vec{y})$$
$$f(\vec{x},x;\vec{y}) = h(\vec{x},x;\vec{y},f(\vec{x},\lfloor\tfrac{x}{2}\rfloor;\vec{y})), \text{ if } x>0$$

In order that safe and normal variables are kept properly distinct the composition scheme is restricted in such a way that a term may be substituted for a normal variable only if it does not depend on safe variables:

From $f(\vec{x};\vec{y})$ and $\vec{u}(\vec{z};)$ and $\vec{v}(\vec{z};\vec{w})$ define $g(\vec{z};\vec{w})$ by

$$g(\vec{z};\vec{w}) = f(\vec{u}(\vec{z};);\vec{v}(\vec{z};\vec{w}))$$

The main result of [2] is that these patterns together with certain simple basic functions, notably constants, the constructors $\mathsf{S}_0(;y) = 2y$ and $\mathsf{S}_1(;y) = 2y+1$, and case distinction define exactly the class of *PTIME* functions.

Before we continue let us look at a few simple examples. We introduce the notations

$$|x| = \lceil \log_2(x+1) \rceil$$
$$[x] = 2^{|x|}$$

for length of x in binary notation and for the least power of 2 exceeding x.

A function of quadratic growth, namely $\mathsf{sq}(x;) = [x]^2$ is defined by

$$\mathsf{sq}(0;) = 1$$
$$\mathsf{sq}(x;) = \mathsf{S}_0(\mathsf{S}_0(\mathsf{sq}(\lfloor\tfrac{x}{2}\rfloor;)))$$

We have $\mathsf{sq}(x;) = 4^{|x|}$ where $|x| = \lceil \log_2(x+1) \rceil$ is the length of x in binary notation. If we attempt to iterate sq to form a function of exponential growth rate like

$$\mathsf{exp}(0;) = 1$$
$$\mathsf{exp}(x;) = \mathsf{sq}(\mathsf{exp}(\lfloor\tfrac{x}{2}\rfloor;))$$

then we violate the stipulation that recursive calls must happen via safe argument positions only. So the definition of exp is ruled out in BC.

In [10] it has been shown that this restricted substitution can be described type-theoretically using a type system derived from intuitionistic S4 modal logic. This type system can then be used to conservatively extend safe recursion with higher-typed functions. The safe recursion operator as well as operators for case distinction can then be given the form of single constants of higher type. Moreover, the presence of higher typed functions facilitates modularization and schematic definitions.

Somewhat more interesting, however, is the extension obtained by allowing safe recursion to define functions with functional *result type*. If this is done without further restrictions then as emerges from the examples below exponentiation becomes definable.

It has been argued in [10, 11] and independently in [1] that an appropriate restriction is to require that the step function describing the passage from $f(\lfloor\frac{x}{2}\rfloor)$ to $f(x)$ must be linear in the sense that the function $f(\lfloor\frac{x}{2}\rfloor)$ is called at most once in the course of the computation of $f(x)$.

If one violates this restriction then exponentially growing functions and hence all elementary time functions become definable.

$$\mathsf{exp}(0) = \lambda y.2y$$
$$\mathsf{exp}(x) = \lambda y.\mathsf{exp}(\lfloor\tfrac{x}{2}\rfloor)(\mathsf{exp}(\lfloor\tfrac{x}{2}\rfloor)(y))$$

Here $\mathsf{exp}(x,y) = 2^{2^{|x|}} \cdot y$, however, the step function $F(u) = \lambda x.u(u(x))$ is not linear in this case and so the type system will reject this definition.

A similar phenomenon occurs if we include binary trees as a basic type. Suppose that we have constructors $\mathsf{leaf}(n)$ and $\mathsf{node}(n,l,r)$ for binary trees labelled with natural numbers. Then the following function constructs a full binary tree of size $|x|$:

$$f(0) = \mathsf{leaf}(0)$$
$$f(x) = \mathsf{node}(0, f(\lfloor \tfrac{x}{2} \rfloor), f(\lfloor \tfrac{x}{2} \rfloor))$$

Again, our type system rules out this definition because the step function, here $\lambda t.\mathsf{node}(0,t,t)$ is not linear.

The above function definition may be considered harmless, if one decrees that the size of a tree is its depth rather than the number of leaves and nodes it contains. However, the following tree recursion brings us back to exponentially growing functions on the integers:

$$g(\mathsf{leaf}(n)) = \lambda y.2y$$
$$g(\mathsf{node}(n,l,r)) = \lambda y.g(l)(g(r)(y))$$

One may argue that, again, g is called twice here; however, this happens on different parts of the recursive argument and indeed ruling this out would make tree recursion essentially useless. In our view the definition of f is the culprit because the corresponding step function $\lambda t.\mathsf{node}(0,t,t)$ doubles a tree. So, in the presence of trees linearity becomes an important restriction already at ground type. [2] This motivates the introduction of a system in which all step functions of recursions are required to be linear. In order to be able to express this formally, we use a typed lambda calculus with two kinds of function spaces: linear function space $A \multimap B$ and modal, nonlinear function space $\Box A \to B$. The precise typing rules associated with modality and linearity are given below; suffice it to say here that the type $\Box A \to B$ is "weaker" than $A \multimap B$; the reason for a function $f = \lambda x\colon A.t$ to be given this weaker type is one of the following:

- x occurs twice in t and these occurrences are not within different branches of a case distinction,
- x is recursed on in t,
- x appears as argument to a function of modal, nonlinear type.

We also have a subtyping mechanism which allows us to consider a function of type $A \multimap B$ as a function of type $\Box A \to B$.

Later on, we will also consider a nonlinear function space which lies between the linear and the modal ones.

An important concept taken from [1] is that of a *safe type*. These are those which are allowed as the result type of a safe recursion. Among the safe types are natural numbers, trees, and linear function spaces of safe types. The precise definition will be given below.

In extension of the type system from [10] we have now type variables ranging over safe types and polymorphic types $\forall X.A$ which allow for simpler typing of constants. Furthermore, we include cartesian products and tensor products.

The basic types are N (integers), $\mathsf{L}(A)$ (lists), and $\mathsf{T}(A)$ (trees) where A must be a safe type.

We have constructors for integers $0 : \mathsf{N}$ and $\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N} \multimap \mathsf{N}$ with intended meaning $\mathsf{S}_0(x) = 2x, \mathsf{S}_1(x) = 2x{+}1$. We have the usual constructors for lists: $\mathsf{nil} : \forall A.\mathsf{L}(A)$ and $\mathsf{cons} : \forall A.A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A)$. We may write $[\,]$ for $\mathsf{nil}[A]$ and $a :: l$ for $\mathsf{cons}[A]\, a\, l$.

We also have constructors for trees $\mathsf{leaf} : \forall X.X \multimap \mathsf{T}(X)$, $\mathsf{node} : \forall X.X \multimap \mathsf{T}(X) \multimap \mathsf{T}(X)$.

[2] Jean-Yves Marion has recently announced a first-order extension of safe recursion with trees in which the first function can be formulated but the second one cannot. This is made possible by using a dag like representation of trees.

The allowed recursion patterns take the form of higher-order constants:

$$\begin{array}{l}
\mathsf{rec}^{\mathsf{N}} : \forall X.X \multimap \Box(\Box\mathsf{N} \to X \multimap X) \to \Box\mathsf{N} \to X \\
\mathsf{rec}^{\mathsf{L}} : \forall A.\forall X.X \multimap \\
\quad \Box(\Box A \to \Box\mathsf{L}(A) \to X \multimap X) \to \Box\mathsf{L}(A) \to X \\
\mathsf{rec}^{\mathsf{T}} : \forall A.\forall X.(\Box A \to X) \to \\
\quad \Box(\Box A \to \Box\mathsf{T}(A) \to \Box\mathsf{T}(A) \to X \multimap X \multimap X) \to \\
\quad \Box\mathsf{T}(A) \to X
\end{array}$$

The restriction that the result type of a recursion must be safe is now implicit in the requirement that type variables always range over safe types.

The intended meaning of the recursors is as follows: $f(x) = \mathsf{rec}^{\mathsf{N}}(X, h, g, x)$ means

$$\begin{array}{l}
f(0) = g \\
f(x) = h(x, f(\lfloor \frac{x}{2} \rfloor)), \text{ if } x > 0
\end{array}$$

Next, $f(l) = \mathsf{rec}^{\mathsf{L}}(A, X, g, h, l)$ means

$$\begin{array}{l}
f(\mathsf{nil}) = g \\
f(\mathsf{cons}(a, l)) = h(a, l, f(l))
\end{array}$$

Finally, $f(t) = \mathsf{rec}^{\mathsf{T}}(A, X, g, h, t)$ means

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

In the system with nonlinear. nonmodal function space we can use the following slightly stronger typing for the recursor $\mathsf{rec}^{\mathsf{N}}$:

$$\mathsf{rec}^{\mathsf{N}} : \forall X.X \multimap (\Box\mathsf{N} \to X \multimap X) \to \Box\mathsf{N} \to X$$

and similar typings for the other recursors.

This reflects the idea that the step function h is used more than once in $\mathsf{rec}^{\mathsf{N}}(g, h)$, but that it does not appear as argument to a recursive function.

Cartesian products and case distinction We include a cartesian product type former $\times$ with the idea that if $e_1 : A_1$ and $e_2 : A_2$ then $\langle e_1, e_2 \rangle : A_1 \times A_2$. A variable occurs linearly in $\langle e_1, e_2 \rangle$ if it occurs linearly in e_1 and e_2. Elements of cartesian products can be decomposed using projection functions $.i : A_1 \times A_2 \multimap A_i$.

The cartesian products allow us to treat case distinction as a single polymorphic constant:

$$\mathsf{case}^{\mathsf{N}} : \forall X.(X \times (\mathsf{N} \multimap X) \times (\mathsf{N} \multimap X)) \multimap \mathsf{N} \multimap X$$

If $f = \mathsf{case}^{\mathsf{N}}(X, g, h_0, h_1)$ then

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

The point about the use of the cartesian product is that a variable occurs linearly in $\mathsf{case}^{\mathsf{N}}(X, g, h_0, h_1)$ if it occurs linearly in each of g, h_0, h_1. Had we used a typing like

$$\mathsf{case}^{\mathsf{N}} : \forall X.X \multimap (\mathsf{N} \multimap X) \multimap (\mathsf{N} \multimap X) \multimap \mathsf{N} \multimap X$$

then for a variable to occur linearly in $\mathsf{case}^{\mathsf{N}}(X, g, h_0, h_1)$ we would have to insist that it occurs in at most one of g, h_0, h_1.

Similarly, we have a case construct for lists:

$$\mathsf{case}^{\mathsf{L}} : \forall A \forall X.X \times (A \multimap \mathsf{L}(A) \multimap X)) \multimap \mathsf{L}(A) \multimap X$$

where $f = \mathsf{case}^{\mathsf{L}}(A, X, h_{\mathsf{nil}}, h_{\mathsf{cons}})$ means

$$\begin{array}{l} f(\mathsf{nil}) = h_{\mathsf{nil}} \\ f(\mathsf{cons}(a, l)) = h_{\mathsf{cons}}(a, l) \end{array}$$

Finally, for trees we have

$$\mathsf{case}^{\mathsf{T}} : \forall A \forall X.((A \multimap X) \times (A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \multimap X)) \multimap \mathsf{T}(A) \multimap X$$

where $f = \mathsf{case}^{\mathsf{T}}(A, X, h_{\mathsf{node}}, h_{\mathsf{leaf}})$ means

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

Notice that these case constructs give functions of linear non-modal type thus cannot be defined in terms of the recursors which always give modal, nonlinear functions. A similar situation occurs in Leivant-Marion's work [15] where the corresponding situation is called *flat recursion*.

Tensor products We also have a tensor product type which differs from the cartesian product in the treatment of linearity. If $e_1 : A_1$ and $e_2 : A_2$ then $e_1 \otimes e_2 : A_1 \otimes A_2$ and a variable occurs linearly in $e_1 \otimes e_2$ if it occurs linearly in either e_1 or e_2 and not in the other. This means that $\lambda x\!:\!A.x \otimes x$ must be given the type $A \to A \otimes A$ or $\Box A \to A \otimes A$, whereas $\lambda x\!:\!A.\langle x, x\rangle$ gets the type $A \multimap A \times A$.

Since the introduction rule for tensor products is more restrictive than the one for cartesian products, accordingly the elimination rule is more liberal. If $e(x, y) : B$ contains two linear variables as indicated of type A_1, A_2 and $e' : A_1 \otimes A_2$ then we may form the term $\mathsf{let}\, e' = x \otimes y \,\mathsf{in}\, e : B$ which binds x, y. Using this, we can define a coercion from tensor product to cartesian product:

$$\lambda z\!:\!A_1 \otimes A_2.\mathsf{let}\, z = x \otimes y \,\mathsf{in}\, \langle x, y\rangle : A_1 \otimes A_2 \multimap A_1 \times A_2$$

and also an evaluation map

$$\lambda z\!:\!(A_1 \multimap A_2) \otimes A_1.\mathsf{let}\, z = f \otimes x \,\mathsf{in}\, f(x) : (A_1 \multimap A_2) \otimes A_1 \multimap A_2$$

On the other hand, the following term does not have type $(A_1 \multimap A_2) \times A_1 \multimap A_2$ since z is not used linearly.

$$\lambda z\!:\!(A_1 \multimap A_2) \times A_1.z.1(z.2) : (A_1 \multimap A_2) \otimes A_1 \multimap A_2$$

Our type system will give the type $\Box((A_1 \multimap A_2) \times A_1) \to A_2$ or $(A_1 \multimap A_2) \times A_1) \to A_2$ to this term according to which nonlinear function spaces we consider.

Duplicable numerals While, as we have seen, functions and trees should in general not be duplicated without recording this in the type numerals can in principle be used more than once. In order to express this fact we can either introduce a constant $d : \mathsf{N} \multimap \mathsf{N} \otimes \mathsf{N}$ or—in the system with nonlinear, nonmodal function space—a type equality $\mathsf{N} \to A = \mathsf{N} \multimap A$.

1.1 Examples

After the negative examples from above let us now see what we *can* do with the given recursion patterns.

Appending two lists. The function $\mathsf{append} : \forall A.\Box\mathsf{L}(A){\to}\mathsf{L}(A){\multimap}\mathsf{L}(A)$ which appends two lists can be defined by

$$\begin{array}{l}\mathsf{append} = \Lambda A.\lambda l_1{:}\,\mathsf{L}(A).\lambda l_2{:}\,\mathsf{L}(A).\\ \quad \mathsf{rec}^{\mathsf{L}(A)}[\mathsf{L}(A)]l_2\\ \quad (\lambda a{:}\,A.\lambda l{:}\,\mathsf{L}(A).\lambda c{:}\,\mathsf{L}(A).\mathsf{cons}[A](a,c))\end{array}$$

The functional $\mathsf{map} : \forall A\forall B.\Box(\Box A{\to}B){\to}\Box\mathsf{L}(A){\to}\mathsf{L}(A)$ which applies a function to all entries in a list can be defined by

$$\begin{array}{l}\mathsf{map} = \Lambda A.\lambda f{:}\,\Box A{\to}B.\\ \quad \mathsf{rec}^{\mathsf{L}}[A][\mathsf{L}(B)]\ \mathsf{nil}[B]\\ \quad (\lambda a{:}\,A.\lambda l{:}\,\mathsf{L}(A).\lambda c{:}\,\mathsf{L}(B).\mathsf{cons}[B](f(a),c))\end{array}$$

In the more refined system with nonlinear, nonmodal function space we have the slightly stronger typing

$$\mathsf{map} : \forall A\forall B.(\Box A{\to}B){\to}\Box\mathsf{L}(A){\to}\mathsf{L}(A)$$

The function $\mathsf{treerev} : \forall A.\Box\mathsf{T}(A){\to}\mathsf{T}(A)$ which hereditarily swaps left and right successors in a tree can be defined as

$$\begin{array}{l}\mathsf{treerev} = \Lambda A.\mathsf{rec}^{\mathsf{T}(A)}[\mathsf{T}(A)]\\ \quad (\lambda a{:}\,A.\mathsf{leaf}[A]a)\\ \quad (\lambda a{:}\,A.\lambda l{:}\,\mathsf{T}(A).\lambda r{:}\,\mathsf{T}(A).\lambda l_rev{:}\,\mathsf{T}(A).\lambda r_rev{:}\,\mathsf{T}(A).\\ \quad\quad \mathsf{node}[A](a,\, r_rev, l_rev))\end{array}$$

Next, we give a sugared version of the familiar insertion sort algorithm. Suppose that we have a function $\leq : \Box A{\to}A{\multimap}\mathsf{N}$ for some type of entries A which is duplicable in the sense that the diagonal $d : A{\multimap}A\otimes A$ is definable. This will e.g. be the case for the type of natural numbers.

We define an insertion function $\mathsf{insert} : \Box\mathsf{L}(A){\to}\Box A{\to}\mathsf{L}(A){\multimap}\mathsf{L}(A)$ such that $\mathsf{insert}(l,a,l')$ inserts a into l' in the correct place assuming that l is longer than l' and that l' is already sorted. The extra parameter l is used to "drive" the recursion enabling us to use l' in a linear way:

$$\begin{array}{l}\mathsf{insert}([\,],a,l') = [\,]\\ \mathsf{insert}(x :: y, a, a' :: l') = \mathsf{if} a \leq a'\\ \quad \mathsf{then}\ a :: a' :: l\\ \quad \mathsf{else}\ a' :: \mathsf{insert}(y,a,l')\end{array}$$

This definition can be formalised using the higher-typed recursion operator $\mathsf{rec}^{\mathsf{L}}[A][\mathsf{L}(A){\multimap}\mathsf{L}(A)]$.

The sorting function of type $\Box\mathsf{L}(A){\to}\mathsf{L}(A)$ is then defined by

$$\begin{array}{l}\mathsf{sort}([\,]) = [\,]\\ \mathsf{sort}(a :: l) = \mathsf{insert}(l, a, \mathsf{sort}(l))\end{array}$$

Here $\mathsf{rec}^{\mathsf{L}(A)}[A][\mathsf{L}(A)]$ has been used.

The correctness of this code hinges on the fact that $\mathsf{sort}(l)$ is not longer than l, hence the particular instance of insert behaves correctly.

The usual recursive definition of insert without extra parameter would yield a function of type $\Box\mathsf{L}(A){\to}\Box A{\to}\mathsf{L}(A)$ which cannot be iterated due to its modal type.

This use of extra parameters to drive recursions is intrinsic to the pattern of safe recursion and already appears in Bellantoni-Cook's proof that all polynomial time functions are definable in their first-order system. The inconvenience caused by this necessity is somewhat palliated by the presence of higher result types as can be seen from the above definition of insert which would be difficult to define with $\mathsf{rec}^{\mathsf{L}}[A][\mathsf{L}(A)]$ alone.

To illustrate the use of tensor products we mention that we can define a function

$$\mathsf{split} : (\Box A{\to}\mathsf{N}){\to}\Box\mathsf{L}(A){\to}\mathsf{L}(A)\otimes\mathsf{L}(A)$$

which splits a list into two parts the first one consisting of those entries which pass a test (given as first argument) and those which don't.

1.2 Expressivity

Bellantoni-Cook's original system can be directly translated into the system with duplicable numerals in the sense that whenever $f(\vec{x};\vec{y})$ is definable in BC with $\vec{x}$ normal and $\vec{y}$ safe then f is definable in SLR with duplicable numerals as a function of type $\Box\mathsf{N}\to\ldots\to\Box\mathsf{N}\to\mathsf{N}\multimap\ldots\multimap\mathsf{N}\multimap\mathsf{N}$ with the $\Box\mathsf{N}$ arguments corresponding to the normal variables.

The expressivity of the system without duplicable numerals remains unexplored.

1.3 Related work

The calculus presented here is closely related to the one developed independently Bellantoni-Niggl-Schwichtenberg [1]. Their system also uses modal and linear types and boasts safe recursion with higher result type. At present, it is based entirely on integers and the only type formers are the two function spaces. Accordingly, it uses the weaker typing for case distinction which does not take into account the fact that only one of the branches is actually evaluated.

The main difference between the two approaches lies, however, in the soundness proof. Whereas op. cit. is based on a syntactical analysis of a normalisation procedure the present proof is based on an interpretation of the calculus in a semantic model.

Another related system is Girard-Asperti's Light Linear Logic. Like [1] this system is a linearly typed lambda calculus admitting a polynomial time normalisation procedure. Although it can be shown that all polynomial time functions are expressible in LLL, the pragmatic aspects, i.e., expressibility of particular algorithms, is unexplored, and superficial evidence suggests that the system would need to be improved in this direction so as to compete with SLR.

A more detailed comparison between the available programming patterns in either system would be very desirable, but must at present await further research. Some preliminary work together with Radha Jagadeesan has shown that at least Bellantoni-Cook's original system [] admits a compositional translation into LLL.

The systems of tiered recursion studied by Leivant and Marion [15, 16, 17] also use restrictions of primitive recursion in order to achieve complexity effects. One difference is that the use of modality is replaced by the use of several copies of the base types ("tiers"). Another difference is that linearity and the ensuing recursion patterns with higher-result type have not been studied in the context of the Leivant-Marion work.

Finally, we mention Caseiro's systems [3]. She studies first-order extensions of safe recursion with inductive datatypes and develops criteria which apply to recursion patterns presently not allowed in SLR like the one used in the direct definition of insertion sort. Unfortunately, these criteria are rather complicated, partly semantical, and do not readily apply to higher-order functions. We hope that by further elaborating the techniques presented in this paper it will be possible to give a type-theoretic account of Caseiro's work which would constitute a further step towards a polynomial time functional programming language. See [12] for a step in this direction.

This article consists of an abridged and slightly revised presentation of the main results of the author's habilitation thesis [13].

1.4 Overview

The next section is devoted to a formal presentation of the calculus to be studied. We also define a set-theoretic interpretation which pins down the intended meaning of the system. We note that no notion of reduction or other kind of operational semantics will be given.

We also give in 2.3 an alternative syntax in which $\Box$ is a type former in its own right and the modal function space can be defined from $\Box$ and $\multimap$.

The rest of the paper is then devoted to showing that the functions of type $\Box\mathsf{N}\to\mathsf{N}$ definable in the system are polynomial time computable. This is done by defining an interpretation of the calculus in which denotations are polynomial time computable by construction. This interpretation is given in several stages. Firstly, in Section 3 we introduce some category-theoretic concepts

needed in the sequel. These are mostly non-standard and should thus not be skipped by the expert.

Then, in Section 4 we define an interpretation of the safe fragment, i.e., linear functions between safe types. These will be interpreted as untyped algorithms (e.g. Turing machines) with runtime bounded by a polynomial of a degree fixed once and for all. In order to interpret recursion later on we must also insist that the size of the result of an application is not larger than the size of the input plus a constant. In order to interpret the linear lambda calculus fragment it then turns out that the size of the result must additionally be traded off against the constant factor of the runtime polynomial.

As an abstract notion of model we use linear combinatory algebra (BCK-algebra) and realisability sets, i.e., we first show that untyped algorithms can be organised into a BCK-algebra H and then give an interpretation of safe types A as H-sets, i.e., a set $|[\![A]\!]|$ together with a relation $\Vdash_{[\![A]\!]} \subseteq H \times |[\![A]\!]|$. A term t of safe type A with sole free variable x of safe type X will be interpreted as a function $[\![t]\!] : |[\![X]\!]| \to |[\![A]\!]|$ which is "computable" by an untyped algorithm $e \in H$ in the sense that whenever $u \Vdash_{[\![X]\!]} x$ then $e \cdot x \Vdash_{[\![A]\!]} [\![t]\!](x)$ where $\cdot$ is application in H.

We have $|[\![\mathsf{N}]\!]| = \mathbb{N}$ and $n' \Vdash_{\mathsf{N}} n \Leftrightarrow n' = \mathrm{num}(n)$ for some simple encoding function $\mathrm{num} : \mathbb{N} \to H$ so that if $t : \mathsf{N} \multimap \mathsf{N}$ then $[\![t]\!]$ is actually polynomial time computable.

We notice that the safe fragment modelled by H consists of pure linear lambda calculus and case distinction only. No recursion operators are involved at that stage.

Next, in Section 6 we provide interpretation for the first-order modal fragment, i.e., for terms t of a type $\Box A \to B$ with A, B safe. These are modelled as functions $[\![t]\!] : |[\![A]\!]| \to |[\![B]\!]|$ such that there exists a polynomial time computable function f with the property that whenever $u \Vdash_{[\![X]\!]} x$ then $f(x) \Vdash_{[\![A]\!]} [\![t]\!](x)$. Recursion patterns can then be identified as operators on these denotations, for instance if $h : |[\![P]\!]| \times |[\![\mathsf{N}]\!]| \to |[\![X]\!] \multimap [\![X]\!]|$ is polynomial time computable in the sense that $p' \Vdash_{[\![P]\!]} p$ and $x' \Vdash_{[\![X]\!]} x$ implies $h'(p', n) \cdot x' \Vdash_{[\![X]\!]} h(p, n, x)$ for some polytime algorithm h' then so is $f : |[\![P]\!]| \times |[\![\mathsf{N}]\!]| \to |[\![X]\!] \multimap [\![X]\!]|$ defined by

$$f(p, 0, x) = x$$
$$f(p, n, x) = h(p, n, f(p, n, \lfloor \tfrac{x}{2} \rfloor)), \text{ otherwise}$$

This uses the fact that whenever $e, x \in H$ then the length of $e \cdot x$ is less than the length of x plus a constant.

In Section 7 we embed the semantics constructed so far into a functor category so as to provide meaning for all types and to obtain recursion patterns as higher-typed constants. This embedding also provides meaning for the nonlinear, nonmodal function space.

In Section 7.2 we formally define the interpretation of SLR in the complete semantics and deduce the main result that first order functions are polynomial time computable.

In Section 8, finally, we show how we can model the extension of the calculus in which natural numbers are duplicable, i.e., in which the two function spaces $\mathsf{N} \multimap A$ and $\mathsf{N} \to A$ are identified and, accordingly, we have a diagonal function $d : \mathsf{N} \multimap \mathsf{N} \otimes \mathsf{N}$. This is done by exhibiting another slightly more complicated BCK-algebra supporting this feature. The rest of the interpretation was set up sufficiently abstractly so that it can be instantiated with this new algebra without further changes.

2 Syntax

We will embark on a formal definition of the calculus under consideration. In view of the semantic nature of our proof of soundness, i.e., that all first-order functions are polynomial time computable, the precise formulation of the syntax is actually not very important as long as it can be interpreted in the semantic model to be given. Nevertheless, we need to fix *some* syntax which we are going to do now.

Aspects. In order to simplify the notation of the various function spaces we use the *aspects* from [10]. An aspect is a pair of the from $a = (m, l)$ where $m \in \{\text{modal}, \text{nonmodal}\}$ and $l \in$

{linear, nonlinear}. Using aspects we can use the generic notation $A \xrightarrow{a} B$ for any of the function spaces by varying a. For example, if $a = (\text{nonmodal}, \text{nonlinear})$ then $A \xrightarrow{a} B$ stands for $\Box A \to B$.

In the system with two function spaces we only use the aspects (nonmodal, nonlinear) (for the linear function space $A \multimap B$) and (modal, nonlinear) (for the modal, nonlinear function space $\Box A \to B$). In the system with nonlinear function spaces we also have the aspect (nonlinear, nonmodal). The fourth possible aspect (modal, linear) is not used in this paper.

Types and subtyping. The types of SLR are given by the following grammar.

A, B	$::=$	X	type variable
	$\mid$	N	integers
	$\mid$	$\mathsf{L}(A)$	lists
	$\mid$	$\mathsf{T}(A)$	trees
	$\mid$	$A \xrightarrow{a} B$	function space
	$\mid$	$\forall X.A$	polymorphic type
	$\mid$	$A \times B$	cartesian product
	$\mid$	$A \otimes B$	tensor product

A type is *safe* if it is built up from variables and N by $\mathsf{L}(-)$, $\mathsf{T}(-)$, $\multimap$ (i.e. $\xrightarrow{a}$ with $a = (\text{nonmodal}, \text{linear})$), $\times$, $\otimes$, $\mathsf{T}(-)$.

We only allow the formation of $\mathsf{L}(A)$ and $\mathsf{T}(A)$ if A is safe.

The aspects are ordered componentwise by "nonlinear" $\leq$ "linear" and "modal" $\leq$ "non-modal".

The subtyping relation between types is the least reflexive, transitive relation closed under the following rules:

$$\frac{A' <: A \qquad B <: B' \qquad a' <: a}{A \xrightarrow{a} B <: A' \xrightarrow{a'} B'}$$

$$\frac{A <: A' \qquad B <: B'}{A \times B <: A' \times B'}$$

$$\frac{A <: A' \qquad B <: B'}{A \otimes B <: A' \otimes B'}$$

$$\frac{A <: B}{\forall X.A <: \forall X.B}$$

Notice the contravariance of the first rule w.r.t. the ordering of aspects so that, e.g., $A \multimap B <: A \to B$ and $A \to B <: \Box A \to B$.

In the system with nonlinear function spaces we also may use the rule

$$\mathsf{N} \to A <: \mathsf{N} \multimap A$$

which states that natural numbers may be duplicated without violating linearity.

Terms. The expressions of SLR are given by the grammar

e	$::=$	x	(variable)
	$\mid$	$(e_1\ e_2)$	(application)
	$\mid$	$\lambda x{:}A.e$	(abstraction)
	$\mid$	$\Lambda X.e$	(type abstraction)
	$\mid$	$e[A]$	(type application)
	$\mid$	$\langle e_1, e_2 \rangle$	(pairing w.r.t. $\times$)
	$\mid$	$e.1 \mid e.2$	(projections)
	$\mid$	$e_1 \otimes e_2$	(pairing w.r.t. $\otimes$)
	$\mid$	$\mathsf{let}\, e_1 = x \otimes y\, \mathsf{in}\, e_2$	($\otimes$-elimination)
	$\mid$	c	(constants)

Here x ranges over a countable set of variables and c ranges over the following set of constants with types as indicated:

$$
\begin{array}{l}
0 : \mathsf{N} \\
\mathsf{S}_0, \mathsf{S}_1 : \mathsf{N} \multimap \mathsf{N} \\
\mathsf{rec}^{\mathsf{N}} : \forall X.X \multimap \Box(\Box\mathsf{N} \to X \multimap X) \to \Box\mathsf{N} \to X \\
\mathsf{case}^{\mathsf{N}} : \forall X.(X \times (\mathsf{N} \multimap X) \times (\mathsf{N} \multimap X)) \multimap \mathsf{N} \multimap X \\
\mathsf{nil} : \forall A.\mathsf{L}(A) \\
\mathsf{cons} : \forall A.A \multimap \mathsf{L}(A) \multimap \mathsf{L}(A) \\
\mathsf{rec}^{\mathsf{L}} : \forall A.\forall X.X \multimap \Box(\Box A \to \Box\mathsf{L}(A) \to X \multimap X) \to \Box\mathsf{L}(A) \to X \\
\mathsf{case}^{\mathsf{L}} : \forall A.\forall X.X \times (A \multimap \mathsf{L}(A) \multimap X) \multimap \mathsf{L}(A) \multimap X \\
\mathsf{leaf} : \forall A.A \multimap \mathsf{T}(A) \\
\mathsf{node} : \forall A.A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \\
\mathsf{rec}^{\mathsf{T}} : \forall A.\forall X.\Box(\Box A \to X) \to \Box(\Box A \to \Box\mathsf{T}(A) \to \Box\mathsf{T}(A) \to X \multimap X \multimap X) \to \Box\mathsf{T}(A) \to X \\
\mathsf{case}^{\mathsf{T}} : \forall A\forall X.((A \multimap X) \times (A \multimap \mathsf{T}(A) \multimap \mathsf{T}(A) \multimap X)) \multimap \mathsf{T}(A) \multimap X
\end{array}
$$

As said before the recursors $\mathsf{rec}^{\mathsf{N}}, \mathsf{rec}^{\mathsf{L}}, \mathsf{rec}^{\mathsf{T}}$ can be given stronger types in the presence of nonlinear function spaces.

Contexts. A *context* is a partial function from variables to pairs of aspects and types. It is typically written as a list of bindings of the form $x \overset{a}{:} A$. If Γ is a context we write $\mathrm{dom}(\Gamma)$ for the set of variables bound in Γ. If $x \overset{a}{:} A \in \Gamma$ then we write $\Gamma(x)$ for A and $\Gamma((x))$ for the aspect a.

A context Γ is nonlinear if all its bindings are of nonlinear aspect.

Two contexts Γ, Δ are disjoint if the sets $\mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta)$ are disjoint. If Γ and Δ are disjoint we write Γ, Δ for the union of Γ and Δ.

2.1 Typing rules

The typing relation $\Gamma \vdash e : A$ between contexts, expressions, and types is defined inductively by the rules in Figure 1. We suppose that all contexts, types, and terms occurring in such a rule are well-formed; in particular, if Γ, Δ or similar appears as a premise or conclusion of a rule then Γ and Δ must be disjoint for the rule to be applicable. The typing rules described here are the affine ones from [10]. The type-checking algorithm from [10] can now be extended to the present calculus yielding a syntax-directed decision procedure for typing.

2.2 Set-theoretic semantics

The calculus SLR has an intended set-theoretic interpretation which in particular associates a function $\mathbb{N} \longrightarrow \mathbb{N}$ to a closed term of type $\Box\mathsf{N} \to \mathsf{N}$. The main result in this paper is that all these functions are computable in polynomial time.

We write $\mathbb{N}$ for the set of natural numbers. If A, B are sets we write $A \times B$ and $A \to B$ for their cartesian product and function space. If A is a set let $\mathsf{L}(A)$ stand for the set of finite lists over A constructed by nil and cons. If A is a set let $\mathsf{T}(A)$ be the set of binary A-labelled trees over A inductively defined by $\mathsf{leaf}(a) \in \mathsf{T}(A)$ when $a \in A$ and $\mathsf{node}(a, l, r) \in \mathsf{T}(A)$ when $a \in A$ and $l, r \in \mathsf{T}(A)$. Let $\mathcal{U}$ be a set containing $\mathbb{N}$ and closed under $\times, \to, \mathsf{T}$.

If η is a partial function from type variables to $\mathcal{U}$ and A is a type then we define a set $[\![A]\!]\eta$ by

$$
\begin{array}{l}
[\![X]\!]\eta = \eta(X) \\
[\![\mathsf{N}]\!]\eta = \mathbb{N} \\
[\![\mathsf{L}(A)]\!]\eta = \mathsf{L}([\![A]\!]\eta) \\
[\![\mathsf{T}(A)]\!]\eta = \mathsf{T}([\![A]\!]\eta) \\
[\![A \overset{a}{\longrightarrow} B]\!]\eta = [\![A]\!]\eta \to [\![B]\!]\eta \\
[\![\forall X.A]\!]\eta = \prod_{B \in \mathcal{U}} [\![A]\!]\eta[X \mapsto B] \\
[\![A \times B]\!]\eta = [\![A \otimes B]\!]\eta = [\![A]\!]\eta \times [\![B]\!]\eta
\end{array}
$$

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

$$\frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B} \qquad \text{(T-Sub)}$$

$$\frac{\Gamma, x \overset{a}{:} A \vdash e : B}{\Gamma \vdash \lambda x{:}A.e : A \overset{a}{\longrightarrow} B} \qquad \text{(T-Arr-I)}$$

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : A \overset{a}{\longrightarrow} B \quad \Gamma, \Delta_2 \vdash e_2 : A \\ \Gamma \text{ nonlinear} \\ x \overset{a'}{:} X \in \Gamma, \Delta_2 \text{ implies } a' \leq a\end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash (e_1\ e_2) : B} \qquad \text{(T-Arr-E)}$$

$$\frac{\Gamma \vdash e : A \quad X \text{ not free in } \Gamma}{\Gamma \vdash \Lambda X.e : \forall X.A} \qquad \text{(T-All-I)}$$

$$\frac{\Gamma \vdash e : \forall X.A \quad B \text{ safe}}{\Gamma \vdash e[B] : A[B/X]} \qquad \text{(T-All-E)}$$

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

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

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \quad \Gamma, \Delta_2 \vdash e_2 : A_2 \quad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \qquad \text{(T-Tens-I)}$$

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : A_1 \otimes A_2 \\ \Gamma, \Delta_2, x \overset{a}{:} A_1, x \overset{a}{:} A_2 \vdash e_2 : B \\ \Gamma \text{ nonlinear}, a = (\text{linear}, \text{nonmodal})\end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1 = x \otimes y\, \mathsf{in}\, e_2 : B} \qquad \text{(T-Tens-E)}$$

$$\frac{c : A}{\Gamma \vdash c : A} \qquad \text{(T-Const)}$$

Figure 1: Typing rules

If A is a closed type then $[\![A]\!]\eta$ is independent of η and we thus write $[\![A]\!]$ in this case. Notice that the interpretation of a safe type always lies in $\mathcal{U}$. Also notice that if $A <: B$ then $[\![A]\!]\eta = [\![B]\!]\eta$.

To each constant $c : A$ we associate an element $[\![c]\!] \in [\![A]\!]$ by the clauses given in the introduction. The interpretation of terms is w.r.t. an partial function η which maps type variables to elements of $\mathcal{U}$ and term variables to arbitrary values:

$$\begin{array}{lcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![\lambda x\colon A.e]\!] & = & \lambda v \in [\![A]\!].[\![e]\!]\eta[x \mapsto v] \\
[\![e_1\ e_2]\!]\eta & = & [\![e_1]\!]\eta([\![e_2]\!]\eta) \\
[\![\Lambda X.e]\!]\eta & = & \lambda A \in \mathcal{U}.[\![e]\!]\eta[X \mapsto A] \\
[\![e[A]]\!]\eta & = & [\![e]\!]\eta([\![A]\!]\eta) \\
[\![\langle e_1, e_2\rangle]\!]\eta & = & ([\![e_1]\!]\eta, [\![e_2]\!]\eta) \\
[\![e.i]\!]\eta & = & v_i \text{ where } [\![e]\!]\eta = (v_1, v_2) \\
[\![e_1 \otimes e_2]\!]\eta & = & ([\![e_1]\!]\eta, [\![e_2]\!]\eta) \\
[\![\mathsf{let}\, e_1 = x \otimes y\, \mathsf{in}\, e_2]\!]\eta & = & [\![e_2]\!]\eta[x \mapsto v_1, y \mapsto v_2] \text{ where } [\![e_2]\!]\eta = (v_1, v_2) \\
[\![c]\!]\eta & = & [\![c]\!]
\end{array}$$

The purpose of this set-theoretic semantics is to specify the meaning of SLR terms. It allows us to do without any notion of term rewriting or evaluation. Of course, by directing the defining equations of the recursors one obtains a normalising rewrite system which computes the set-theoretic meaning of first-order functions. However, there is no reason why such rewrite system should terminate in polynomial time. In order to obtain polynomial time algorithms from SLR-terms one must rather study the soundness proof we give and from it extract a compiler which transforms SLR-programs of first-order type into polynomial time algorithms. That this is possible in principle follows from the fact that our soundness proof is constructive; a practical implementation, however, must await further work.

2.3 Alternative syntax with modal types

It is sometimes convenient to have a unary type former $\Box(-)$ and to define the modal function space from the linear one as

$$\Box A \to B =_{\mathrm{def}} \Box(A) \multimap B$$

Similarly, following Girard, we may introduce a modality $!(-)$ for duplication and define the nonlinear function space as

$$A \to B =_{\mathrm{def}} !(A) \multimap B$$

The reason why we have not done this in the official version is that we were aiming for a system which does not use any new term formers or constants in conjunction with the modalities and can automatically infer the best possible typing of a given term. This is not possible with unary modalities as shown by the following example: The term

$$e =_{\mathrm{def}} \lambda f\colon A \multimap B.\lambda x\colon A.fx$$

can be given any of the following incomparable types

$$\begin{array}{l}
\Box(A \multimap B) \multimap \Box A \multimap \Box B \\
!(A \multimap B) \multimap\ !\, A \multimap\ !\, B \\
(A \multimap\ !\, B) \multimap A \multimap\ !\, B \\
(A \multimap \Box B) \multimap A \multimap \Box B
\end{array}$$

and a few more. So, a flexible yet relatively simple type system like the one for SLR does not seem possible for such a system. It could be that a system with "aspect variables" in the style of Hindley-Milner type variables could lead to a viable solution; again details remain to be studied.

If we are more modest and refrain from all inference then we can have a very simple system with modalities and single linear function space. Such system will fail to have the subject reduction property, but since our semantic soundness proof does not need any notion of reduction on terms this need not concern us.

The types of this system, to be called $\lambda^{\Box !}$ for the moment are given by the grammar

$$A ::= X \mid A_1 \multimap A_2 \mid\ !(A) \mid \Box(A) \mid \forall X.A \mid A_1 \otimes A_2 \mid A_1 \times A_2 \mid \mathsf{N} \mid \mathsf{L}(A) \mid \mathsf{T}(A)$$

The terms are those of SLR and in addition we have new term formers referring to the modalities:

$$e ::= \ldots \mid \Box(e) \mid\ !(e) \mid \mathrm{unbox}(e) \mid \mathrm{derelict}(e)$$

as well as a constant

$$\delta : \mathsf{N} \multimap\ !(\mathsf{N})$$

allowing us to duplicate terms of integer type.

Contexts are sets of bindings $x{:}A$ with disjoint variables like in the simply-typed lambda calculus. There are no aspects and no subtyping.

A context Γ is called *modal* if $\Gamma(x)$ is of the form $\Box(A)$ for every variable $x \in \mathrm{dom}(\Gamma)$. It is called *nonlinear* if $\Gamma(x)$ is of the form $\Box(A)$ or $!(A)$ in this case. So, a modal context is in particular nonlinear.

The constants of this system are the same as the ones for SLR with their types amended according to the above definition.

The typing rules are as follows.

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

$$\frac{\Gamma \vdash e : A \quad \Gamma \text{ modal}}{\Gamma, \Delta \vdash \Box(e) : \Box(A)} \qquad \text{(T-Box-I)}$$

$$\frac{\Gamma \vdash e : \Box(A)}{\Gamma \vdash \mathsf{unbox}(e) : !(A)} \qquad \text{(T-Box-E)}$$

$$\frac{\Gamma \vdash e : !(A)}{\Gamma \vdash \mathsf{derelict}(e) : A} \qquad \text{(T-Bang-E)}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \text{ nonlinear}}{\Gamma \vdash\ !(e) : !(A)} \qquad \text{(T-Bang-I)}$$

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

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B \qquad \Gamma, \Delta_2 \vdash e_2 : A \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash (e_1\ e_2) : B} \qquad \text{(T-Arr-E)}$$

$$\frac{\Gamma \vdash e : A \quad X \text{ not free in } \Gamma}{\Gamma \vdash \Lambda X.e : \forall X.A} \qquad \text{(T-All-I)}$$

$$\frac{\Gamma \vdash e : \forall X.A \quad S \text{ safe}}{\Gamma \vdash e[S] : A[S/X]} \qquad \text{(T-All-E)}$$

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

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

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \quad \Gamma, \Delta_2 \vdash e_2 : A_2 \quad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2} \qquad \text{(T-Tens-I)}$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A_1 \otimes A_2 \qquad \Gamma, \Delta_2, x{:}A_1, x{:}A_2 \vdash e_2 : B \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1{=}x \otimes y\, \mathsf{in}\, e_2 : B} \qquad \text{(T-Tens-E)}$$

$$\frac{\Gamma, \Delta_1 \vdash e_1 : A \qquad \Gamma, \Delta_2, x{:}A \vdash e_2 : B \qquad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1{=}x\, \mathsf{in}\, e_2 : B} \qquad \text{(T-Let)}$$

$$\frac{c : A}{\Gamma \vdash c : A} \qquad \text{(T-Const)}$$

We will show later in Section 7.2 how our semantic soundness proof also covers this system with only superficial amendments.

As said before, the disadvantage of this system is that due to the presence of the extra term formers our programs will be more verbose than those in SLR. For example, the squaring function will be defined as

$$\mathrm{sq} =_{\mathrm{def}} \lambda x{:}\Box(\mathsf{N}).\mathsf{rec}^{\mathsf{N}}[\mathsf{N}]\, 0\ !(\lambda x{:}\Box(\mathsf{N}).\lambda y{:}\mathsf{N}.\mathsf{S}_0(\mathsf{S}_0(y))) : \Box(\mathsf{N}) \multimap \mathsf{N}$$

If we want to apply squaring to a constant, say $2 : \mathsf{N}$ then we have to write

$$\mathrm{sq}(\Box(2)) : \mathsf{N}$$

If we want to apply squaring again, then we have to do this each time:

$$\mathrm{sq}(\Box(\mathrm{sq}(\Box(2))))$$

We notice that typing in this system is not closed under well-typed substitution, i.e., an analogue of rule T-Subst is not admissible. The reason is that rule T-Box-I does not obviously commute with substitutions; an explicit counterexample and a detailed discussion can be found in [19].

3 Category-theoretic background

We assume known basic definitions like category, functor, natural transformations, the category of sets and functions, cartesian-closed categories. See [14] or similar for a reference. The subsequent presentation of the more advanced material will be somewhat terse, but should be accessible to the benevolent mathematically-skilled reader.

3.1 Affine linear categories

We begin by defining the appropriate notion of model for typed linear lambda calculus.

Definition 3.1 *An affine linear category (ALC) is given by the following data:*

- *a category* $\mathbb{C}$,
- *for any two objects* $A, B \in \mathbb{C}$ *an object* $A \otimes B$, *called* tensor product, *and morphisms* $\pi : A \otimes B \longrightarrow A$ *and* $\pi' : A \otimes B \longrightarrow B$, *called projections, which are jointly monomorphic in the following sense. If* $f, g : C \longrightarrow A \otimes B$ *and* $\pi \circ f = \pi \circ g$ *and* $\pi' \circ f = \pi' \circ g$ *then* $f = g$,
- *for any two maps* $f : A \longrightarrow A'$ *and* $g : B \longrightarrow B'$ *a map* $f \otimes g : A \otimes B \longrightarrow A' \otimes B'$ *such that* $\pi \circ (f \otimes g) = f \circ \pi$ *and* $\pi' \circ (f \otimes g) = g \circ \pi'$,
- *an isomorphism* $\alpha : A \otimes (B \otimes C) \longrightarrow (A \otimes B) \otimes C$ *such that* $\pi \circ \pi \circ \alpha = \pi$, $\pi' \circ \pi \circ \alpha = \pi \circ \pi'$, $\pi' \circ \alpha = \pi' \circ \pi'$,
- *an isomorphism* $\gamma : A \otimes B \longrightarrow B \otimes A$ *such that* $\pi \circ \gamma = \pi'$ *and* $\pi' \circ \gamma = \pi$,
- *a terminal object* $\top$ *such that* $\pi : A \otimes \top \longrightarrow A$ *and* $\pi' : \top \otimes A \longrightarrow A$ *are isomorphisms*

This definition warrants some explanation. First, and most importantly, we note that in view of the requirement on the projections the other constructions $f \otimes g$, α, and γ, are uniquely determined by their defining equations.

Next we note that if $\mathbb{C}$ happens to have cartesian products then for every pair of objects A, B we have a monomorphism

$$\langle \pi, \pi' \rangle : A \otimes B \longrightarrow A \times B$$

These leads to the intuition that the tensor product can be seen as an extra property on pairs: given $f : C \longrightarrow A$ and $g : C \longrightarrow B$ then it may or may not be the case that $\langle f, g \rangle$ factors through $A \otimes B$. The definition of ALC states in this sense that certain definable maps involving cartesian products factor through tensor products, for example the associativity map $\langle \langle \pi \circ \pi, \pi' \circ \pi \rangle, \pi' \rangle : (A \times B) \times C \longrightarrow A \times (B \times C)$ does.

What we have essentially used here is the existence of a diagonal map $\delta : \top \longrightarrow \top \otimes \top$ satisfying $\pi \circ \delta = \pi' \circ \delta = \mathrm{id}$. This motivates the following definition.

Definition 3.2 *An object* D *in an ALC is called* duplicable *if there exists a (uniquely determined) morphism* $\delta : D \longrightarrow D \otimes D$ *such that* $\pi \circ \delta = \pi' \circ \delta = \mathrm{id}$.

We write $\mathbb{C}_{\mathrm{dup}}$ for the full subcategory consisting of the duplicable objects.

Lemma 3.3 *If* $\mathbb{C}$ *is an ALC then* $\top \in \mathbb{C}_{\mathrm{dup}}$ *and whenever* $A, B \in \mathbb{C}_{\mathrm{dup}}$ *so is* $A \otimes B$. *Moreover,* $A \otimes B$ *is a cartesian product in* $\mathbb{C}_{\mathrm{dup}}$.

Proof. The diagonal for $\top$ is obtained as a special case of the isomorphism $X \cong X \otimes \top$. The diagonal $\delta^{A \otimes B}$ for $A \otimes B$ is obtained from $\delta^A : A \longrightarrow A \otimes A$ and $\delta^B : B \longrightarrow B \otimes B$ as

$$A \otimes B \longrightarrow (A \otimes A) \otimes (B \otimes B) \longrightarrow (A \otimes B) \otimes (A \otimes B)$$

where the first morphism is $\delta^A \otimes \delta^B$ and the second one is a wiring map. □

The above discussion of global elements generalises to the following lemma.

Lemma 3.4 *If D is duplicable and $f : D \longrightarrow A$, $g : D \longrightarrow B$ then there exists a unique map $h = (f \otimes g) \circ \delta : D \longrightarrow A \otimes B$ such that $\pi \circ h = f$ and $\pi' \circ h = g$.*

For the reader familiar with symmetric monoidal categories (SMC), see [18] we remark that every ALC forms an SMC, but not vice versa.

Definition 3.5 *Let A, B be objects in an ALC $\mathbb{C}$. The linear function space or linear exponential of B by A is given by an object $A \multimap B$, a morphism* $\mathrm{ev} : (A \multimap B) \otimes A \longrightarrow B$*—the evaluation map—and for every morphism $f : X \otimes A \longrightarrow B$ a unique morphism* $\mathrm{curry}(f) : X \longrightarrow A \multimap B$ *called the currying or exponential transpose of f such that* $\mathrm{ev} \circ (\mathrm{curry}(f) \otimes \mathrm{id}_A) = f$.

An ALC in which all linear function spaces exist will be called *affine linear closed category* (ALCC). We remark that the analogue for SMC is called *symmetric monoidal closed category*. In an ALCC the mapping $X \mapsto A \multimap X$ extends to a functor on $\mathbb{C}$ which is right adjoint to the tensor product functor $_ \otimes A$.

3.2 Extensional presheaves

Let $\mathbb{C}$ be a category with terminal object $\top$. We have a functor $\mathcal{G} : \mathbb{C} \longrightarrow \mathbf{Sets}$ defined by $\mathcal{G}(X) = \mathbb{C}(\top, X)$. The elements of $\mathcal{G}(X)$ are called *global elements* of X. A morphism $f : X \to Y$ acts on global elements by composition.

The category $\mathbb{C}$ is well-pointed if the functor $\mathcal{G}$ is faithful, i.e., if two morphisms are equal, if they are equal when applied to arbitrary global elements.

We denote by $\widehat{\mathbb{C}}$ the functor category $\mathbf{Sets}^{\mathbb{C}^{op}}$. An object of $\widehat{\mathbb{C}}$ assigns to each object $X \in \mathbb{C}$ a set F_X and to every morphism $u \in \mathbb{C}(X, Y)$ a function $F_u : F_Y \to F_X$ in such a way that $F_{\mathrm{id}}(x) = x$ and $F_{u \circ v}(x) = F_v(F_u(x))$ for each $x \in F_X$ and $u \in \mathbb{C}(Y, X)$, $v \in \mathbb{C}(Z, Y)$. A morphism $\mu \in \widehat{\mathbb{C}}(F, G)$ assigns a function $\mu_X : F_X \to G_X$ to each $X \in \mathbb{C}$ in such a way that $\mu_Y(F_u(x)) = G_u(\mu_X(x))$ for each $x \in F_X$ and $u \in \mathbb{C}(X, Y)$. The objects of $\widehat{\mathbb{C}}$ are called *presheaves*; the morphisms are called *natural transformations*.

For each object $X \in \mathbb{C}$ we have the representable presheaf $\mathcal{Y}(X) \in \widehat{\mathbb{C}}$ defined by $\mathcal{Y}(X)_Y = \mathbb{C}(Y, X)$ and $\mathcal{Y}(X)_f(g) = g \circ f$. The assignment $\mathcal{Y}$ extends to a functor $\mathcal{Y} : \mathbb{C} \to \widehat{\mathbb{C}}$—the *Yoneda embedding*—by $\mathcal{Y}(f)_Z(g) = f \circ g$ whenever $f \in \mathbb{C}(X, Y)$ and $g \in \mathcal{Y}(X)_Z = \mathbb{C}(Z, X)$. The well-known *Yoneda Lemma* says that this functor is full and faithful; indeed, if $\mu : \mathcal{Y}(X) \to \mathcal{Y}(Y)$ then $f =_{\mathrm{def}} \mu_X(\mathrm{id}_X) \in \mathbb{C}(X, Y)$, and we have $\mu = \mathcal{Y}(f)$. Notice that since $\mu_Z(g) = f \circ g$ we have $\mathcal{G}(f) = \mu_\top$. In view of concreteness of $\mathbb{C}$ we can take this latter equation as the *definition* of f.

The Yoneda Lemma says more generally, that the natural transformations from $\mathcal{Y}(X)$ to some presheaf F are in 1-1 correspondence with the elements of F_X; indeed, if $\mu : \mathcal{Y}(X) \to F$ then $\mu_X(\mathrm{id}_X) \in F_X$ and $\mu_Y(f \in \mathbb{C}(Y, X)) = F_f(\mu_X(\mathrm{id}_X))$ by naturality.

The functor category $\widehat{\mathbb{C}}$ forms an intuitionistic universe of sets, in particular it is cartesian closed. On objects the product and function spaces of two presheaves $F, G \in \widehat{\mathbb{C}}$ are given by $(F \times G)_X = F_X \times G_X$ and $(F \Rightarrow G)_X = \widehat{\mathbb{C}}(\mathcal{Y}(X) \times F, G)$. This means that an element of $(F \Rightarrow G)_X$ assigns to each $Y \in \mathbb{C}$ and each morphism $f \in \mathbb{C}(Y, X)$ a function $F_Y \to G_Y$ in a natural way. In particular, from $u \in (F \Rightarrow G)_X$ we get a function from F_X to G_X by application to the identity morphism. Notice here the similarity to the treatment of implication in Kripke models. A terminal object is given by $\top_X = \{0\}$.

The Yoneda embedding $\mathcal{Y} : \mathbb{C} \longrightarrow \widehat{\mathbb{C}}$ preserves all existing cartesian products and function spaces up to canonical isomorphism.

If $\mathbb{C}$ already has cartesian products then $F \Rightarrow G$ can alternatively be given by the formula

$$(F \Rightarrow G)_X = \widehat{\mathbb{C}}(F, G^X)$$

where $G^X_Y = G_{X \times Y}$. In this case, we also have $\mathcal{Y}(X) \Rightarrow G \cong G^X$.

A presheaf $F \in \widehat{\mathbb{C}}$ is *extensional* if for each object $X \in \mathbb{C}$ and elements $u, v \in F_X$ it is the case that $u = v$ iff $F_x(u) = F_x(v)$ for each $x \in \mathcal{G}(X)$.

If $\mathbb{C}$ is well-pointed then every representable presheaf is extensional; if F, G are extensional so are $F \times G$ and $F \Rightarrow G$. We write $\mathrm{Ext}(\mathbb{C})$ for the full subcategory of $\widehat{\mathbb{C}}$ consisting of the extensional presheaves. If $\mathbb{C}$ is well-pointed then so is $\mathrm{Ext}(\mathbb{C})$. Notice also that in this case we have $\mathcal{Y} : \mathbb{C} \longrightarrow \mathrm{Ext}(\mathbb{C})$ since representable presheaves are extensional. This means that extensional presheaves allow us to faithfully embed a well-pointed category into a cartesian closed well-pointed category.

If $\mathbb{C}$ is a well-pointed ALC then we can define an ALCC structure on $\mathrm{Ext}(\mathbb{C})$ in such a way that the Yoneda embedding preserves tensor products as well as all existing linear function spaces in $\mathbb{C}$. The tensor product $F \otimes G$ of $F, G \in \mathrm{Ext}(\mathbb{C})$ is given on objects by

$$(F \otimes G)_X = \{(f,g) \mid f{\in}F_X \wedge g{\in}G_X \wedge \exists U, V, t, \bar{f}, \bar{g}.t \in \mathbb{C}(X, U \otimes V) \wedge f{=}F_{\pi \circ t}(\bar{f}) \wedge g{=}G_{\pi' \circ t}(\bar{g})\}$$

The linear function space $F \multimap G$ is given by

$$(F \multimap G)_X = \mathrm{Ext}(\mathbb{C})(F, G^{\otimes X})$$

where $G_Y^{\otimes X} = G_{X \otimes Y}$.

Again, it is the case that $\mathcal{Y}(X) \multimap G \cong G^{\otimes X}$.

We remark that the above definition of tensor product and linear function space are special cases of a general construction due to Day [6].

We have a comonad $! : \mathrm{Ext}(\mathbb{C}) \longrightarrow \mathrm{Ext}(\mathbb{C})$ given by

$$(!F)_X = \{f{\in}F_X \mid \exists D{\in}\mathbb{C}_{\mathrm{dup}}, t{\in}\mathbb{C}(X, D), \bar{f}{\in}F_D.f = F_t(d)\}$$

The counit $\mathsf{derelict} : !F \longrightarrow F$ is simply the inclusion $(!F)_X \subseteq F_X$. Clearly, $!!F = !F$ so that when $m : !F \longrightarrow G$ is a morphism then we actually have $!m : !F \longrightarrow !G$.

A presheaf of the form $!F$ is duplicable in $\mathrm{Ext}(\mathbb{C})$ and when $D \in \mathbb{C}$ is duplicable then $!\mathcal{Y}(D) = \mathcal{Y}(D)$. So, $!$ provides a canonical way of turning a presheaf into a duplicable one.

It is instructive to see what $!$ does to representable presheaves. Namely, $!\mathcal{Y}(X)_Y$ consists of all those $\mathbb{C}$-morphisms $f : Y \to X$ which can be factored through a duplicable object, i.e., can be written as $f = u \circ v$ where $u : D \to X, v : Y \to D$ for some $D \in \mathbb{C}_{\mathrm{dup}}$. Clearly, in this case, f can be duplicated, i.e., we can find $f' : Y \longrightarrow X \otimes X$ such that $\pi \circ f' = \pi' \circ f' = f$, namely, $f' = (u \otimes u) \circ \delta_D \circ v$.

Finally, we notice that $!(F \otimes G) = !F \otimes !G$ for all presheaves $F, G \in \mathrm{Ext}(\mathbb{C})$ and that $\mathcal{G}(!F) = \mathcal{G}(F)$ since $\top$ is always duplicable.

4 *BCK*-algebra

Definition 4.1 *A BCK-algebra is given by a set A, a binary operation (written as juxtaposition) associating to the left and three constants $B, C, K \in A$ such that the following equations are valid.*

$$\begin{aligned} Kxy &= x \\ Bxyz &= x(yz) \\ Cxyz &= xzy \end{aligned}$$

An identity combinator I with $Ix = x$ can be defined as $I = CKK$[3].

Lemma 4.2 *Let A be a BCK-algebra and t be a term in the language of BCK-algebras and containing constants from A. If free variable x appears at most once in t then 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 A, i.e., all ground instances of the equation hold in A.*

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 $\lambda x.t = Bt_1(\lambda x.t_2)$. If $t = t_1 t_2$ and x does not appear in t_2 then $\lambda x.t = C(\lambda x.t_1)t_2$. □

[3] Thanks to Andrzej Filinski for pointing this out to me.

The reason why *BCK*-algebras are interesting in the context of polynomial time computation is that all functions which are computable in time $O(|x|^p)$ for a fixed p and which are bounded by a linear function with unit slope can be organised into a *BCK*-algebra as we will now show.

4.1 Pairing function and length

In this section we describe a pairing function and a size measure on integers with respect to which the size of a pair exceeds the sum of the sizes of its components by a constant only. This will greatly simplify the subsequent calculations.

Usually complexity of number-theoretic functions is measured in terms of the length in binary given explicitly by $|x| = \lceil \log_2(x+1) \rceil$. 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 $|\langle x, y \rangle| = |x| + |y| + O(1)$[4]. The best we can achieve is a logarithmic overhead:

Lemma 4.3 *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 the functions* .1 *and* .2, getnum, isnum, *and* ispair *defined by*

$$\begin{array}{l} \langle x, y \rangle.1 = x \\ \langle x, y \rangle.2 = y \\ z.1 = z.2 = 0, \textit{ otherwise} \\ \mathsf{ispair}(\langle x, y \rangle) = 1 \\ \mathsf{ispair}(z) = 0, \textit{ otherwise} \\ \mathsf{getnum}(\mathsf{num}(x)) = x \\ \mathsf{getnum}(z) = 0, \textit{ otherwise} \\ \mathsf{isnum}(\mathsf{num}(x)) = 1 \\ \mathsf{isnum}(z) = 0 \end{array}$$

are computable in linear time and such that moreover we have

$$\begin{array}{l} |\langle x, y \rangle| \leq |x| + |y| + 2||y|| + 3 \\ |\mathsf{num}(x)| \leq |x| + 1 \end{array}$$

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

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. What we've 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. □

Now we redefine our length measure so that the above pairing function produces constant overhead:

Definition 4.4 *The length function* $\ell(x)$ *is defined recursively by*

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

Note that $\ell(0) = 0$.

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

Lemma 4.5 *The following inequalities hold for every* $x \in \mathbb{N}$.

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

[4] Thanks to John Longley for a short proof of this fact.

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.

4.2 Polynomial time algorithms as a *BCK*-algebra

Preliminaries. Assume some reasonable coding of Turing machines and configurations as natural numbers using the above pairing function. For Turing machine e and input $\vec{x}$ we let $\mathsf{init}(e, \vec{x})$ denote the initial configuration of Turing machine e applied to input $\vec{x}$. For configuration $c \in \mathbb{N}$ (which includes the contents of the tapes as well as the machine itself) we let $\mathsf{step}(c)$ denote the configuration obtained from c by performing a single computation step. We let $\mathsf{term}(c) = 0$ if c is a terminated configuration and $\mathsf{term}(c) = 1$ otherwise. We may assume that $\mathsf{term}(c) = 0$ implies $\mathsf{step}(c) = c$. Finally, we let $\mathsf{out}(c)$ be the output contained in a terminated configuration c. We may assume that $\mathsf{term}(c) = 1$ implies $\mathsf{out}(c) = 0$. It is intuitively clear that these basic functions are computable in linear time as they only involve case distinctions and simple manipulations of bitstrings; see [4] for a formal proof.

For the rest of Section let $p > 2$ be a fixed integer.

Definition 4.6 *A computation $\{e\}(x)$ is called* short *(w.r.t. p) if it terminates in not more than $d(\ell(e) + \ell(x))^p$ steps where $d = \ell(e) + \ell(x) - \ell(\{e\}(x))$.*

An algorithm e is called short if $\{e\}(x)$ is short for all x.

The difference d between $\ell(e) + \ell(x)$ and $\ell(\{e\}(x))$ is called the *defect* of computation $\{e\}(x)$. Notice that if $\{e\}(x)$ is short then it must have nonzero defect so $\ell(\{e\}(x)) < \ell(e) + \ell(x)$ for every x. Also notice that if f is computable in time $O(\ell(x)^p)$ and $\ell(f(x)) = \ell(x) + O(1)$ then by padding (inserting comments) we can obtain a short algorithm e for f.

Proposition 4.7 *There exists a function $\mathsf{app} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ and a constant γ such that*

- *$\mathsf{app}(e, x)$ is computable in time $d(\ell(e) + \ell(x) + \gamma)^p$ where $d = \ell(e) + \ell(x) - \ell(\mathsf{app}(e, x))$*
- *If $\{e\}(x)$ is short then $\{e\}(x) = \mathrm{app}(e, x)$.*

Proof. Given e, x we simulate $\{e\}(x)$ for at most $(\ell(e) + \ell(x))^{p+1}$ steps. If the computation has halted then we can compute the defect $d = \ell(e) + \ell(x) - \ell(\{e\}(x))$. We now check whether the actual runtime was smaller than $d(\ell(e) + \ell(x))^p$. If no or if the computation hasn't halted in the first place we discard the result and set $\mathsf{app}(e, x) = 0$, otherwise we forward the result to the output, i.e., $\mathsf{app}(e, x) = \{e\}(x)$.

The number of simulation steps performed therefore equals $d(\ell(e) + \ell(x))^p$ where $d = \ell(e) + \ell(x) - \ell(\mathsf{app}(e, x))$ in any case.

If the computation $\{e\}(x)$ is short then the dedicated time suffices to finish it so that $\{e\}(x) = \mathsf{app}(e, x)$ in this case. The total running time of $\mathsf{app}(e, x)$ consists of the simulation steps plus a certain number of steps needed for initialisation, some arithmetic, and moving around intermediate results. The number of these steps is linear in the binary length of the input, thus quadratic in $\ell(e) + \ell(x)$ by Lemma 4.5 and thus can be accounted for by an appropriate choice of the constant γ in view of $p > 2$. □

We will henceforth write $\mathsf{app}(e,x)$ as ex where appropriate.

Before embarking on the proof that the abovedefined application function induces a *BCK*-algebra structure on the natural numbers we will try to motivate the notion of short computation and in particular the role of the defect.

The starting point is that we want to construct an untyped universe of such computations which can later on serve as step functions in safe recursions. Certainly, these algorithms should themselves be polynomial time computable. Moreover, in order that their use as step functions does not lead beyond polynomial time we must require a growth restriction of the form $\ell(f(x)) = \ell(x) + O(1)$. Remember that if, e.g., $\ell(f(x)) = 2\ell(x)$ then $\ell(f^{|y|}(x)) = 2^{|y|}\ell(x)$, thus we quit polynomial time.

Next, in order that application itself be polynomial time computable we must restrict to algorithms running in time $O(\ell(x)^p)$ for some fixed p.

Next, we have to look at the coefficient of the leading term of the polynomial governing the runtime. If our algorithms have running time $d\ell(x)^p + O(\ell(x)^{p-1})$ for arbitrary d then the application runs in time $O(\ell(x)^{p+1})$ thus, again, application is not among the algorithms considered and accordingly, no higher-order functions are possible. If we also bound the coefficient of the leading term and only consider algorithms running in $d\ell(x)^p + O(\ell(x)^{p-1})$ for fixed d and p then once again we lose closure under composition, as in order to evaluate $f(g(x))$ we must evaluate both f and g requiring time $2d\ell(x)^p + O(\ell(x)^{p-1})$. The solution is to couple runtime and output size via the defect so that if $u := g(x)$ is large (which would mean that the second computation $f(u)$ runs longer) then this is made up for by a shorter runtime of $g(x)$. We shall now see formally that this works and in particular that a B-combinator is definable.

Lemma 4.8 (Parametrisation) *For every e there exists an algorithm e' such that $e\langle x,y\rangle = e'xy$.*

Proof. Let e_0 be the following algorithm which on input x outputs the program which on input y outputs $\mathsf{app}(e,\langle x,y\rangle)$.

Now, assuming that e_0 has been reasonably encoded, we have $\ell(\{e_0\}(x)) \leq \ell(e) + \ell(x) + c_0$ where c_0 is some fixed constant. Note that we use here the fact that $\ell(\langle u,v\rangle) = \ell(u) + \ell(v) + 3$ so it is possible to "hardwire" both e and x without sacrificing essentially more than $\ell(e) + \ell(x)$ in length. By padding e_0 we obtain an algorithm e_1 with the same behaviour as e_0 and such that $\ell(\{e_1\}(x)) < \ell(e_1) + \ell(x)$. Since $\{e\}(x)$ terminates in time linear in $|x|$, thus quadratic in $\ell(x)$, we can—by further padding e_1—obtain an algorithm e_2 such that $e_2x = \{e_0\}(x)$.

Now, by construction, we have $\{e_2x\}(y) = e\langle x,y\rangle$ and the computation $\{e_2x\}(y)$ takes less than $d(\ell(e) + \ell(x) + \ell(y) + 3 + \gamma)^p$ steps where γ is the constant from Prop. 4.7 and $d = \ell(e) + \ell(x) + \ell(y) + 3 - \ell(e\langle x,y\rangle)$. Therefore, by further padding e_2 to make up for $\gamma + 3$ we obtain the desired algorithm e'. □

Theorem 4.9 *The set of natural numbers together with the above application function* app *is a BCK-algebra.*

Proof. The combinator K is obtained by parametrising the (linear time computable) left inverse to the pairing function.

For the composition combinator B we start with the following algorithm B_0 defined by

$$\{B_0\}(w) = \mathsf{app}(w.1.1, \mathsf{app}(w.1.2, w.2))$$

We have $\{B_0\}(\langle\langle x,y\rangle,z\rangle) = x(yz)$ and the time t_{tot} needed to evaluate $\{B_0\}(\langle\langle x,y\rangle,z\rangle)$ is less

than $t_1 + t_2 + t_b$ where

$$\begin{aligned} u &= yz \\ w &= x(yz) \\ d_1 &= \ell(y) + \ell(z) - \ell(u) \\ d_2 &= \ell(x) + \ell(u) - \ell(w) \\ t_1 &= d_1(\ell(y) + \ell(z) + \gamma)^p \\ t_2 &= d_2(\ell(x) + \ell(u) + \gamma)^p \end{aligned}$$

and where t_b—the time needed for shuffling around intermediate results—is linear in $|x| + |y| + |z| + |u| + |w|$ thus $O((\ell(x) + \ell(y) + \ell(z))^2)$ where we have used the inequalities $\ell(u) \leq \ell(y) + \ell(z)$ and $\ell(w) \leq \ell(x) + \ell(y) + \ell(z)$ to get rid of the u and w.

This means that we can find a constant c_2 such that

$$t_{\mathsf{tot}} \leq (d_1 + d_2)(\ell(x) + \ell(y) + \ell(z) + c_2)^p$$

Here we have used the fact that $\ell(u) \leq \ell(y) + \ell(z)$.

Now the defect of the computation $\{B_0\}(\langle\langle x, y\rangle, z\rangle)$ equals $\ell(B_0)+\ell(x)+\ell(y)+\ell(z)+6-\ell(w) = \ell(B_0) + 6 + d_1 + d_2$. Therefore, by choosing $\ell(B_0)$ large enough we obtain

$$\mathsf{app}(B_0, \langle\langle x, y\rangle, z\rangle) = \{B_0\}(\langle\langle x, y\rangle, z\rangle) = x(yz)$$

The desired algorithm B is then obtained by applying Lemma 4.8 twice.

Notice, that the existence of the B combinator hinges on the fact that the time of a computation decreases as the size of the output goes up. Had we not imposed the dependency of running time on output size via the defect it would not have been possible to define the B combinator.

Let us finally define the C combinator. We start with the following algorithm C_0 given by

$$\{C_0\}(w) = \mathsf{app}(\mathsf{app}(w.1.1, w.2), w.1.2)$$

Clearly,

$$\{C_0\}(\langle\langle x, y\rangle, z\rangle) = xzy$$

The total time t_{tot} needed for this computation is bounded by $t_1 + t_2 + t_b$ where

$$\begin{aligned} u &= xz \\ w &= uy \\ d_1 &= \ell(x) + \ell(z) - \ell(u) \\ d_2 &= \ell(u) + \ell(y) - \ell(w) \\ t_1 &= d_1(\ell(x) + \ell(z) + \gamma)^p \\ t_2 &= d_2(\ell(u) + \ell(y) + \gamma)^p \end{aligned}$$

and, again, t_b is $O((\ell(x) + \ell(y) + \ell(z))^2)$. Therefore, we can find a constant c_0 such that

$$t_{\mathsf{tot}} \leq (d_1 + d_2)(\ell(x) + \ell(y) + \ell(z) + c_0)^p$$

The defect of the computation $\{C_0\}(\langle\langle x, y\rangle, z\rangle)$ is

$$d = \ell(C_0) + 6 + \ell(x) + \ell(y) + \ell(z) - \ell(w) = \ell(C_0) + 6 + d_1 + d_2$$

Therefore, assuming w.l.o.g. that $\ell(C_0) \geq c_0$ we obtain

$$\mathsf{app}(C_0, \langle\langle x, y\rangle, z\rangle) = \{C_0\}(\langle\langle x, y\rangle, z\rangle) = xzy$$

The desired combinator C is again obtained by applying Lemma 4.8 twice. □

Definition 4.10 *The BCK-algebra thus constructed will be called H_p.*

Abbreviations. Let H be a *BCK*-algebra. In view of Lemma 4.2 we will freely use linear lambda terms involving constants from H in order to denote particular elements of H. Moreover, we write $\lambda x_1 x_2 \ldots x_n.t$ for $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.t$. We write T for the pairing combinator $\lambda xyf.fxy$ and P_1, P_2 for the projections $\lambda p.p(\lambda xy.x)$ and $\lambda p.p(\lambda xy.y)$. Note that $P_i(Tt_1t_2) = t_i$.

Itis in general not a good idea, to use projections in order to decompose a variable meant to encode a pair. The reason is that in order to maintain linearity we can use either P_1 or P_2, but not both. The correct way to decompose a pair is to applied it to a function of two arguments which are then bound to the components of a pair. Suppose, for example, that $u, v \in H$ and that we want to define an element $u \otimes v \in H$ such that $(u \otimes v)(Txy) = T(ux)(vy)$. Writing

$$(u \otimes v) =_{\text{def}} \lambda p.T(u(P_1\ p))(v(P_2\ p))$$

does not work since the λ-abstraction is not defined because p occurs twice in its body. We can, however, achieve the desired effect by putting

$$(u \otimes v) =_{\text{def}} \lambda p(\lambda xy.T(ux)(vy))$$

4.3 Truth values and numerals

In every *BCK*-algebra truth values and numerals can be encoded. In concrete examples it is, however, often convenient to use other representations for these basic datatypes than the canonical ones which is why we give them the status of extra structure.

Definition 4.11 *A BCK-algebra H supports truth values and natural numbers if there are distinguished elements $\mathsf{tt}, \mathsf{ff}, D, \mathsf{S}_0, \mathsf{S}_1, G$ and an injection $\mathsf{num} : \mathbb{N} \to H$ such that the following equations are satisfied.*

$$\begin{array}{l} D\ \mathsf{tt}\ x\ y = x \\ D\ \mathsf{ff}\ x\ y = y \\ \mathsf{S}_0\ \mathsf{num}(x) = \mathsf{num}(2x) \\ \mathsf{S}_1\ \mathsf{num}(x) = \mathsf{num}(2x+1) \\ G\ \mathsf{num}(0) = T\ \mathsf{tt}\ (T\ \mathsf{tt}\ \mathsf{tt}) \\ G\ \mathsf{num}(2(x+1)) = T\ \mathsf{ff}\ (T\ \mathsf{tt}\ \mathsf{num}(x+1)) \\ G\ \mathsf{num}(2x+1) = T\ \mathsf{ff}\ (T\ \mathsf{ff}\ \mathsf{num}(x)) \end{array}$$

If φ is an informal statement let $[\varphi]$ be tt if φ is true and ff otherwise. We have

$$\begin{array}{l} P_1(G\ \mathsf{num}(n)) = [n{=}0] \\ P_1(P_2(G\ \mathsf{num}(n))) = [n \text{ is even}] \\ P_2(P_2(G\ \mathsf{num}(n+1))) = \mathsf{num}(\lfloor \frac{n+1}{2} \rfloor) \end{array}$$

Proposition 4.12 *Every BCK-algebra supports truth values and natural numbers.*

Proof. Define $\mathsf{tt} =_{\text{def}} \lambda xy.x$ and $\mathsf{ff} =_{\text{def}} \lambda xy.y$ and $D =_{\text{def}} \lambda txy.txy$. This accounts for the truth values. The injection num is defined recursively by $\mathsf{num}(0) = T\ \mathsf{tt}(T\ \mathsf{tt}\ \mathsf{tt})$, $\mathsf{num}(2(x+1)) = T\ \mathsf{ff}(T\ \mathsf{tt}\ \mathsf{num}(x))$, $\mathsf{num}(2x+1) = T\ \mathsf{ff}(T\mathsf{ff}\ \mathsf{num}(x))$. Now put

$$\begin{array}{ll} \mathsf{S}_0 = \lambda x.x(\lambda ty.D\ t & \\ \quad \mathsf{num}(0) & \text{case } x = 0 \\ \quad (T\ \mathsf{ff}(T\ \mathsf{tt}\ (T\ \mathsf{ff}\ y)) & \text{case } x \neq 0 \\ \mathsf{S}_1 = \lambda x.T\ \mathsf{ff}(T\ \mathsf{ff}\ x) & \\ G = I & \end{array}$$

□

Proposition 4.13 *The algebras H_p support natural numbers and truth values with the settings* $\mathtt{tt} = 1$, $\mathtt{ff} = 0$, *and* num *defined as in Lemma 4.3, i.e.,* $\mathsf{num}(x) = 2x + 1$.

Proof. The missing constants are obtained by parametrisation from the obvious algorithms computing them. □

5 Realisability sets

In this section we define and explore an analogue of the category of realisability sets introduced by Moggi and others based on a *BCK*-algebra supporting truth values and natural numbers. We refer to, e.g., [9] for an introduction to modest sets and realisability. We shall see that due to the absence of an S-combinator hence of diagonalisation, the thus obtained category of modest sets is not cartesian closed. It is, however, an affine linear category w.r.t. to a natural tensor product based on the pairing function and it also has cartesian products, which, however, lack right adjoints, i.e., function spaces.

Unless stated otherwise let H be an arbitrary *BCK*-algebra supporting truth values and natural numbers. For a concrete example the reader may of course think of H_p for H.

Definition 5.1 *An H-set is a pair $X = (|X|, \Vdash_X)$ where $|X|$ is a set and $\Vdash_X \subseteq H \times X$ is a relation such that for each $x \in |X|$ there exists $t \in H$ such that $t \Vdash_X x$.*

A morphism from H-set X to H-set Y is a function $f : |X| \longrightarrow |Y|$ such that there exists an element $e \in H$ with

$$\forall x{\in}X.\forall t{\in}H.t \Vdash_X x \Rightarrow e\ t \Vdash_Y f(x)$$

We write $e \Vdash_{X \multimap Y} f$ in this case.

If $f : |X| \longrightarrow |Y|$ is a set-theoretic function then we say that f is realised by e, if $e \Vdash_{X \multimap Y} f$. So an H-set morphism from X to Y is a function that can be realised.

We will sometimes write X instead of $|X|$ and $\Vdash$ instead of $\Vdash_X$.

Definition and Theorem 5.1 *The H-sets together with their morphisms form an affine linear category $\mathcal{H}$ with the following settings.*

- *Identities and composition are given by set-theoretic identity and composition which are realised by virtue of the I and B combinators.*
- *The tensor product of H-sets X, Y is given by*

$$\begin{array}{l} |X \otimes Y| = |X| \times |Y| \text{ (set-theoretic cartesian product)} \\ t \Vdash_{X \otimes Y} (x, y) \iff \exists u, v.t = Tuv \wedge u \Vdash x \wedge v \Vdash y \end{array}$$

- *The projections are given by $\pi(a, b) = a$ and $\pi'(a, b) = b$.*
- *The terminal object is $\top = \{\langle\rangle\}$ with $\mathtt{tt} \Vdash_\top \langle\rangle$.*

Proof. The projections are obviously jointly monic and the defining equations for the associated morphisms and operators imply that those are defined as in the category of sets, e.g., associativity is given by $\alpha(x,(y,z)) = ((x,y),z)$. Therefore, all that remains to be shown is that the projections as well as these associated morphisms are realisable.

The projections are realised by P_1 and P_2.

If $f : X_1 \longrightarrow Y_1$ and $g : X_2 \longrightarrow Y_2$ then $(f \otimes g)(x_1, x_2)$ equals $(f(x_1), g(x_2))$ and this can be realised by $\lambda p.p(\lambda t_1 t_2.T(dt_1)(et_2))$ when $d \Vdash f$ and $e \Vdash g$.

Symmetry ($X \otimes Y \cong Y \otimes X$) is realised by $\lambda p.p(\lambda xy.Tyx)$.

Associativity ($X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z$) is realised by $\lambda p.p(\lambda xv.v(\lambda yz.T(Txy)z))$.

The unique map $\langle\rangle : X \longrightarrow \top$ is realised by $\lambda x.\mathtt{tt}$.

The isomorphism $X \cong X \otimes \top$ is realised by $\lambda x.Tx\mathtt{tt}$; its inverse is realised by $\lambda t.t(\lambda xy.x)$. Similarly, $X \cong \top \otimes X$. □

Proposition 5.2 *In $\mathcal{H}$ all linear function spaces exist and are given as follows. If $X, Y \in \mathcal{H}$ then $X \multimap Y$ has as underlying set the set of morphisms from X to Y. The realisability relation $\Vdash_{X \multimap Y}$ is as defined above in Def. 5.1, i.e.,*

$$e \Vdash_{X \multimap Y} f \iff \forall t, x.t \Vdash_X x \Rightarrow et \Vdash_Y f(x)$$

The application map $\mathsf{ev} : (X \multimap Y) \otimes X \longrightarrow Y$ is defined by $\mathsf{ev}(f, x) = f(x)$.

Proof. Application is realised by $\lambda z.z(\lambda fx.fx)$. If $f : Z \otimes X \longrightarrow Y$ is realised by e then for each $z \in Z$ the function $x \mapsto f(z, x)$ is realised by $\lambda v.e(\lambda k.kuv)$ when u is a realiser for z. Therefore, we have a function from Z to $(X \multimap Y)$. This function itself is realised by $\lambda uv.e(\lambda k.kuv)$. □

Proposition 5.3 *The category $\mathcal{H}$ has cartesian products given by*

$$|X \times Y| = |X| \times |Y|$$
$$e \Vdash_{X \times Y} (x, y) \iff e\ \mathtt{tt} \Vdash x \wedge e\ \mathtt{ff} \Vdash y$$

Proof. Projections $X \times Y \longrightarrow X$ and $X \times Y \longrightarrow Y$ are realised by $\lambda e.e\ \mathtt{tt}$ and $\lambda e.e\ \mathtt{ff}$, respectively. If $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$ are realised by d and e then the "target-tupled" function $\langle f, g\rangle : Z \longrightarrow X \times Y$ defined by $\langle f, g\rangle(c) = (f(c), g(c))$ is realised by $\lambda xt.(Dtde)x$. □

5.1 Natural numbers and other datatypes

Definition 5.4 *The H-set of natural numbers N has $\mathbb{N}$ as underlying set and realising relation defined by $\mathsf{num}(n) \Vdash_{\mathsf{N}} n$.*

Lemma 5.5 *Suppose that $H = H_p$. The $\mathcal{H}$-morphisms from $\mathsf{N} \otimes \cdots \otimes \mathsf{N}$ (n factors) are the functions $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ which are computable in time $O((|x_1| + \cdots + |x_n|)^p)$ and moreover satisfy $|f(\vec{x})| = |x_1| + \cdots + |x_n| + O(1)$.*

Proof. Direct from Proposition 4.7 and the discussion following Lemma 4.5. □

Proposition 5.6 (Constructors and case distinction) *Let X be an H-set. The functions*

$$\begin{array}{l} 0 : |\top| \longrightarrow |\mathsf{N}| \\ \mathsf{S}_0 : |\mathsf{N}| \longrightarrow |\mathsf{N}| \\ \mathsf{S}_1 : |\mathsf{N}| \longrightarrow |\mathsf{N}| \\ \mathsf{case}^{\mathsf{N}} : |C \times ((\mathsf{N} \multimap C) \times (\mathsf{N} \multimap C))| \longrightarrow |\mathsf{N} \multimap C| \end{array}$$

with $0(\langle\rangle) = 0$, $\mathsf{S}_0(x) = 2x$, $\mathsf{S}_1(x) = 2x+1$, and $\mathsf{case}^{\mathsf{N}}(g, h_0, h_1)(0) = g$, $\mathsf{case}^{\mathsf{N}}(g, h_0, h_1)(2x+i) = h_i(x)$, otherwise, are morphisms in $\mathcal{H}$.

Proof. Obvious for $0, \mathsf{S}_0, \mathsf{S}_1$. A realiser for $\mathsf{case}^{\mathsf{N}}$ is given by

$$\lambda p.\lambda x.G\ x(\lambda uv.Du\ (p\ \mathsf{tt})\ (v(\lambda qy.p\ \mathsf{ff}\ q\ y)))$$

□

Definition 5.7 *Let A be an H-set. We define $\mathsf{T}(A) \in \mathcal{H}$ by $|\mathsf{T}(A)| = \mathsf{T}(|A|)$, i.e., the set of $|A|$-labelled binary trees. The realisability relation $\Vdash_{\mathsf{T}(A)}$ is defined inductively by*

- *If $x \Vdash a$ then $T\ \mathsf{tt}\ x \Vdash \mathsf{leaf}(x)$.*
- *If $x \Vdash a$ and $y \Vdash l$ and $z \Vdash r$ then $T\mathsf{ff}\ (Ta(Tlr)) \Vdash \mathsf{node}(a, l, r)$.*

Proposition 5.8 *The constructors leaf and node give rise to morphisms of the following type*

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

Furthermore, the function

$$\mathsf{case}_X^{\mathsf{T}(A)} : (A \multimap X) \times (A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \multimap X) \multimap \mathsf{T}(A) \multimap X$$

given by

$$\begin{array}{l} \mathsf{case}_X^{\mathsf{T}(A)}(h_{\mathsf{leaf}}, h_{\mathsf{node}})(\mathsf{leaf}(a)) = h_{\mathsf{leaf}}(a) \\ \mathsf{case}_X^{\mathsf{T}(A)}(h_{\mathsf{node}}, h_{\mathsf{node}})(\mathsf{leaf}(a)) = h_{\mathsf{node}}(a, l, r) \end{array}$$

is realisable.

Proof. Analogous to Prop. 5.6 □

Similarly, we can define H-sets of lists and other inductively defined data types.

6 Interpreting recursion

In general (e.g. when $H = H_p$) the category $\mathcal{H}$ itself does not contain all *PTIME*-functions and thus does not allow us to represent patterns of safe recursion on notation.

In order to achieve this we introduce a notion of polynomial time computable function between H-sets which strictly contains the H-set morphisms. Safe recursion then takes the form of an operator on such polynomial time computable functions with the proviso that the step functions be H-set morphisms. In order to give safe recursion the shape of a higher-typed constants we move

to extensional presheaves over a category obtained by integrating the polynomial time computable functions with the H-set morphisms.

In order to be able to define these polynomial time computable functions we need to restrict our attention to BCK-algebras in which application is polynomial time computable. In addition we need to impose further technical conditions all of which are met by the algebras H_p. The reason for introducing another level of abstraction and not working directly with the H_p is that we will later on instantiate these results with another BCK-algebras.

Definition 6.1 *A BCK-algebra H supporting truth values and natural numbers is called* polynomial time computable *(polynomial or PTIME for short) if its carrier set can be identified with a subset of strings so that it is amenable to algorithmic manipulation and there exists a function $\ell : H \longrightarrow \mathbb{N}$ such that*

- *there exists an algorithm f and a polynomial p such that for each $x, y \in H$ the computation $f(x, y)$ terminates after not more than $p(\ell(x) + \ell(y))$ steps with result xy,*
- *for each $x, y \in H$ the following inequalities hold:*

$$\begin{array}{l} \ell(xy) \leq \ell(x) + \ell(y) \\ \ell(x) \leq |x| \\ |Kx| > |x| \\ |Bx| > |x| \\ |Cx| > |x| \\ |Bxy| > |x| + |y| \\ |Cxy| > |x| + |y| \\ |\mathsf{num}(n)| \geq |n| \end{array}$$

The first two inequalities are abstracted from the particular example H_p. They are needed in order to show that iterations of functions represented in H are polynomial time computable.

The other inequalities are sanity conditions ensuring that the partial applications of B, C, K copy their input into the result and do not reduce the size by some sort of information compression.

We summarise a few basic facts about polynomial BCK-algebra in the following lemma:

Lemma 6.2 *Let H be a polynomial time computable BCK-algebra.*

i. The BCK-algebras H_p are polynomial time computable;

ii. For each $x, y \in H$ we have $|Txy| > |x| + |y|$;

iii. The requirement $|\mathsf{num}(n)| \geq |n|$ is satisfied for the canonical encoding (Section 4.12) of natural numbers in BCK-algebra;

iv. $\ell(\mathsf{num}(n)) = O(|n|)$, hence $\Theta(|n|)$;

v. There exists a PTIME-function getnum *such that* $\mathsf{getnum}(\mathsf{num}(n)) = n$.

Proof. Part i is obvious from the definition of the H_p. Part ii follows from $Txy = C(CIx)y$, hence $|Txy| > |CIx| + |y| > |CI| + |x| + |y| \geq |x| + |y|$. For iii we use ii and induction over n. Part iv follows from the existence of constructor functions S_0 and S_1 together with the inequation $\ell(xy) \leq \ell(x) + \ell(y)$.

For part v we note that the function getnum admits a recursive definition in terms of G. The total number of unfoldings of the recursive definition can be a priori bounded by $|x|$.

□

We assume henceforth that our generic BCK-algebra H is polynomial.

Definition 6.3 *Let X, Y be H-sets. A $PTIME$-function from X to Y is a function $f : |X| \longrightarrow |Y|$ such that there exists a one-argument algorithm g and a polynomial p such that whenever $e \Vdash_X x$ then the computation $g(e)$ terminates after not more than $p(|e|)$ steps and $g(e) \Vdash_Y f(x)$.*

We use the notation $f : X \longrightarrow^P Y$ to indicate that f is a $PTIME$-function from X to Y

An algorithm g together with polynomial p as in the above definition will often be called a *realiser* for $PTIME$-function f. If g is a realiser with polynomial p then $\ell(g(e)) \leq p(|e|)$ by $\ell(x) \leq |x|$.

A realiser for a $PTIME$-function f from $X \otimes Y$ to Z can equivalently be given as a two-argument algorithm g together with a two-variable polynomial p such that $d \Vdash x$ and $e \Vdash y$ implies that $\{g\}(x, y)$ terminates in not more than $p(|x|, |y|)$ steps and yields a realiser for $f(x, y)$.

Notice that every morphism of H-sets is a $PTIME$-function between H-sets, but not vice versa. The $PTIME$-functions with ordinary composition clearly form a category but this fact will not be needed.

Also notice that by the estimates of $|\mathsf{num}(n)|$ in Lemma 6.2 a map $f : \mathsf{N} \longrightarrow^P \mathsf{N}$ is the same as a polynomial time computable function on the integers in the usual complexity-theoretic sense.

If $A_1, \ldots, A_m, B_1, \ldots, B_n, C$ are H-sets then a map

$$f : A_1 \otimes \ldots \otimes A_m \longrightarrow^P (B_1 \otimes \ldots \otimes B_n) \multimap C$$

can be viewed as an $m + n$-ary map with the first m inputs "normal" in the sense that the result depends polynomially on their size and the second n inputs "safe" in the sense that the size of the result is majorised by the sum of their sizes plus a constant independent of their sizes. More, formally, if k, q is a realiser for such map f then we have

$$\ell(f(\vec{x}; \vec{y})) \leq q(\ell(\vec{x})) + \sum_{i=1}^{n} \ell(y_i)$$

In this way the polynomial time functions between H-sets generalise Bellantoni's "polymax-bounded"-functions to linearity and higher types. We see that the maximum operation is now replaced by summation which exploits the greater generality offered by linearity.

Theorem 6.4 (Safe recursion on notation) *Let P, X be H-sets. If $h : P \otimes \mathsf{N} \longrightarrow^P X \multimap X$ is a $PTIME$-function as indicated then so is the function $f : |P \otimes \mathsf{N}| \longrightarrow |X \multimap X|$ defined by*

$$\begin{array}{l} f(p, 0)(g) = g \\ f(p, x)(g) = h(p, x)(f(p, \lfloor x/2 \rfloor)(g)) \qquad \textit{when } x > 0 \end{array}$$

Proof. Let k be a realiser for h, i.e. an algorithm such that $t \Vdash_P p$ implies that $k(t, \mathsf{num}(n))$ terminates in not more than than $q(|t|, |\mathsf{num}(n)|)$ steps for some fixed polynomial q. In view of $\ell(x) \leq |x|$ this implies $\ell(k(t, \mathsf{num}(n))) \leq q(|t|, |\mathsf{num}(n)|)$.

In order to realise $\mathsf{rec}^{\mathsf{N}}(h)$ we consider the recursive algorithm $g : H \times H \longrightarrow H$ defined by

$$\begin{array}{l} g(t, x) = \textbf{if } (P_1(G\ x)) = \mathsf{tt} \\ \quad \textbf{then } \lambda v.v \qquad\qquad\qquad\qquad (\text{Case } x = 0) \\ \quad \textbf{else } \lambda v.k(t, x)\ (g(t, \mathsf{div2}\ x)\ v) \end{array}$$

where

$$\mathsf{div2} = \lambda x.P_2(P_2(G\ x))$$

It is clear from the definition that g is a realiser for g provided we can show that it has polynomial runtime for inputs of the form $t \in \mathrm{dom}(P)$ and $x = \mathsf{num}(n)$ for $n \in \mathbb{N}$. Notice that for other inputs $g(t, x)$ may diverge even if k is assumed total. For example x could be such that $\mathsf{div2}\ x = x$.

The length of $g(t,x)$ satisfies the following estimate.

$$\begin{array}{lcl}\ell(g(t,\mathsf{num}(0))) & \leq & c\\ \ell(g(t,\mathsf{num}(n))) & \leq & c+q(|t|,|\mathsf{num}(n)|)+\ell(g(t,\mathsf{num}(\lfloor\frac{n}{2}\rfloor)))\text{, if } n>0\end{array}$$

Therefore,

$$\ell(g(t,\mathsf{num}(n)))\leq |n|\cdot q(|t|,|\mathsf{num}(n)|)+c$$

In order to estimate the time $T(t,x)$ needed to compute $g(t,x)$ we first note that, in fact, g can be written as

$$\begin{array}{l}g(t,x)=\mathbf{if}(P_1(G\ x))=\mathsf{tt}\\ \quad\mathbf{then}\ I \qquad\qquad\qquad\qquad\qquad\qquad (\text{Case } x=0)\\ \quad\mathbf{else}\ B\ k(t,x)\ (g(t,\mathsf{div2}\ x)\ v)\end{array}$$

so that

$$\begin{array}{lcl}T(t,\mathsf{num}(0)) & \leq & c\\ T(t,\mathsf{num}(n)) & \leq & q(|t|,|\mathsf{num}(n)|)+T(t,\mathsf{num}(\lfloor\frac{n}{2}\rfloor))\\ & & +p'(q(|t|,|\mathsf{num}(n)|)+\ell(g(t,\mathsf{num}(\lfloor\frac{n}{2}\rfloor))))\text{, if } n>0\end{array}$$

Here c is a constant and p' is a polynomial obtained from enlarging the polynomial p witnessing that application in H is polynomial time w.r.t. ℓ a little bit so as to account for bookkeeping.

Hence, by induction on n:

$$\begin{array}{l}T(t,\mathsf{num}(n))\\ \quad\leq c+|n|\cdot(q(|t|,|\mathsf{num}(n)|)+\\ \quad\ +p'(q(|t|,|\mathsf{num}(n)|)+|n|\cdot q(|t|,|\mathsf{num}(n)|)+c))\\ \quad\leq c+|\mathsf{num}(n)|(q(|t|,|\mathsf{num}(n)|)+\\ \quad\ +p'(q(|t|,|\mathsf{num}(n)|)+\mathsf{num}(n)\cdot q(|t|,|\mathsf{num}(n)|)+c))\end{array}$$

which is polynomial in $|t|,|\mathsf{num}(n)|$ as required. □

Corollary 6.5 (Duplicable safe parameters) *If P,D,X are H-sets and D is duplicable and $h:P\otimes\mathsf{N}\longrightarrow^P D\otimes X\multimap X$ is a PTIME-function then the function $f:|P\otimes\mathsf{N}|\longrightarrow|D\otimes X\multimap X|$ defined by*

$$\begin{array}{l}f(p,0)(d,g)=g\\ f(p,x)(d,g)=h(p,x)(d,f(p,\lfloor\frac{x}{2}\rfloor)(d,g))\end{array}$$

is a PTIME-function.

Proof. Define $Y=D\multimap X$ and $h':P\otimes\mathsf{N}\longrightarrow^P Y\multimap Y$ by

$$h'(p,n)(u)(d)=\mathsf{let}\ (d_1,d_2)=\delta(d)\ \mathsf{in}\ h(p,x)(d_1,u(d_2))$$

where δ is the diagonal morphism for D. This definition can be formalised as a composition of h with a $\mathcal{H}$-map from $D\otimes X\multimap X$ to $Y\multimap Y$ definable in the language of ALCC using δ and therefore yields a *PTIME*-function without further proof.

Applying Theorem 6.4 yields $f':P\otimes\mathsf{N}\longrightarrow^P Y\multimap Y$ satisfying the corresponding recurrence. The desired f is obtained from f' by

$$f(p,x)(d,g)=(f'(p,x)(\lambda dg.g))\ d\ g$$

Again, this can be seen as composition of f with a map from $(Y\multimap Y)$ to $D\multimap X\multimap X$. □

Theorem 6.6 (Safe tree recursion) *Let $P, X \in \mathcal{H}$ and*

$$h_{\mathsf{leaf}} : P \otimes A \longrightarrow^P X$$
$$h_{\mathsf{node}} : P \otimes A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \longrightarrow^P X \otimes X \multimap X$$

be PTIME-functions as indicated. then the function $f : |P \otimes \mathsf{T}(A)| \longrightarrow |X|$ defined by

$$f(p, \mathsf{leaf}(a)) = h_{\mathsf{leaf}}(p, a)$$
$$f(p, \mathsf{node}(a, l, r)) = h_{\mathsf{node}}(p, a, l, r)(f(l), f(r))$$

is a PTIME-function from $P \otimes \mathsf{T}(A)$ to X.

Proof. Let k_{leaf} and k_{node} be realisers for h_{leaf} and h_{node} viewed as binary, resp. quaternary *PTIME*-functions with witnessing polynomials $q_{\mathsf{leaf}}(t, a)$ and $q_{\mathsf{node}}(t, a, l, r)$. In order to realise f we recursively define a function $g : H \times H \longrightarrow H$ by

$$\begin{array}{l} g(t, x) = D\ x_1 \\ \quad (k_{\mathsf{leaf}}(t, x_2)) \\ \quad (k_{\mathsf{node}}(t, x_{21}, x_{22}, x_{23})(T\ f(z_0, x_{22})\ f(z_0, x_{23}))) \\ \text{where } x_1 = P_1 x, x_2 = P_2 x, x_{21} = P_1(P_2 x), x_{22} = P_1(P_2(P_2 x)), x_{23} = P_2(P_2(P_2 x)) \end{array}$$

Again, it is clear that if this algorithm runs sufficiently fast then it will realise the above function on trees. It thus remains to show that if $t \in \mathrm{dom}(P)$ and $x \in \mathrm{dom}(\mathsf{T}(A))$ then $f(t, x)$ is computable in polynomial time.

Now, we have

$$\ell(f(t, T\mathsf{tt}\ a)) \leq q_{\mathsf{leaf}}(|t|, |a|)$$
$$\ell(f(t, T\mathsf{ff}(Ta(Tlr)))) \leq c + q_{\mathsf{node}}(|t|, |a|, |l|, |r|) + \ell(f(t, l)) + \ell(f(t, r))$$

where c is a constant accounting for the ℓ-length of the function combining the results of the recursive calls.

Therefore, if $x \in \mathrm{dom}(\mathsf{T}(A))$ then using $|Txy| > |x| + |y|$ we obtain

$$\ell(f(t, x)) \leq |x| \cdot q(|t|, |x|)$$

where $q(|t|, |x|)$ majorise $q_{\mathsf{leaf}}(|t|, |a|)$ and $q_{\mathsf{node}}(|t|, |x|, |x|, |x|) + c$.

Now it follows that the runtime of $f(t, x)$ can be estimated by

$$\begin{array}{l} T(t, T\mathsf{tt}\ a) \leq p'(q(|t|, |a|)) \leq p'(|t|, |T\mathsf{tt}a|) \\ T(t, T\mathsf{ff}\ x) \leq q(|t|, |x|) + T(t, l) + T(t, r) + \\ \qquad p'(q(|t|, |x|) + |l| q(|t|, |l|) + |r| q(|t|, |r|)) \\ \qquad \leq T(t, l) + T(t, r) + p'(|t|, |x|) \end{array}$$

where $x = T\mathsf{ff}(Ta(Tlr))$ and p' is a suitably large polynomial. Therefore, (always under the assumption that $x \in \mathrm{dom}(\mathsf{T}(A))$)

$$T(t, x) \leq |x| \cdot p'(|t|, |x|)$$

□

Corollary 6.7 *Let $P, X, D \in \mathcal{H}$ with D duplicable and suppose that*

$$h_{\mathsf{leaf}} : P \otimes A \longrightarrow^P D \multimap X$$
$$h_{\mathsf{node}} : P \otimes A \otimes \mathsf{T}(A) \otimes \mathsf{T}(A) \longrightarrow^P D \otimes X \otimes X \multimap X$$

be PTIME-functions as indicated. then the function $f : |P \otimes \mathsf{T}(A)| \longrightarrow |D \multimap X|$ defined by

$$f(p, \mathsf{leaf}(a))(d) = h_{\mathsf{leaf}}(f, a)(d)$$
$$f(p, \mathsf{node}(a, l, r))(d) = h_{\mathsf{node}}(p, a, l, r)(d, f(l)(d), f(r)(d))$$

is a PTIME-function from $P \otimes \mathsf{T}(A)$ to X.

Proof. Analogous to the proof of Cor. 6.5 using Thm. 6.6 with result type $D \multimap X$. □

Again, we omit the treatment of lists as it is analogous to the previously treated cases.

7 Recursion operators as higher-typed constants

In this section we show how to embed the category of H-sets as well as the *PTIME*-functions between them into a single functor category in which the recursion patterns identified in the preceding four propositions take the form of higher-order constants involving modalities.

The strategy is to first combine H-set morphisms and *PTIME*-functions into a single category $\mathcal{H}^{\Box}$ which is structurally similar to the category $\mathcal{B}$ of "polymax-bounded" functions used in [7].

Definition 7.1 *The category $\mathcal{H}^{\Box}$ has as objects pairs $X = (X_0, X_1)$ where both X_0 and X_1 are H-sets. A morphism from X to Y consists of a PTIME-function $f_0 : X_0 \longrightarrow Y_0$ and a PTIME-function $f_1 : X_0 \longrightarrow (X_1 \multimap Y_1)$. The identity morphism is given by the identity function at X_0 and the constant function yielding the identity morphism at X_1. The composition of (f_0, f_1) and (g_0, g_1) is given by $g_0 \circ f_0$ and $x \mapsto g_1(g_0(x)) \circ f_0(x)$.*

Proposition 7.2 *The following data endow $\mathcal{H}^{\Box}$ with the structure of an ALC.*

The tensor product of $X = (X_0, X_1)$ and $Y = (Y_0, Y_1)$ is given by

$$(X_0, X_1) \otimes (Y_0, Y_1) = (X_0 \otimes Y_0, X_1 \otimes Y_1)$$

The first projection $\pi : X \otimes Y \longrightarrow X$ is given by the obvious canonical maps

$$\pi_0 : X_0 \otimes Y_0 \longrightarrow^P X_0$$
$$\pi_1 : X_0 \otimes Y_0 \longrightarrow^P (X_1 \otimes Y_1) \multimap X_1$$

The second projection is defined analogously.

The terminal object is given by $\top = (\top, \top)$.

Proof. Routine verification. □

The category $\mathcal{H}^{\Box}$ also has cartesian products given by $(X_0, X_1) \times (Y_0, Y_1) = (X_0 \times Y_0, X_0 \times X_1)$ where $\times$ is the cartesian product in $\mathcal{H}$.

An object of the form $(X_0, \top)$ is called *normal*; an object of the form $(\top, X_1)$ is called *safe*.

We notice that an $\mathcal{H}^{\Box}$-map from a safe object to a normal one must be constant.

The category $\mathcal{H}$ can be embedded fully and faithfully into $\mathcal{H}^{\Box}$ via $X \mapsto (\top, X)$ and $\mathcal{H}(X, Y) \ni f \mapsto (\mathrm{id}_{\top}, \hat{f})$ where $\hat{f} : \top \longrightarrow X \multimap Y$ is obtained as the transpose of f composed with the isomorphism $\top \otimes X \cong X$.

This embedding preserves tensor product and cartesian product up to equality and we will therefore treat it as an inclusion thus identifying $\mathcal{H}$ with the full subcategory of $\mathcal{H}^{\Box}$ consisting of the safe objects.

Proposition 7.3 *If $A \in \mathcal{H}^{\Box}$ is arbitrary and B is normal then*

$$A \otimes B \cong A \times B$$

Proof. Suppose that $f : Z \longrightarrow A$ and $g : Z \longrightarrow B$. This means that we have *PTIME*-functions $f_0 : Z_0 \longrightarrow^P A_0$ and $f_1 : Z_0 \longrightarrow^P (Z_1 \multimap A_1)$, as well as $g_0 : Z_0 \longrightarrow^P B$ as indicated. The component $g_1 : Z_0 \longrightarrow^P Z_1 \multimap \top$ is trivial. Now the function sending $z \in |Z_0|$ to $(f_0(z), g_0(z)) \in |A_0 \otimes B_0|$ is a *PTIME*-function, too. Together with f_1 it furnishes the desired $\mathcal{H}^{\Box}$-morphism $\langle f, g \rangle : Z \longrightarrow A \otimes B$. □

Lemma 7.4 *An object $D = (D_0, D_1)$ is duplicable in $\mathcal{H}^{\Box}$ if either D_0 is empty or D_1 is duplicable in $\mathcal{H}$.*

Proof. Immediate calculation. □

Proposition 7.5 *The category $\mathcal{H}^{\Box}$ has linear exponentials $X \multimap Y$ if X and Y are safe. In this case $X \multimap Y$ is also safe and explicitly given by $(\top, X_1 \multimap Y_1)$.*

Proof. If X, Y are safe and $P = (P_0, P_1)$ is arbitrary then a $\mathcal{H}^{\Box}$ morphism from $P \otimes X$ to Y is given by a *PTIME*-function from P_0 to $(P_1 \otimes X) \multimap Y$. But $(P_1 \otimes X) \multimap Y$ is isomorphic to $P_1 \multimap (X \multimap Y)$ whence we obtain a *PTIME*-function from P_0 to $P_1 \multimap X \multimap Y$ which gives a $\mathcal{H}^{\Box}$-morphism from P to $X \multimap Y$. Inverting this process gives the other direction of the required natural isomorphism. □

We note that $\mathcal{H}^{\Box}$ is not an ALCC; in particular the linear function space $(\mathsf{N}, \top) \multimap (\top, \mathsf{N})$ does not exist in $\mathcal{H}^{\Box}$. Suppose for a contradiction that $A = (A_0, A_1)$ was such function space. Then, in particular, we would have an evaluation map $\mathrm{ev} : A_0 \otimes \mathsf{N} \longrightarrow^P A_1 \multimap \mathsf{N}$ which has the property that for every "true" *PTIME*-function f there exist elements $a_0 \in |A_0|$ and $a_1 \in |A_1|$ such that $f(x) = \mathrm{ev}(a_0, x)(a_1)$. But this would mean that ev is a universal polynomial time computable function which is impossible by diagonalisation.

The lacking function spaces can be added to $\mathcal{H}^{\Box}$ by moving to the functor category $\mathrm{Ext}(\mathcal{H}^{\Box})$ described in Section 3.2. In order that this functor category exists we must make sure that the category $\mathcal{H}^{\Box}$ is small.

This can be achieved by requiring that the underlying sets of H-sets be taken from a suitably chosen universe $\mathcal{U}$ closed under all set-theoretic operations required to form the H-sets of interest. Note that such universe can be defined by a simple inductive process and in particular no "large cardinal assumption" is needed for its existence.

It now follows from the results presented in Section 3.2 that $\mathrm{Ext}(\mathcal{H}^{\Box})$ is an ALCC and that the Yoneda embedding $\mathcal{Y} : \mathcal{H}^{\Box} \longrightarrow \mathrm{Ext}(\mathcal{H}^{\Box})$ preserves the ALC structure as well as existing linear function spaces and cartesian products. In particular, the linear function spaces between safe objects are preserved by the embedding.

Moreover, $\mathrm{Ext}(\mathcal{H}^{\Box})$ supports a comonad ! with the property that whenever $D \in \mathcal{H}^{\Box}$ is duplicable then $!D = D$ in $\mathrm{Ext}(\mathcal{H}^{\Box})$ and for arbitrary presheaf $F \in \mathrm{Ext}(\mathcal{H}^{\Box})$ the presheaf $!F$ is duplicable.

7.1 Polynomial-time functions via a comonad

In this section we identify a comonad $\Box$ on $\mathrm{Ext}(\mathcal{H}^{\Box})$ which has the property that if X is safe then $\Box(X) \cong (X, \top)$ so that by the characterisation of linear function space with representable presehaves in Section 3.2 we have $\Box(X) \multimap F_{(Y_0, Y_1)} \cong F_{(Y_0 \otimes X, Y_1)}$.

For presheaf F we define $\Box F$ by

$$\Box F_{(Z_0, Z_1)} = F_{(Z_0, \top)}$$

Notice that if X is safe then $\Box(\mathcal{Y}(X)) \cong \mathcal{Y}(\Box X)$: At argument Z both presheaves consist of *PTIME*-functions from Z_0 to X.

If $Z \in \mathcal{H}^{\Box}$ write Z_0 for the *normal* object $(Z_0, \top)$ and $p_Z : Y \longrightarrow Z_0$ for the obvious projection arising from $Z \cong Z_0 \otimes Z_1$. The *counit* $\mathsf{unbox}_F : \Box F \longrightarrow F$ is then defined by $(\mathsf{unbox}_F)_Z = F_{p_Z}$.

Like in the case of the comonad ! we have $\Box\Box = \Box$ and so when $f : \Box F \longrightarrow G$ we have $\Box f : \Box F \longrightarrow \Box G$.

Proposition 7.6 *For presheaves F, G we have $F \otimes \Box G = F \times \Box G$ and $\Box F \multimap G = \Box F \to G$ where $\times, \to$ are cartesian product and ordinary function space of presheaves. Moreover, $\Box(F \otimes G) = \Box F \otimes \Box G = \Box F \times \Box G$.*

Proof. Suppose that $X, Y \in \mathcal{H}^{\Box}$, write Y_0 for the normal object $(Y_0, \top)$. Suppose, furthermore, that $f \in F_X$ and $g \in \Box G_Y$. We have $(f, g) \in (F \otimes \Box G)_{(X,Y)}$ by Prop 7.3 and $\bar{f} = f, \bar{g} = g$.

The equality of the function spaces then is a direct consequence:

$$\begin{array}{ll} & \Box F \multimap G_{(X,Y)} \\ = & \mathrm{Ext}(\mathcal{H}^{\Box})((X,Y) \otimes \Box F, G) \\ = & \mathrm{Ext}(\mathcal{H}^{\Box})((X,Y) \times \Box F, G) \\ = & \Box F \to G_{(X,Y)} \end{array}$$

The last part is similar to the first. □

Proposition 7.7 *Let $F \in \mathrm{Ext}(\mathcal{H}^{\Box})$. We have $!\Box F = \Box\, !F = \Box F$.*

Proof. Clearly, $\Box\, !F \subseteq \Box F$ and $!\Box F \subseteq \Box F$. To show $\Box F \subseteq\, !\Box F$ we notice that whenever $X = (X_0, X_1) \in \mathcal{H}^{\Box}$ then $D = (X_0, \top)$ is duplicable and so, if $f \in \Box F_X$ then $f \in\, !\Box F_X$ can be witnessed by the projection $X \longrightarrow D$ and f itself. Since D is normal this also shows the other inclusion. □

Finally, we notice that $\mathcal{G}(\Box F) \cong \mathcal{G}(F)$ as $\mathcal{G}(\Box F) \cong F_{\top} = F_{(\top,\top)} = \Box F_{(\top,\top)} \cong \mathcal{G}(\Box F)$. Also recall that $\mathcal{G}(!\,F) = \mathcal{G}(F)$. We will now see how the recursion patterns defined in Theorems 6.4 and 6.6 can be lifted to $\mathrm{Ext}(\mathcal{H}^{\Box})$.

Theorem 7.8 *Let X be a safe presheaf. There exists a global element*

$$\mathsf{rec}^{\mathsf{N}} : !(\Box\mathsf{N} \multimap X \multimap X) \multimap \Box\mathsf{N} \multimap X \multimap X$$

such that the following equation holds for global elements $h : \Box\mathsf{N} \multimap X \multimap X$, $g : X$, $x \in \mathcal{G}(\mathsf{N}) \setminus \{0\}$.

$$\begin{array}{l} \mathsf{rec}^{\mathsf{N}}\ h\ 0\ g = g \\ \mathsf{rec}^{\mathsf{N}}\ h\ x\ g = h\ x\ (\mathsf{rec}^{\mathsf{N}}\ h\ \ \lfloor \frac{x}{2} \rfloor\ \ g) \end{array}$$

Proof. We must define a natural transformation from $!(\Box\mathsf{N} \multimap X \multimap X)$ to $\Box\mathsf{N} \multimap X \multimap X$.

Suppose that $Z \in \mathcal{H}^{\Box}$ and assume

$$h \in\, !(\Box\mathsf{N} \multimap X \multimap X)_Z$$

witnessed by $t : Z \longrightarrow D$ and $\bar{h} \in (\Box\mathsf{N} \multimap X \multimap X)_D$ where $D = (D_0, D_1)$ is duplicable.

Now we have

$$\begin{array}{l} (\Box\mathsf{N} \multimap X \multimap X)_Z \cong \mathcal{H}^{\Box}((Z_0 \otimes \mathsf{N}, Z_1)\,,\, (\top, X \multimap X)) \\ (\Box\mathsf{N} \multimap X \multimap X)_D \cong \mathcal{H}^{\Box}((D_0 \otimes \mathsf{N}, D_1)\,,\, (\top, X \multimap X)) \end{array}$$

$$\bar{h} : D_0 \otimes \mathsf{N} \longrightarrow^P (D_1 \otimes X) \multimap X$$

and

$$h : Z_0 \otimes \mathsf{N} \longrightarrow^P (Z_1 \otimes X) \multimap X$$

and $t_0 : Z_0 \longrightarrow^P D_0$, $t_1 : Z_0 \longrightarrow^P Z_1 \multimap D_1$ and

$$h(z_0, x)(z_1, g) = \bar{h}(t_0(z_0), x)(t_1(z_0, z_1), g)$$

We must define (again, neglecting isomorphism) a function

$$f = \mathsf{rec}^{\mathsf{N}}_{Z}(h) : Z_0 \otimes \mathsf{N} \longrightarrow^{P} Z_1 \multimap X \multimap X$$

such that

$$\begin{aligned} f(0)(g) &= g \\ f(x)(g) &= h(z_0, x)(f(\lfloor \tfrac{x}{2} \rfloor)(g)) \end{aligned}$$

Since $D = (D_0, D_1)$ is duplicable we know by Lemma 7.4 that either D_0 is empty or D_1 is a duplicable H-set. In the former case, Z_0 must also be empty and so f will be the empty function. Otherwise, Corollary 6.5 gives us a function $f' : D_0 \otimes \mathsf{N} \longrightarrow^{P} D_1 \multimap X \multimap X$ such that

$$\begin{aligned} f'(d_0, 0)(d_1, g) &= g \\ f'(d_0, x)(d_1, g) &= \bar{h}(d_0, x)(f'(d_0, \lfloor \tfrac{x}{2} \rfloor)(g)) \end{aligned}$$

Now we define

$$f(z_0, x) = \lambda z_1 g. f'(t_0(z_0), x)\ (t_1(z_0)\ z_1)\ g$$

and the desired equations follow by induction on x. Since the thus defined f is uniquely determined by the recursive equations it does not depend on the witness $\bar{h}$.

Naturality of the assignment $f \mapsto \mathsf{rec}^{\mathsf{N}}(h)$ means that recursive definition respects substitution of parameters and is also readily established by induction on x. □

Similarly, we can lift tree recursion to $\mathrm{Ext}(\mathcal{H}^{\Box})$:

Theorem 7.9 *Let A, X be safe presheaves. There exists a global element*

$$\mathsf{rec}^{\mathsf{T}(A)} : !(\Box A \multimap X) \multimap !(\Box A \multimap \Box \mathsf{T}(A) \multimap \Box \mathsf{T}(A) \multimap X \multimap X \multimap X) \multimap \Box \mathsf{T}(A) \multimap X$$

such that the following equations are valid for appropriate global elements.

$$\begin{aligned} &\mathsf{rec}^{\mathsf{T}(A)}(g, h, \mathsf{leaf}(a)) = g \\ &\mathsf{rec}^{\mathsf{T}(A)}(g, h, \mathsf{node}(a, l, r)) = h(a, l, r, \mathsf{rec}^{\mathsf{T}(A)}(g, h, l), \mathsf{rec}^{\mathsf{T}(A)}(g, h, r)) \end{aligned}$$

Proof. Analogous to the previous one, this time using Cor. 6.7. □

In this way, other recursion patterns we might be interested in can also be lifted to $\mathrm{Ext}(\mathcal{H}^{\Box})$.

7.2 Interpretation of SLR

We are now ready to define an interpretation of SLR without rule S-Ax in $\mathrm{Ext}(\mathbb{C})$. We show later in Section 8 how to encompass that rule if so desired.

To each aspect a we associate a functor

$$\begin{aligned} F_a(X) &= X, \text{ if } a = (\text{nonmodal}, \text{linear}) \\ F_a(X) &= !X, \text{ if } a = (\text{nonmodal}, \text{nonlinear}) \\ F_a(X) &= \Box X, \text{ if } a = (\text{modal}, \text{nonlinear}) \end{aligned}$$

We also define a natural transformation $\varepsilon_a : F_a \longrightarrow Id$ as either the identity, $\mathsf{derelict}$, or unbox.

If $f : F_a(X) \longrightarrow Y$ then we have $F_a(f) : F_a(X) \longrightarrow F_a(X)$ since $F_a F_a = F_a$ for each a. If $a <: a'$ then we define a natural transformation $\iota_{a,a'} : F_a(X) \longrightarrow F_{a'}(X)$ by

$$\begin{aligned} \iota_{a,a'} &= \varepsilon_a && \text{if } a = (\text{nonmodal}, \text{linear}) \\ \iota_{a,a'} &= \mathrm{id} && \text{if } a = a' \\ \iota_{a,a'} &= !(\mathsf{unbox}) && \text{if } a = (\text{modal}, \text{nonlinear}), a' = (\text{nonmodal}, \text{nonlinear}) \end{aligned}$$

Let η be a partial function mapping type variables to objects of $\mathcal{H}$ and A be a type. The presheaf $[\![A]\!]\eta$ is defined by

$$\begin{array}{l}
[\![X]\!]\eta = \eta(X) \\
[\![\mathsf{N}]\!]\eta = \mathsf{N} \\
[\![\mathsf{L}(A)]\!]\eta = \mathsf{L}([\![A]\!]\eta) \\
[\![\mathsf{T}(A)]\!]\eta = \mathsf{T}([\![A]\!]\eta) \\
[\![A \stackrel{a}{\longrightarrow} B]\!]\eta = F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta \\
[\![\forall X.A]\!]\eta = \prod_{B \in \mathcal{H}} [\![A]\!]\eta[X \mapsto B] \\
[\![A \times B]\!]\eta = [\![A]\!]\eta \times [\![B]\!]\eta \\
[\![A \otimes B]\!]\eta = [\![A]\!]\eta \otimes [\![B]\!]\eta
\end{array}$$

Here $\prod_{B \in \mathcal{H}}$ is a $|\mathcal{H}|$-indexed cartesian product of presheaves, defined pointwise.

We notice that if A is safe then $[\![A]\!]\eta$ is a safe object so that the defining clauses for $\mathsf{L}(A)$ and $\mathsf{T}(A)$ make sense.

A context $\Gamma = x_1 \stackrel{a_1}{:} A_1, \ldots, x_n \stackrel{a_n}{:} A_n$ gets interpreted as the tensor product

$$[\![\Gamma]\!]\eta =_{\mathrm{def}} F_{a_1}([\![A_1]\!]\eta) \otimes \cdots \otimes F_{a_n}([\![A_n]\!]\eta)$$

A *derivation* of a judgement $\Gamma \vdash e : A$ gets interpreted as a morphism

$$[\![\Gamma \vdash e : A]\!]\eta : [\![\Gamma]\!]\eta \longrightarrow [\![A]\!]\eta$$

Before actually defining this interpretation let us warn the reader that we will *not* prove that the interpretation is independent of the chosen typing derivation. Neither will we prove that it enjoys one or the other substitution property and neither will we prove that it validates whatsoever equational theory between terms. We are confident that such properties could be established if so desired, but they are not needed for the present development.

7.2.1 Constants and variables

A variable gets interpreted as the corresponding projection morphism possibly followed by a counit ε_a. For example, if $a = (\mathrm{modal}, \mathrm{nonlinear})$ then $\Gamma, x \stackrel{a}{:} A \vdash x : A$ gets interpreted as the morphism

$$[\![\Gamma, x \stackrel{a}{:} A]\!]\eta = [\![\Gamma]\!]\eta \otimes \Box[\![A]\!]\eta \longrightarrow \Box[\![A]\!]\eta \longrightarrow [\![A]\!]\eta$$

The first order constants such as $\mathsf{S}_0, \mathsf{S}_1, \mathsf{node}$, etc. get interpreted by applying the Yoneda embedding to their interpretations in $\mathcal{H}^{\Box}$. The recursors are interpreted as the terminal projection $[\![\Gamma]\!] \longrightarrow \top$ followed by the global elements defined in Theorems 7.8, 7.9.

7.2.2 Application and abstraction

It follows from Proposition 7.7 and Lemma 3.3 that $[\![\Gamma]\!]\eta$ is duplicable whenever Γ is nonlinear.

More generally, if Γ is nonlinear and Δ_1, Δ_2 are arbitrary as in rules T-Arr-E and T-Tens-E then we can define a map

$$v_{\Gamma,\Delta_1,\Delta_2} : [\![\Gamma, \Delta_1, \Delta_2]\!]\eta \longrightarrow [\![\Gamma, \Delta_1]\!]\eta \otimes [\![\Gamma, \Delta_2]\!]\eta$$

as

$$v_{\Gamma,\Delta_1,\Delta_2} =_{\mathrm{def}} w_1 \circ (\delta \otimes ([\![\Delta_1]\!]\eta \otimes [\![\Delta_2]\!]\eta)) \circ w_2$$

where δ is the diagonal on $[\![\Gamma]\!]\eta$ and w_1, w_2 are wiring maps.

Suppose now that $\Gamma, x \stackrel{a}{:} A \vdash e : B$ and let $f : [\![\Gamma]\!]\eta \otimes F_a([\![A]\!]\eta) \longrightarrow [\![B]\!]\eta$ be the interpretation of e. The currying or exponential transpose of this morphism yields a map $[\![\Gamma]\!] \longrightarrow F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta$ which serves as the interpretation of $\lambda x\colon A.e : A \stackrel{a}{\longrightarrow} B$.

Now suppose that

$$\begin{array}{l}\Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B \\ \Gamma, \Delta_2 \vdash e_2 : A \\ \Gamma, \Delta_2 <: a \\ \Gamma \text{ nonlinear}\end{array}$$

as in the premises to rule T-Arr-E and let

$$\begin{array}{l} f_1 : [\![\Gamma]\!]\eta \otimes [\![\Delta_1]\!]\eta \longrightarrow F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta \\ f_2 : [\![\Gamma]\!]\eta \otimes [\![\Delta_2]\!]\eta \longrightarrow [\![A]\!]\eta \end{array}$$

be the interpretations of e_1 and e_2. The side condition on the aspects in Γ, Δ_2 together with the fact that $\Box$ commutes with $\otimes$ allows us to "raise" f_2 to a morphism $\Box f : [\![\Gamma]\!]\eta \otimes [\![\Delta_2]\!]\eta \longrightarrow F_a([\![A]\!]\eta)$.

Now the interpretation of $e_1 e_2$ is obtained as $\mathsf{ev} \circ (f_1 \otimes \Box f_2) \circ v_{\Gamma, \Delta_1, \Delta_2}$ where $\mathsf{ev} : (F_a([\![A]\!]\eta) \multimap [\![B]\!]\eta) \otimes [\![A]\!]\eta \longrightarrow [\![B]\!]\eta$ is the evaluation morphism and $v_{\Gamma, \Delta_1, \Delta_2}$ is defined as described above using the fact that Γ is nonlinear.

7.2.3 Polymorphic abstraction and application

Suppose that $\Gamma \vdash e : A$ and that X does not occur in Γ. Then the interpretation of $\Gamma \vdash \Lambda X.e : \forall X.A$ is defined by

$$[\![\Gamma \vdash \Lambda X.e : \forall X.A]\!]\eta = \langle [\![\Gamma \vdash e : A]\!]\eta[X \mapsto B] \mid B \in \mathcal{H} \rangle$$

If $\Gamma \vdash e : \forall X.A$ and B is safe then we define

$$[\![\Gamma \vdash e[B] : A[B/X]]\!]\eta = \pi_{[\![B]\!]\eta} \circ [\![\Gamma \vdash e : \forall X.A]\!]\eta$$

7.2.4 Cartesian products

Suppose that $\Gamma \vdash e_1 : A_1$, $\Gamma \vdash e_2 : A_2$. Then we define

$$[\![\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2]\!]\eta = \langle [\![\Gamma \vdash e_1 : A_1]\!]\eta, [\![\Gamma \vdash e_2 : A_2]\!]\eta \rangle$$

where $\langle -, - \rangle$ is the pairing operation associated with cartesian products in $\mathcal{H}^{\Box}$. If $\Gamma \vdash e : A_1 \times A_2$ then we define

$$[\![\Gamma \vdash e.i : A_i]\!]\eta = \pi_i \circ [\![\Gamma \vdash e : A_1 \times A_2]\!]\eta$$

where $\pi_i : [\![A_1 \times A_2]\!]\eta \longrightarrow [\![A_i]\!]\eta$ is the projection morphism.

7.2.5 Tensor products

Suppose that $\Gamma, \Delta_i \vdash e_i : A_i$ and that Γ is nonlinear as in the premise to rule T-Tens-I. Using the diagonal on $[\![\Gamma]\!]\eta$ we can define a map

$$v : [\![\Gamma, \Delta_1, \Delta_2]\!]\eta \longrightarrow [\![\Gamma, \Delta_1]\!]\eta \otimes [\![\Gamma, \Delta_2]\!]\eta$$

as in the previous case. Then we define,

$$[\![\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : A_1 \otimes A_2]\!]\eta = ([\![\Gamma, \Delta_1 \vdash e_1 : A_1]\!]\eta \otimes [\![\Gamma, \Delta_2 \vdash e_2 : A_2]\!]\eta) \circ v$$

Now suppose that

$$\begin{array}{l}\Gamma, \Delta_1 \vdash e_1 : A_1 \otimes A_2 \\ \Gamma, \Delta_2, x \overset{a_1}{:} A_1, x \overset{a_2}{:} A_2 \vdash e_2 : B \\ \Gamma, \Delta_1 <: a_1 \wedge a_2 \\ \Gamma \text{ nonlinear}\end{array}$$

as in the premise to T-Tens-E.

Let $f_1 : [\![\Gamma, \Delta \vdash e_1]\!]\eta \longrightarrow [\![A_1 \otimes A_2]\!]$ and $f_2 : [\![\Gamma, \Delta_2, x \overset{a_1}{:} A_1, x \overset{a_2}{:} A_2]\!] \longrightarrow [\![B]\!]$ be the interpretations of e_1 and e_2.

Since $\Gamma, \Delta_1 <: a_1 \wedge a_2$ we can raise f_1 to form a morphism

$$F_{a_1 \wedge a_2}(f_1) : [\![\Gamma, \Delta_1]\!] \longrightarrow F_a([\![A_1]\!]) \otimes F_a([\![A_2]\!])$$

from which we obtain

$$g =_{\text{def}} (\iota_{a_1 \wedge a_2, a_2} \otimes \iota_{a_1 \wedge a_2, a_2}) \circ F_{a_1 \wedge a_2}(f_1) : [\![\Gamma, \Delta_1]\!] \longrightarrow F_{a_1}([\![A_1]\!]) \otimes F_{a_2}([\![A_2]\!])$$

We then define the interpretation of $\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{let}\, e_1 = x \otimes y \,\mathsf{in}\, e_2 : B$ as

$$f_2 \circ ([\![\Gamma, \Delta_2]\!] \otimes g) \circ v_{\Gamma, \Delta_1, \Delta_2}$$

where $v_{\Gamma, \Delta_1, \Delta_2}$ is as above in Section 7.2.2

7.2.6 Subtyping and subsumption

By induction on the definition of subtyping we define a coercion map $\iota_{A,B} : [\![A]\!]\eta \longrightarrow [\![B]\!]\eta$ when $A <: B$ using identity, composition, the mappings $\iota_{a,a'}$, and functoriality of the semantic type formers $\multimap, \otimes, \times, \prod$.

If $\Gamma \vdash e : B$ was obtained from $\Gamma \vdash e : A$ and $A <: B$ by rule T-Sub then we define

$$[\![\Gamma \vdash e : B]\!]\eta = \iota_{A,B} \circ [\![\Gamma \vdash e : A]\!]\eta$$

7.3 Main result

Let us temporarily write $[\![-]\!]^S$ for the set-theoretic interpretation and $[\![-]\!]^F$ for the interpretation in $\mathrm{Ext}(\mathcal{H}^\Box)$.

If $f \in \mathrm{Ext}(\mathcal{H}^\Box)(\Box\mathsf{N}, \mathsf{N})$ then it follows from the Yoneda Lemma that $\mathcal{G}(f) : \mathbb{N} \cong \mathcal{G}(\Box\mathsf{N}) \longrightarrow \mathcal{G}(\mathsf{N}) \cong \mathbb{N}$ is a polynomial time computable function. This means that if $e : \Box\mathsf{N} \to \mathsf{N}$ is a closed term then $\mathcal{G}([\![f]\!]^F)$ is a *PTIME*-function and the same goes for functions with several arguments. It remains to show that the thus obtained function coincides with the intended set-theoretic meaning. To do this, we use again a logical relation.

Let η, ζ be partial functions mapping type variables to sets in $\mathcal{U}$ and safe presheaves, respectively. Let ρ be a partial function which assigns to each type variable X a binary relation $\rho(X) \subseteq \eta(X) \times \mathcal{G}(\zeta(X))$. We define a relation $R_{\eta,\zeta,\rho}(A) \subseteq [\![A]\!]^S\eta \times \mathcal{G}([\![A]\!]^F\zeta)$ by

$$\begin{array}{rcl}
x\, R_{\eta,\zeta,\rho}(X)\, y & \iff & x\, \rho(X)\, y \\
x\, R_{\eta,\zeta,\rho}(\mathsf{N})\, y & \iff & x = y \\
u\, R_{\eta,\zeta,\rho}(A \overset{a}{\longrightarrow} B)\, v & \iff & \\
\multicolumn{3}{c}{\forall x \in [\![A]\!]^S\eta . \forall y \in \mathcal{G}([\![A]\!]^F\rho) . x\, R_{\eta,\zeta,\rho}(A)\, y \Rightarrow u(x)\, R_{\eta,\zeta,\rho}(B)\, v(y)} \\
u\, R_{\eta,\zeta,\rho}(\forall X.A)\, v & \iff & \\
\multicolumn{3}{c}{\forall U, V \in \mathcal{H} . \forall R \subseteq |U| \times |V| . \pi_U(u)\, R_{\eta[X \mapsto U], \zeta[X \mapsto V], \rho[X \mapsto R]}\, (A)\, \pi_V(v)} \\
(u_1, u_2)\, R_{\eta,\zeta,\rho}(A_1 \times A_2)\, (v_1, v_2) & \iff & u_i\, R_{\eta,\zeta,\rho}(A_i)\, v_i \;\; i = 1, 2 \\
(u_1, u_2)\, R_{\eta,\zeta,\rho}(A_1 \otimes A_2)\, (v_1, v_2) & \iff & u_i\, R_{\eta,\zeta,\rho}(A_i)\, v_i \;\; i = 1, 2
\end{array}$$

The relations $R_{\eta,\zeta,\rho}(\mathsf{L}(A))$ and $R_{\eta,\zeta,\rho}(\mathsf{T}(A))$ are defined inductively by

$$\begin{array}{l}
\mathsf{nil}\, R_{\eta,\zeta,\rho}(\mathsf{L}(A))\, \mathsf{nil} \quad \text{always} \\
\mathsf{cons}(a, l)\, R_{\eta,\zeta,\rho}\, \mathsf{cons}(a', l') \iff \\
\qquad a\, R_{\eta,\zeta,\rho}(A)\, a' \wedge l\, R_{\eta,\zeta,\rho}(\mathsf{L}(A))\, l'
\end{array}$$

$$\begin{array}{l}
\mathsf{leaf}(a)\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, \mathsf{leaf}(a') \iff a\, R_{\eta,\zeta,\rho}(A)\, a' \\
\mathsf{node}(a, l, r)\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, \mathsf{node}(a', l', r') \iff \\
\qquad a\, R_{\eta,\zeta,\rho}(A)\, a' \wedge l\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, l' \wedge r\, R_{\eta,\zeta,\rho}(\mathsf{T}(A))\, r'
\end{array}$$

Now, a direct inspection of the definitions shows that we have

$$[\![c]\!]^S \, R_A \, \mathcal{G}([\![c]\!]^F)$$

for every SLR-constant $c : A$.

Next we show by induction on subtyping derivations that whenever $A <: B$ then

$$\forall x \in [\![A]\!]^S \eta . \forall y \in \mathcal{G}([\![A]\!]^F \rho) . x \; R_{\eta,\zeta,\rho}(A) \; y \Rightarrow x \; R_{\eta,\zeta,\rho}(B) \; \iota_{A,B}(y)$$

Notice here that $[\![A]\!]^S = [\![B]\!]^S$ whenever $A <: B$. Finally, by induction on typing derivations we prove the following extension of the "Fundamental Lemma of Logical Relation" [20].

Proposition 7.10 *Suppose that $\Gamma \vdash e : A$ and that η, ζ, ρ are assignments as above. Suppose, further that $\gamma^S \in \mathcal{G}([\![\Gamma]\!]^S)$ and $\gamma^F \in \mathcal{G}([\![\Gamma]\!]^F)$ are such that $\gamma^S(x) \; R_{\eta,\zeta,\rho}(\Gamma(x)) \; \gamma^F(x)$ for each $x \in \mathrm{dom}(\Gamma)$. Then*

$$[\![\Gamma \vdash e : A]\!]\eta^S(\gamma^S) \; R_{\eta,\zeta,\rho}(A) \; [\![\Gamma \vdash e : A]\!]\rho^F$$

Corollary 7.11 *The set-theoretic interpretation of a closed term $f : \Box\mathsf{N} \to \mathsf{N}$ is a polynomial time computable function.*

Proof. By specialising the above proposition to $\Gamma = \emptyset, A = \Box\mathsf{N} \to \mathsf{N}$ and expanding the definitions. □

7.4 Interpretation of $\lambda^{\Box !}$

We can use $\mathrm{Ext}(\mathcal{H}^\Box)$ also in order to give meaning to the alternative system $\lambda^{\Box !}$ thus providing a proof that also the definable functions of the latter system are *PTIME*. Spelling this out in detail would be a rather boring exercise in typesetting so we will only set out a few of the important points. Types are interpreted as objects of $\mathrm{Ext}(\mathcal{H}^\Box)$. The new type formers $\Box, !$ are interpreted by the eponymous comonads on $\mathrm{Ext}(\mathcal{H}^\Box)$. The associated operations on morphisms provide meaning for the term formers associated with the modalities. The defining clauses for cartesian and tensor product follow the interpretation of SLR.

It seems plausible that any other reasonable formulation of modal/linear lambda calculus including the one in [1] can be interpreted in $\mathrm{Ext}(\mathcal{H}^\Box)$ in a similar fashion.

8 Duplicable numerals

The algebra H has the disadvantage that even values of type N may not be duplicated, i.e, there does not exist an element $\delta \in H$ such that $\delta \mathsf{num}(x) = \lambda f . f \mathsf{num}(x) \mathsf{num}(x)$. Indeed, assuming such diagonal element would contradict Theorem 6.4 for the following reason. Multiplication is easily seen to be a morphism from $\mathsf{N} \otimes \mathsf{N}$ to N. Using the hypothetical δ we could realise the diagonal function from N to $\mathsf{N} \otimes \mathsf{N}$ and thus by composition the squaring function would be a morphism from N to N. Iterating it using safe recursion on notation would allow us to define a function of exponential growth.

In Bellantoni-Cook's original system and in SLR duplication of values of integer type is, however, permitted and sound. The reason is that multiplication is not among the basic functions of these systems and as an invariant it is maintained that a function depending on several safe arguments of integer type is bounded by the maximum of these arguments plus a constant. Obviously, multiplication does not have this property.

In order to obtain an analogue of the algebra H we need to get a handle on the maximum of the lengths of the two components of a pair. This motivates the following definitions. First, like in Lemma 4.3 we fix disjoint injections

$$\begin{aligned}&\mathsf{num} : \mathbb{N} \longrightarrow \mathbb{N}\\ &\mathsf{pad} : \mathbb{N} \longrightarrow \mathbb{N}\\ &\langle\cdot,\cdot\rangle : \mathbb{N}\times\mathbb{N}\to\mathbb{N}\end{aligned}$$

together with test functions isnum, ispad, ispair and inverses getnum, getpad, .1, .2 computable in linear time and satisfying specifications and size restrictions analogous to the ones in Lemma 4.3. We need two bits now to distinguish the ranges of the injections, so we will have $|\mathsf{num}(x)| = |\mathsf{pad}(x)| = 2$ and $|\langle u, v\rangle| = |u| + |v| + 2||v|| + 4$. (Recall that two bits are needed to separate the $|v|$ block from u, v.)

Definition 8.1 (Linear length, maximum length) *The length functions $\ell_{\mathrm{lin}}(x)$ (linear length) and $\ell_{\max}(x)$ (maximum length) are defined recursively as follows.*

$$\begin{aligned}&\ell_{\mathrm{lin}}(\mathsf{num}(x)) = 1\\ &\ell_{\max}(\mathsf{num}(x)) = |x|\\ &\ell_{\mathrm{lin}}(\mathsf{pad}(x)) = |x| + 1\\ &\ell_{\max}(\mathsf{pad}(x)) = 0\\ &\ell_{\mathrm{lin}}(\langle x, y\rangle) = 4 + \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y)\\ &\ell_{\max}(\langle x, y\rangle) = \max(\ell_{\max}(x), \ell_{\max}(y))\\ &\ell_{\mathrm{lin}}(x) = |x|,\ \textit{otherwise}\\ &\ell_{\max}(x) = 0,\ \textit{otherwise}\end{aligned}$$

We may assume that $\ell_{\mathrm{lin}}(0) = \ell_{\max}(0) = 0$.

Lemma 8.2 *The following inequalities hold for every $x \in \mathbb{N}$.*

$$\begin{aligned}&|x| \geq \ell_{\mathrm{lin}}(x) + \ell_{\max}(x)\\ &|x| \leq \ell_{\mathrm{lin}}(x)(1 + ||x|| + \ell_{\max}(x))\end{aligned}$$

Proof. By course of values induction on x. If $x = \mathsf{num}(y)$ then

$$|x| = |y| + 2 \geq 1 + |y| = \ell_{\mathrm{lin}}(x) + \ell_{\max}(x)$$

and

$$|x| = |y| + 2 \leq 1 \cdot (1 + 2 + \ell_{\max}(x)) \leq \ell_{\mathrm{lin}}(x)(1 + ||x|| + \ell_{\max}(x))$$

since $||2|| = 2$.

If $x = \mathsf{pad}(y)$ then

$$|x| = |y| + 2 \geq |y| + 1 = \ell_{\mathrm{lin}}(x) + \ell_{\max}(x)$$

and

$$|x| = |y| + 2 \leq (|y| + 1)(1 + 2) \leq \ell_{\mathrm{lin}}(x)(1 + ||x|| + \ell_{\max}(x))$$

The interesting case is when $x = \langle u, v\rangle$. In this case, we have

$$\begin{aligned}|\langle u, v\rangle| \geq |u| + |v| + 4 &\overset{\mathsf{IH}}{\geq} \ell_{\mathrm{lin}}(u) + \ell_{\mathrm{lin}}(v) + \ell_{\max}(u) + \ell_{\max}(v) + 4 \geq\\ &\geq \ell_{\mathrm{lin}}(\langle u, v\rangle) + \ell_{\max}(\langle u, v\rangle)\end{aligned}$$

thus establishing the first inequality. For the second we calculate as follows.

$$
\begin{array}{ll}
& \ell_{\mathrm{lin}}(\langle u,v\rangle)(1+||\langle u,v\rangle||+\ell_{\mathrm{max}}(\langle u,v\rangle) \\
= & (\ell_{\mathrm{lin}}(u)+\ell_{\mathrm{lin}}(v)+4)(1+||\langle u,v\rangle||+\max(\ell_{\mathrm{max}}(u),\ell_{\mathrm{max}}(v))) \\
\geq & \ell_{\mathrm{lin}}(u)(1+||u||+\ell_{\mathrm{max}}(u))+\ell_{\mathrm{lin}}(v)(1+||v||+\ell_{\mathrm{max}}(v))+4(1+||\langle u,v\rangle||) \\
\stackrel{\mathsf{IH}}{\geq} & |u|+|v|+2||v||+4 \\
= & |\langle u,v\rangle|
\end{array}
$$

In the remaining case we have

$$|x| \geq |x| + 0 = \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{max}}(x)$$

and

$$|x| \leq |x|(1+||x||) = \ell_{\mathrm{lin}}(x)(1+||x||+\ell_{\mathrm{max}}(x))$$

□

Corollary 8.3 *There exists a quadratic polynomial p such that $|x| \leq p(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{max}}(x))$.*

Proof. Writing u for $\sqrt{|x|}$ we obtain for

$$u^2 \leq \ell_{\mathrm{lin}}(x)(1+u+\ell_{\mathrm{max}}(x))$$

for x large enough from $||x|| \leq \sqrt{|x|}$. This gives

$$u \leq \ell_{\mathrm{lin}}(x) + \sqrt{\ell_{\mathrm{lin}}(x)^2 + 4\ell_{\mathrm{lin}}(x)(1+\ell_{\mathrm{max}}(x))} \leq \ell_{\mathrm{lin}}(x) + (\ell_{\mathrm{lin}}(x) + 2 + 2\ell_{\mathrm{max}}(x))$$

hence the result by squaring. □

Accordingly, any function computable in time $O(|x|)$ is computable in time $O((\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{max}}(x))^2)$.

We assume an encoding of computations such that for each algorithm e and integer $N \geq \ell_{\mathrm{lin}}(e)$ we can find an algorithm e' such that the runtime of $\{e'\}(x)$ is not greater than $|N|$ plus the time needed to compute $\{e\}(x)$ and such that $\ell_{\mathrm{lin}}(e') \geq N$. Similarly, we assume that we can arbitrarily increase the maximum length of an algorithm. That this is in principle possible hinges on the two injections num and pad.

Let $p > 2$ be a fixed integer.

Definition 8.4 *A computation $\{e\}(x)$ is called* short *(w.r.t. p) if it needs less than $d(\ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x) + \max(\ell_{\mathrm{max}}(e),\ell_{\mathrm{max}}(x)))^p$ steps where $d = \ell_{\mathrm{lin}}(e)+\ell_{\mathrm{lin}}(x)-\ell_{\mathrm{lin}}(\{e\}(x))$ and in addition $\ell_{\mathrm{max}}(\{e\}(x)) \leq d + \max(\ell_{\mathrm{max}}(e),\ell_{\mathrm{max}}(x))$. The difference d between $\ell_{\mathrm{lin}}(e)+\ell_{\mathrm{lin}}(x)$ and $\ell_{\mathrm{lin}}(\{e\}x)$ is again called the defect of the computation $\{e\}(x)$.*

An algorithm e is called short if $\{e\}(x)$ is short for all x.

For what follows it is useful to recall the following basic rules of "maxplus-arithmetic".

- $x + \max(y,z) = \max(x+y, x+z)$
- more generally, $f(\max(x,y)) = \max(f(x),f(y))$ whenever f is monotone.
- $\max(x,y) \leq z \iff x \leq z \wedge y \leq z$.
- $\max(x,y) \leq x+y$.
- $\max(x, y+z) \leq y + \max(x,z)$.

The following shows that maximum-bounded number-theoretic functions are computable by short algorithms.

Lemma 8.5 *If $f : \mathbb{N}^n \to \mathbb{N}$ is computable in time $O(\max(|x_1|,\ldots,|x_n|)^p)$ and $|f(x_1,\ldots,x_n)| = \max(|x_1|,\ldots,|x_n|)+O(1)$ then there exists a short algorithm e such that*

$$f(x_1,\ldots,x_n) = \{e\}(\langle \mathsf{num}(x_1), \langle \mathsf{num}(x_2),\ldots,\mathsf{num}(x_n)\rangle \ldots\rangle)$$

Proof. Immediate from the definitions. □

Proposition 8.6 *There exists a function* $\mathsf{app} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ *and a constant* γ *such that*

- $\mathsf{app}(e, x)$ *is computable in time* $d(\ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x) + \gamma + \max(\ell_{\max}(e), \ell_{\max}(x)))^p$ *where* $d = \ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x) - \ell_{\mathrm{lin}}(\mathsf{app}(e, x))$.
- $\ell_{\max}(\mathsf{app}(e, x)) \leq d + \max(\ell_{\max}(e), \ell_{\max}(x)))$
- $\mathsf{app}(e, x) = \{e\}(x)$ *for every short computation* $\{e\}(x)$.

Proof. Given e, x we simulate $\{e\}(x)$ for at most $(\ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x))T$ steps where $T = (\ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x) + \max(\ell_{\max}(e), \ell_{\max}(x)))^p$. If the computation has halted by then we compute the defect $d = \ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x) - \ell_{\mathrm{lin}}(\{e\}(x))$ and see whether the actual running time was smaller than dT and moreover, $\ell_{\max}(\{e\}(x)) \leq d + \max(\ell_{\max}(e), \ell_{\max}(x))$. If yes, we forward the result, i.e., we put $\mathsf{app}(e, x) = \{e\}(x)$. Otherwise we put $\mathsf{app}(e, x) = 0$.

The verification runs analogous to the one in Prop. 4.7. □

Again, we will abbreviate $\mathsf{app}(e, x)$ by $e \cdot x$ or ex.

Lemma 8.7 (Parametrisation) *For every* e *there exists an algorithm* e' *such that* $e\langle x, y\rangle = e'xy$.

Proof. Let e_0 be the algorithm which on input x returns the *algorithm* which on input y returns $\mathsf{app}(e, \langle x, y\rangle)$.

Now, $\ell_{\mathrm{lin}}(\{e_0\}(x)) \leq \ell_{\mathrm{lin}}(e) + \ell_{\mathrm{lin}}(x) + c$ and $\ell_{\max}(\{e_0\}(x)) \leq \max(\ell_{\max}(e), \ell_{\max}(x))$ where c is some fixed constant.

Sufficient padding of e_0 thus yields the desired algorithm e'. □

Theorem 8.8 *The set of natural numbers together with the above application function* app *is a BCK-algebra.*

Proof. The K-combinator is straightforward. For the composition combinator B we look again at the algorithm B_0 from the proof of Theorem 4.9 given by

$$\{B_0\}(w) = \mathsf{app}(w.1.1, \mathsf{app}(w.1.2, w.2))$$

We have $\{B_0\}(\langle\langle x, y\rangle, z\rangle) = x(yz)$ and the time t_{tot} needed to perform this computation is less than $t_1 + t_2 + t_b$ where

$$\begin{aligned}
u &= yz \\
w &= x(yz) \\
d_1 &= \ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) - \ell_{\mathrm{lin}}(u) \\
d_2 &= \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(u) - \ell_{\mathrm{lin}}(w) \\
t_1 &= d_1(\ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) + \gamma + \max(\ell_{\max}(y), \ell_{\max}(z)))^p \\
t_2 &= d_2(\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(u) + \gamma + \max(\ell_{\max}(x), \ell_{\max}(u)))^p
\end{aligned}$$

and where t_b—the time needed for shuffling around intermediate results—is linear in $|x| + |y| + |z| + |u| + |w|$ and thus quadratic in $\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) + \ell_{\mathrm{lin}}(z) + \max(\ell_{\max}(x), \ell_{\max}(y), \ell_{\max}(z))$. Now

$$\ell_{\max}(u) \leq d_1 + \max(\ell_{\max}(y), \ell_{\max}(z))$$

So,

$$\begin{array}{ll} t_2 \leq & d_2(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(u)+\gamma+ \\ & \quad \max(\ell_{\max}(x),(\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(u))+\max(\ell_{\max}(y),\ell_{\max}(z))))^p \\ \leq & d_2(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\gamma+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))^p \end{array}$$

This means that we can find a constant c such that

$$t_{\mathsf{tot}} \leq (d_1+d_2)(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+c+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))^p$$

Now the defect of the computation $\{B_0\}(\langle\langle x,y\rangle,z\rangle)$ equals $\ell_{\mathrm{lin}}(B_0)+4+\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(w)=\ell_{\mathrm{lin}}(B_0)+4+d_1+d_2$.

Moreover,

$$\begin{array}{ll} & \ell_{\max}(w) \\ \leq & d_2+\max(\ell_{\max}(x),\ell_{\max}(u)) \\ \leq & d_2+\max(\ell_{\max}(x),d_1+\max(\ell_{\max}(y),\ell_{\max}(z))) \\ \leq & d_2+d_1+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)) \end{array}$$

Therefore, by choosing $\ell_{\mathrm{lin}}(B_0)$ large enough we obtain

$$\mathsf{app}(B_0,\langle\langle x,y\rangle,z\rangle)=\{B_0\}(\langle\langle x,y\rangle,z\rangle)=x(yz)$$

The desired algorithm B is then obtained by applying Lemma 8.7 twice.

Notice that the coupling of defect and maximum length is essential for the definability of composition.

We come to the twisting combinator C. Again, we start with the algorithm C_0 defined by

$$\{C_0\}(w)=\mathsf{app}(\mathsf{app}(w.1.1,w.2),w.1.2)$$

Clearly,

$$\{C_0\}(\langle\langle x,y\rangle,z\rangle)=xzy$$

The total time t_{tot} needed for this computation is bounded by $t_1+t_2+t_b$ where

$$\begin{array}{l} u=xz \\ w=uy \\ d_1=\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(u) \\ d_2=\ell_{\mathrm{lin}}(u)+\ell_{\mathrm{lin}}(y)-\ell_{\mathrm{lin}}(w) \\ t_1=d_1(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(z)+\gamma+\max(\ell_{\max}(x),\ell_{\max}(z))^p) \\ t_2=d_2(\ell_{\mathrm{lin}}(u)+\ell_{\mathrm{lin}}(y)+\gamma+\max(\ell_{\max}(u),\ell_{\max}(y))^p) \end{array}$$

and the time t_b needed for administration is

$$O((\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))^2)$$

Now,

$$\begin{array}{ll} t_2 \leq & d_2(\ell_{\mathrm{lin}}(u)+\ell_{\mathrm{lin}}(y)+\gamma+ \\ & \quad \max(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(u)+\max(\ell_{\max}(x),\ell_{\max}(z)),\ell_{\max}(y)))^p \\ \leq & d_2(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))^p \end{array}$$

and we can find a constant c such that

$$t_{\mathsf{tot}} \leq (d_1+d_2)(\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)+\max(\ell_{\max}(x),\ell_{\max}(y),\ell_{\max}(z)))$$

The defect of the computation $\{C_0\}(\langle\langle x,y\rangle,z\rangle)$ is

$$d=\ell_{\mathrm{lin}}(C_0)+8+\ell_{\mathrm{lin}}(x)+\ell_{\mathrm{lin}}(y)+\ell_{\mathrm{lin}}(z)-\ell_{\mathrm{lin}}(w)=\ell_{\mathrm{lin}}(C_0)+8+d_1+d_2$$

Therefore, by choosing $\ell_{\mathrm{lin}}(C_0)$ large enough we obtain the desired combinator as in the proof of Theorem 4.9. □

The thus obtained *BCK*-algebra will henceforth be called M_p. The next theorem shows that all the previous machinery about $\mathcal{H}^\Box$ can be applied to M_p as well.

Theorem 8.9 *The BCK-algebra M_p is polynomial with respect to the length measure $\ell(x) = \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{max}}(x)$.*

Proof. It is clear that xy is computable in time polynomial in $\ell(x) + \ell(y)$. For the estimate $\ell(xy) \leq \ell(x) + \ell(y)$ we calculate as follows: If $x, y \in M_p$ then $\ell_{\mathrm{lin}}(xy) \leq \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y)$ and $\ell_{\mathrm{max}}(xy) \leq d + \max(\ell_{\mathrm{max}}(x), \ell_{\mathrm{max}}(y)$ where $d = \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) - \ell_{\mathrm{lin}}(xy)$. Thus, $\ell(xy) = \ell_{\mathrm{lin}}(xy) + \ell_{\mathrm{max}}(xy) \leq \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(y) - d + d + \max(\ell_{\mathrm{max}}(x), \ell_{\mathrm{max}}(y)) \leq \ell(x) + \ell(y)$.

The other estimates involving $|-|$ are direct from the definition. □

The following shows that in the category of M_p-sets natural numbers are duplicable.

Proposition 8.10 *Let A be an M_p-set such that there exists $k \in \mathbb{N}$ with $\ell_{\mathrm{lin}}(x) \leq k$ for all $x \in \mathrm{dom}(A)$. Then A is duplicable, i.e., the diagonal function $\delta_A : A \longrightarrow A \otimes A$ defined by $\delta_A(x) = (x, x)$ is realisable.*

Proof. Consider the algorithm dup given by

$$\{\mathsf{dup}\}(x) = \lambda f.f\ x\ x$$

It is clear that *if* this algorithm is short then it is a realiser for the diagonal function $\delta : \mathsf{N} \longrightarrow \mathsf{N} \otimes \mathsf{N}$. Its running time is linear thus at most quadratic in $\ell_{\mathrm{lin}}(x) + \ell_{\mathrm{max}}(x)$. In view of the characterisation in Lemma 8.5 it thus remains to show that it meets the growth restrictions

$$\ell_{\mathrm{lin}}(\mathsf{dup}(x)) = \ell_{\mathrm{lin}}(x) + O(1)$$
$$\ell_{\mathrm{max}}(\mathsf{dup}(x)) = \ell_{\mathrm{max}}(x) + O(1)$$

If $x \in \mathrm{dom}(A)$ then $\ell_{\mathrm{lin}}(x) \leq k$ by assumption, so

$$\ell_{\mathrm{lin}}(\mathsf{dup}(x)) = \ell_{\mathrm{lin}}(\lambda f.fxx) = \ell_{\mathrm{lin}}(x) + \ell_{\mathrm{lin}}(x) + O(1) = O(1)$$

Next,

$$\ell_{\mathrm{max}}(\mathrm{dup}(x)) = \ell_{\mathrm{max}}(\lambda f.fxx) = \max(\ell_{\mathrm{max}}(x), \ell_{\mathrm{max}}(x)) + O(1) = \ell_{\mathrm{max}}(x) + O(1)$$

□

Now since N is duplicable in $\mathcal{H}^\Box$, we have $\mathsf{N} = !\mathsf{N}$ in the category $\mathrm{Ext}(\mathcal{H}^\Box)$ constructed over $H = M_p$ thus providing an interpretation of SLR with the type equality $\mathsf{N} \to A = \mathsf{N} \multimap A$ and hence a soundness proof for the latter system.

9 Computational interpretation

The interpretation in $\mathrm{Ext}(\mathcal{H}^\Box)$ although maybe complicated is entirely constructive and can be formalised for instance in extensional Martin-Löf type theory with quotient types, see [8]. Such formalisation gives rise to an algorithm which computes a polynomial time algorithm for every term of type $\Box\mathsf{N} \to \mathsf{N}$. The efficiency of such compilation algorithm and, more importantly, of the produced polynomial time algorithms hinges on the form of the soundness proof. A detailed analysis falls outside the scope of this work and must await further investigation.

We would like at this point, however, comment on the apparent overhead involved with the runtime monitoring in the application function of the *BCK*-algebras involved. Intuitively, explicit runtime bounds could be omitted since we know anyway that all well-typed programs terminate within the allocated time. However, in doing so we change the behaviour of those programs which contain a subcomputation an application which returns zero because of runtime exhaustion. In principle this might affect the results; in order to show that this is not so one can replace the category $\mathcal{H}$ by a category in which runtime bounds are built into the definition of morphisms.

Definition 9.1 *Let $p > 2$. The category $\mathcal{H}'$ has as objects pairs $X = (|X|, \Vdash_X)$ where $|X|$ is a set and $\Vdash \subseteq \mathbb{N} \times |X|$ is a surjective relation. A morphism from X to Y is a function $f : |X| \longrightarrow |Y|$ such that there exists an algorithm e with the property that whenever $t \Vdash_X x$ then $\{e\}(x)$ terminates with a result y such that $y \Vdash_Y f(x)$ and moreover the runtime of $\{e\}(x)$ is less than $(\ell(e) + \ell(x) - \ell(y))(\ell(e) + \ell(x))^p$.*

An analogous definition can be given relative to M_p.

One can now show that $\mathcal{H}'$ has essentially the same category-theoretic properties as $\mathcal{H}$, but no explicit runtime computations appear in $\mathcal{H}$. The big disadvantage of $\mathcal{H}'$ as opposed to $\mathcal{H}$ is that we have to carry out calculations on the level of algorithms for every single construction in $\mathcal{H}'$, whereas with $\mathcal{H}$ these calculations can be concentrated in the proof that H_p forms a polynomial time BCK-algebra. After that all the verifications can be carried out on the higher level of abstraction given by untyped linear lambda calculus.

References

[1] S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Ramification, Modality, and Linearity in Higher Type Recursion. this volume, 1999.

[2] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the poly-time 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] Peter Clote. Computation models and function algebras. available electronically under http://thelonius.tcs.informatik.uni-muenchen.de/∼clote/Survey.ps.gz, 1996.

[5] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology, and Philosophy of Science II*, pages 24–30. Springer Verlag, 1965.

[6] Brian J. Day. An embedding theorem for closed categories. In *Category Seminar Sydney 1972–73, LNM 420.* Springer Verlag, 1974.

[7] Martin Hofmann. An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras. *Bulletin of Symbolic Logic*, 3(4):469–485, 1997.

[8] Martin Hofmann. *Extensional constructs in intensional type theory.* BCS distinguished dissertation series. Springer, 1997. 214 pp.

[9] Martin Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

[10] Martin Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *Proceedings of CSL '97, Aarhus. Springer LNCS 1414*, pages 275–294, 1998.

[11] Martin Hofmann. Semantics of linear/modal lambda calculus. To appear in *Journal of Functional Programming*, 1998.

[12] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*, pages 204–213. IEEE, Computer Society Press, 1999.

[13] Martin Hofmann. Type systems for polynomial-time computation, 1999. Habilitation thesis, TU Darmstadt, Germany. Edinburgh University LFCS Technical Report, ECS-LFCS-99-406.

[14] Joachim Lambek and Philip Scott. *Introduction to Higher-Order Categorical Logic.* Cambridge University Press, 1986.

[15] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterisations of polytime. *Fundamentae Informaticae*, 19:167–184, 1993.

[16] 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.

[17] Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity IV: Predicative functionals and Poly-space. Manuscript, 1997.

[18] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1971.

[19] Frank Pfenning and Hao-Chi Wong. On a modal lambda calculus for S4. In *Proceedings of the 11th Conference on Mathematical Foundations of Programming Semantics (MFPS), New Orleans, Louisiana.* Electronic Notes in Theoretical Computer Science, Volume 1, Elsevier, 1995.

[20] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.