Semantics of linear/modal lambda calculus

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. While this previous work was concerned with the syntactic metatheory of SLR in this paper we develop a semantics of SLR in terms of Chu spaces over a certain category of sheaves from which it follows that all expressible functions are indeed in *PTIME*. We notice a similarity between the Chu space interpretation and CPS translation which as we hope will have further applications in functional programming.

1 Introduction

In [10] we have introduced a lambda calculus SLR which generalises the Bellantoni-Cook characterisation of *PTIME* [4] 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.* the syntactic properties of SLR were studied this paper is devoted to a semantic analysis of its strength. Notably, we prove that linear recursion with functional result type does not lead beyond polynomial time. This is done by interpreting SLR in a certain model of linear logic, namely a variant of so-called Chu spaces [20, 13], in which all type 1 functions are by definition in *PTIME*. The desired result then follows by relating this interpretation to the standard set-theoretic one by means of a certain logical relation.

The work described here is part of a research programme aimed at extending the Bellantoni-Cook system and similar function algebra characterisations (see [6] for a survey) to higher-order typed ("HOT") functional programming languages. Section 9 contains a discussion of related work.

1.1 The system BC

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 [7] 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

*Laboratory for Foundations of Computer Science, University of Edinburgh, JCMB, KB, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

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:

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

$$\begin{aligned} f(\vec{x},0;\vec{y}) &= g(\vec{x};\vec{y}) \\ f(\vec{x},x;\vec{y}) &= h(\vec{x},x;\vec{y},f(\vec{x},\left\lfloor \tfrac{x}{2}\right\rfloor;\vec{y})),\ \text{if } x>0 \end{aligned}$$

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 [4] 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

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

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

$$\begin{aligned} \mathsf{sq}(0;) &= 1 \\ \mathsf{sq}(x;) &= \mathsf{S}_0(\mathsf{S}_0(\mathsf{sq}(\left\lfloor \tfrac{x}{2}\right\rfloor;))) \end{aligned}$$

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

$$\begin{aligned} \mathsf{exp}(0;) &= 1 \\ \mathsf{exp}(x;) &= \mathsf{sq}(\mathsf{exp}(\left\lfloor \tfrac{x}{2}\right\rfloor;)) \end{aligned}$$

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.

It is the purpose of the work described in this paper and in [10] to extend this framework to higher types, i.e., functions of arbitrary functional type. Unlike Bellantoni-Cook's system BC^ω [2] which merely is a simply-typed lambda calculus with two base types, we aim at a faithful generalisation of the pattern of safe composition to higher types.

[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.

1.2 Modal types

It turns out that the right way to do this is by using a modal operator $\Box$ on types with the understanding that $\Box A$ is the type of *normal* objects of type A. In particular, $\Box\mathsf{N}$ is the type of normal integers whereas N itself is the type of safe integers. Since normal values may always be used in place of safe ones there is a coercion function $\mathsf{unbox}_A : \Box A \to A$.

Safe recursion can then be formulated as a single higher-typed constant called a *recursor*.

$$\mathsf{saferec} : \Box\mathsf{N} \to \mathsf{N} \to (\Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N}$$

where $f(x) = \mathsf{saferec}(x, g, h)$ means $f(0) = g$ and $f(x) = h(x, f(\lfloor \frac{x}{2} \rfloor))$ when $x > 0$. We see that the second (recursive) argument of h is of type N so that h cannot recur over this argument. Notice that in the presence of lambda calculus parameters need not be explicitly mentioned in the type of the recursor. E.g., if $g : A \to \mathsf{N}$ and $h : A \to \Box\mathsf{N} \to \mathsf{N}$ then we can define $f : A \to \Box\mathsf{N} \to \mathsf{N}$ by $\lambda a{:}\, A.\lambda x{:}\, \Box\mathsf{N}.\mathsf{saferec}(x, g(a), h(a))$.

How can we ever create an object of type $\Box\mathsf{N}$ in order to apply $\mathsf{saferec}$? The idea is that if an expression $t : \mathsf{N}$ contains only free variables of modal type, i.e., of type $\Box A$ for some A then we should be allowed to form a term $\mathsf{raise}(t)$ of type $\Box\mathsf{N}$. This corresponds to the rule of necessitation in modal logic.

Such new term formers unbox and raise make terms less readable and complicate programming. The system described in [10] makes unbox an implicit subtype coercion and avoids raise by restricting $\Box$ to argument positions of function types. In other words, rather than having a type operator $\Box$ we have two function spaces $A \to B$ (ordinary function space) and $\Box A \to B$ (modal function space). The free variables of a term are still grouped into two zones: modal and nonmodal ones. In order to apply a function $f : \Box A \to B$ to a term $t : A$ one must check that t depends on modal variables only. In function abstraction the "aspect" of a variable need not be given explicitly, e.g., if $g : \mathsf{N}$ and $h : \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ we can write $f = \lambda x{:}\, \mathsf{N}.\mathsf{saferec}(x, g, h)$ and the type $\Box\mathsf{N} \to \mathsf{N}$ will be inferred for f. More generally, if

$$f = \lambda u{:}\, \mathsf{N} \to \mathsf{N}.\lambda x{:}\, \mathsf{N}.\mathsf{saferec}(u(x), g, h)$$

then f has the type $\Box(\mathsf{N} \to \mathsf{N}) \to \Box\mathsf{N} \to \mathsf{N}$. The "type" $\Box(\mathsf{N} \to \mathsf{N})$ refers to a function which does not contain any free variables of nonmodal type. For example, we can apply this f to S_0, but not to $\lambda x{:}\, \mathsf{N}.y$, unless y is a modal variable.

1.3 Linear recursion

The system described in [10] additionally contains *linear* function types which contain functions using their argument at most once (as far as this can be detected syntactically). The purpose of these linear types is that they allow one to formulate a restricted version of safe recursion with *functional* result type which is conservative, i.e., does not lead beyond *PTIME*.

In general, if we allow functional result type in a safe recursion then we can define exponentiation.

Example 1.1 *We have*

$$\begin{aligned} g &= \mathsf{S}_0 : \mathsf{N} \to \mathsf{N} \\ h &= \lambda x{:}\, \mathsf{N}.\lambda u{:}\, \mathsf{N} \to \mathsf{N}.(\lambda y{:}\, \mathsf{N}.u(u(y))) : \Box\mathsf{N} \to (\mathsf{N} \to \mathsf{N}) \to (\mathsf{N} \to \mathsf{N}) \end{aligned}$$

and the function $f : \Box\mathsf{N} \to \mathsf{N} \to \mathsf{N}$ *defined by safe recursion with result type* $\mathsf{N} \to \mathsf{N}$, *i.e.,* $f(0) = g$, $f(x) = h(x, f(\lfloor \frac{x}{2} \rfloor))$ *has exponential growth rate, namely we have* $f(x, y) = 2^{[x]} \cdot y$.

The reason for this growth is that u is called twice in the body of h. In order to rule this out syntactically, we introduce a linear recursor

$$\mathsf{linrec}_A : \Box\mathsf{N}{\to}A{\to}(\Box\mathsf{N}{\to}A{\multimap}A){\to}A$$

for $A = \mathsf{N}^k{\to}\mathsf{N}$. Here $\multimap$ denotes the linear function space.

Informally, a function(al) is linear if it uses its argument at most once. This is obviously not the case for the functional $\lambda x{:}\,\mathsf{N}.\lambda u{:}\,\mathsf{N}{\to}\mathsf{N}.\lambda y{:}\,\mathsf{N}.u(u\ y)$ appearing in the definition in Example 1.1 above. It has the type $\Box\mathsf{N}{\to}(\mathsf{N}{\to}\mathsf{N}){\to}(\mathsf{N}{\to}\mathsf{N})$ rather than $\Box\mathsf{N}{\to}(\mathsf{N}{\to}\mathsf{N}){\multimap}(\mathsf{N}{\to}\mathsf{N})$ as required by linrec and so this function cannot be defined using linrec.

Its type-theoretic formulation requires a similar setup as we already have for the modal function space. The variables are now associated with one out of four "aspects" where an aspect is a pair (l, m) where $l \in \{\text{linear}, \text{nonlinear}\}$ and $m \in \{\text{modal}, \text{nonmodal}\}$.

A variable of linear aspect can appear at most once in a term (see below for two exceptions to this rule). If a variable appears more than once then it must be given nonlinear aspect. A term $e : B$ containing free variable $x : A$ can be given type $A{\multimap}B$ if $e : B$ can be derived with x having linear aspect. If this is not possible then the weaker type $A{\to}B$ must be assigned. For example, we have

$$\begin{array}{l}
\lambda x{:}\,A.x : A{\multimap}A \\
\lambda f{:}\,A{\to}B.\lambda x{:}\,A.f\ x : (A{\to}B){\multimap}A{\to}B \\
\lambda f{:}\,A{\multimap}A.\lambda x{:}\,A.f(f(x)) : (A{\multimap}A){\to}A{\multimap}A \\
\lambda f{:}\,A{\to}A.\lambda g{:}\,A{\to}A.\lambda x{:}\,A.f(g\ x) \\
\qquad : (A{\to}A){\multimap}(A{\to}A){\to}A{\to}A
\end{array}$$

In the last example g is dubbed nonlinear although it appears only once. The reason is of course that it appears as the argument of a nonlinear function.

Apart from counting the number of occurrences there are two further sources for linearity brought about by the base type N. First, a variable of type N is linear by definition. This is expressed by a subtyping $\mathsf{N}{\to}A \leq \mathsf{N}{\multimap}A$. (We always have the subtyping $A{\multimap}B \leq A{\to}B$.).

Second, we have the following case construct for integers. If $e_1 : \mathsf{N}$ and $e_2 : A$ and $e_3, e_4 : \mathsf{N}{\to}A$ then we have $\mathsf{case}_A e_1\ \mathsf{zero}\ e_2\ \mathsf{even}\ e_3\ \mathsf{odd}\ e_4 : A$ with the intended meaning that if e_1 is zero then the result is e_2, if e_1 is $2n$ then the result is $e_3\ n$ and if e_1 is $2n{+}1$ then the result is $e_4\ n$. The important point is that if a variable appears linearly in each branch of a case construct then it appears linearly in the whole case expression although it makes up to three literal occurrences. We illustrate this by giving more terms with their (principal) types.

$$\begin{array}{l}
\mathsf{divtwo} = \lambda x{:}\,\mathsf{N}.\mathsf{case}_{\mathsf{N}} x \\
\qquad \mathsf{zero}\ 0 \\
\qquad \mathsf{even}\ \lambda y{:}\,\mathsf{N}.y \\
\qquad \mathsf{odd}\ \lambda y{:}\,\mathsf{N}.y : \mathsf{N}{\to}\mathsf{N}
\end{array}$$

$$\begin{array}{l}
\lambda x{:}\,\mathsf{N}.\lambda f{:}\,\mathsf{N}{\to}\mathsf{N}.\mathsf{case}_{\mathsf{N}} x \\
\qquad \mathsf{zero}\ f(x) \\
\qquad \mathsf{even}\ f \\
\qquad \mathsf{odd}\ f : \mathsf{N}{\to}(\mathsf{N}{\to}\mathsf{N}){\multimap}\mathsf{N}
\end{array}$$

To illustrate the fact that functions on integers are always linear we notice that the last function of the previous suite gets type $(A{\to}A){\multimap}(A{\to}A){\multimap}(A{\to}A)$ in case $A = \mathsf{N}$ for then f has in fact type $\mathsf{N}{\multimap}\mathsf{N}$.

We also notice that by this convention the first-order safe recursor $\mathsf{saferec}$ from above is subsumed under the linear recursor linrec_A for $A = \mathsf{N}$, i.e., $k = 0$.

In [10] we have used $\mathsf{linrec}_{\mathsf{N}^3\to\mathsf{N}}$ to define an addition function $\mathsf{add} : \Box\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}$ with the specification that $\mathsf{add}\ l\ x\ y\ c$ equals $\hat{x}+\hat{y}+(c \bmod 2)$ where $\hat{x} = x \bmod 2^{|l|}$. This is not an entirely trivial task because on the binary integers addition must be defined using the algorithm for digit-wise addition with carry. In particular, a naive attempt to define addition with ordinary safe recursion fails because in the recursive call the carry bit may change.

$$
\begin{array}{l}
\mathsf{add} = \lambda l\colon \mathsf{N}.\\
\quad \mathsf{linrec}_{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}}\ l\\
\quad\quad (\lambda x\colon \mathsf{N}.\lambda y\colon \mathsf{N}.\lambda c\colon \mathsf{N}.c \bmod 2)\\
\quad\quad (\lambda u\colon \mathsf{N}.\lambda a\colon \mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathsf{N}.\lambda x\colon \mathsf{N}.\lambda y\colon \mathsf{N}.\lambda c\colon \mathsf{N}.\\
\quad\quad\quad \mathsf{let\,carry} = (x \wedge (y \vee c)) \vee ((\neg x) \wedge (y \wedge c))\ \mathsf{in}\\
\quad\quad\quad\quad \mathsf{case}_{\mathsf{N}} x \oplus y \oplus c\\
\quad\quad\quad\quad\quad \mathsf{zero}\ \mathsf{S}_0(a\,\lfloor x/2\rfloor\,\lfloor y/2\rfloor\ \mathit{carry})\\
\quad\quad\quad\quad\quad \mathsf{even}\ \lambda u\colon \mathsf{N}.\mathsf{S}_0(a\,\lfloor x/2\rfloor\,\lfloor y/2\rfloor\ \mathit{carry})\\
\quad\quad\quad\quad\quad \mathsf{odd}\ \lambda u\colon \mathsf{N}.\mathsf{S}_0(a\,\lfloor x/2\rfloor\,\lfloor y/2\rfloor\ \mathit{carry}))
\end{array}
$$

Here $\mathsf{let}\ldots\mathsf{in}$ is syntactic sugar for a β-expansion and the auxiliary functions $\vee, \wedge, \ldots$ are the indicated boolean functions of type $N\to N$ and $N\to N\to N$ defined using case_A. They only look at the last bit of a number. Finally, $\lfloor -/2\rfloor$ and $- \bmod 2$ are quotient and remainder with respect to division by two also defined using $\mathsf{case}_{\mathsf{N}}$.

Notice that although the function a makes two literal appearances in the body of the third argument to linrec the latter is still counted as a linear functional because the appearances belong to different branches of a case construct.

We see from this example that the main purpose of linear recursion is to provide a user-friendly syntax for first-order recursion with substitution of parameters. If we were to use first-order recursion with substitution we would have to put together the substitution functions using another case distinction.

Indeed, we will semantically reduce linear recursion to recursion with parameter substitution which is known not to lead beyond polynomial time. If we would provide an operator for recursion with parameter substitution then we would have to extract the substituting functions and put them into a separate position thus leading to a much less readable syntax.

Linear recursion with second-order result type such as $(\mathsf{N}\to\mathsf{N})\to\mathsf{N}$ also leads beyond *PTIME* as can be seen from the following example:

Example 1.2 *Let A be $(\mathsf{N}\to\mathsf{N})\to\mathsf{N}$ and define $g : A$ by $\lambda u\colon \mathsf{N}\to\mathsf{N}.u\ 1$ and $h : \Box\mathsf{N}\to A\multimap A$ by*

$$\lambda x\colon \mathsf{N}.\lambda F\colon A.\lambda u\colon \mathsf{N}\to\mathsf{N}.F(\lambda z\colon \mathsf{N}.u(u(z))))$$

Here we have $f(x)(u) = u^{([x])}(1)$ as can be seen by notational induction on x; hence $f(x)(\mathsf{S}_0) = 2^{[x]}$.

Then $f : \Box\mathsf{N}\to A$ defined by $f(0) = g$ and $f(x) = h(x, f(\lfloor \frac{x}{2}\rfloor))$ satisfies $f(x)(u) = u^{([x])}(1)$ as can be seen by notational induction on x; hence $f(x)(\mathsf{S}_0) = 2^{2^{|x|}}$.

Notice that, again, a nonlinear use of a functional argument was the culprit. Indeed, we believe that linear recursion with result types $(\mathsf{N}\to\mathsf{N})\multimap\mathsf{N}$ stays within *PTIME*, but the methods described in this paper do not allow us to prove this.

The following example shows that linear recursion with result type $\Box\mathsf{N}\to\mathsf{N}$ must also be forbidden:

Example 1.3 *Let $A = \Box\mathsf{N}\to\mathsf{N}$ and define $g : A$ by $\lambda y\colon \mathsf{N}.y$ and $h : \Box\mathsf{N}\to A\multimap A$ by $\lambda x\colon \mathsf{N}.\lambda u\colon A.\lambda y\colon \mathsf{N}.u(\mathsf{sq}(y))$ where $\mathsf{sq} : \Box\mathsf{N}\to\mathsf{N}$ is given by $\mathsf{sq}(x) = [x]^2$. Then $f : \Box\mathsf{N}\to A$ defined by $f(0) = g$ and $f(x) = h(x, f(\lfloor \frac{x}{2}\rfloor))$ satisfies $f(x)(y) = y^{2^{|x|}}$.*

1.4 Semantics of SLR

Bellantoni-Cook's proof that the system BC defines *PTIME*-functions is based on the invariant that if $f(\vec{x};\vec{y})$ is definable then $f(\vec{x};\vec{y})$ is in *PTIME* and moreover $|f(\vec{x};\vec{y})| \leq p(|\vec{x}|) + \max(|\vec{y}|)$ for some n-place polynomial p.

In [9] we show how to lift this invariant to an invariant of the purely modal fragment of SLR by using presheaves over the category of the *PTIME*-functions which satisfy Bellantoni and Cook's growth restrictions. In this paper we extend this method to the linear recursor with first-order result type. The main idea is to use an interpretation in which a linear functional F of type $(\mathsf{N}{\to}\mathsf{N}){\multimap}\mathsf{N}$ is modelled as an element $\mathsf{arg} : \mathsf{N}$ and a function $\mathsf{rest} : \mathsf{N}{\to}\mathsf{N}$ such that $F(u) = \mathsf{rest}(u(\mathsf{arg}))$. In this way, semantically, the linear recursion can be reduced to recursion with ground result type and substitution of parameters.

The advantage of using linear recursion rather than recursion with parameter substitution right away is a pragmatic one. A recursor with substitution of parameters

$$\mathsf{substrec} : \Box\mathsf{N}{\to}(\mathsf{N}{\to}\mathsf{N}){\to}(\Box\mathsf{N}{\to}\mathsf{N}{\to}\mathsf{N}{\to}\mathsf{N}){\to}(\Box\mathsf{N}{\to}\mathsf{N}{\to}\mathsf{N}){\to}\mathsf{N}{\to}\mathsf{N}$$

with intended semantics

$$\begin{aligned} &f(x,y) = \mathsf{substrec}\,(x,g,h,u,y) \iff \\ &\quad f(0,y) = g(y) \\ &\quad f(x,y) = h(x, f(\left\lfloor \tfrac{x}{2} \right\rfloor, u(x,y))) \end{aligned}$$

requires the user to isolate the substituting function u from the defining equations for f. This requires code duplication if a case distinction is part of the definition of f like in the addition example below. Furthermore, it leads to less readable programs as the definition of f is split among the three components g, h, u.

So linear recursion—like any high-level programming construct—leads to shorter and more intuitive programs.

2 Syntax

The basic type of SLR is N—the type of binary natural numbers. If A, B are types so are $A{\to}B$ (function space), $\Box A{\to}B$ (modal function space), $A{\multimap}B$ (linear function space). In order to simplify the rules we use a generic notation $A \xrightarrow{a} B$ where $a \in \{\text{nonlinear}, \text{linear}\} \times \{\text{modal}, \text{nonmodal}\} \setminus \{\text{linear}, \text{modal}\}$ for any of the three function spaces. Such a pair a is called an *aspect*. If, e.g., $a = (\text{modal}, \text{nonlinear})$ then $A \xrightarrow{a} B$ means $\Box A{\to}B$.

The aspects are ordered componentwise by "nonlinear" $\leq$ "linear" and "modal" $\leq$ "nonmodal". Notice that we do not use the fourth theoretically possible aspect (linear, nonmodal) which would correspond to "linear, modal"-functions. We assume that every modal function is automatically nonlinear.

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

$$\frac{A' \leq A \qquad B \leq B' \qquad a' \leq a}{A \xrightarrow{a} B \leq A' \xrightarrow{a'} B'}$$

$$\mathsf{N}{\to}A \leq \mathsf{N}{\multimap}A$$

Notice the contravariance of the first rule w.r.t. the ordering of aspects so that, e.g., $A{\multimap}B \leq A{\to}B$ and $A{\to}B \leq \Box A{\to}B$. Types A and B are called equivalent if $A \leq B$ and $B \leq A$. Notice that $\mathsf{N}{\multimap}A$ and $\mathsf{N}{\to}A$ are equivalent. In [10] we also had an equivalence between $\Box\mathsf{N}\multimap A$ and $\Box\mathsf{N}\to A$, but since it does not seem to have any applications and is rather difficult to model we prefer to omit it. In fact, in our model the type $\Box\mathsf{N}\multimap\mathsf{N}$ contains those functions of type $\Box\mathsf{N}\to\mathsf{N}$ which arise from a function of type $\mathsf{N}\to\mathsf{N}$ by subsumption.

We emphasize, however, that although $\Box\mathsf{N}\to\mathsf{N}$ and $\Box\mathsf{N}\multimap\mathsf{N}$ are kept distinct, linearity is primarily a higher-order concept used to express that a higher-order function evaluates its functional argument at at most one point.

The types $\mathsf{N}^k\to\mathsf{N}$ are defined by $\mathsf{N}^0\to\mathsf{N} =_{\mathrm{def}} \mathsf{N}$ and $\mathsf{N}^{k+1}\to\mathsf{N} =_{\mathrm{def}} \mathsf{N}\to\mathsf{N}^k\to\mathsf{N}$.

The expressions of SLR are given by the grammar

$$
\begin{array}{rll}
e ::= & x & \text{(variable)} \\
\mid & (e_1\ e_2) & \text{(application)} \\
\mid & \lambda x{:}\,A.e & \text{(abstraction)} \\
\mid & c & \text{(constants)} \\
\mid & \mathsf{case}_A\, e_1\ \mathsf{zero}\ e_2\ \mathsf{even}\ e_3\ \mathsf{odd}\ e_4 & \text{(case distinction)}
\end{array}
$$

where x ranges over a countable set of variables and c ranges over the set $\mathbb{N}\cup\{\mathsf{S}_0,\mathsf{S}_1,\mathsf{linrec}_A\}$. The type in case_A is arbitrary; the type A in linrec_A must be of the form $\mathsf{N}^k\to\mathsf{N}$. The expressions are identified up to renaming of bound variables.

The type $\tau(c)$ of a constant is given by

$$
\begin{array}{l}
\tau(c) = \mathsf{N}, \text{when } c \text{ is an integer constant} \\
\tau(\mathsf{S}_0) = \tau(\mathsf{S}_1) = \mathsf{N}\to\mathsf{N} \\
\tau(\mathsf{linrec}_A) = \Box N\to A\to(\Box N\to A\multimap A)\to A
\end{array}
$$

A *type assignment* 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 type assignment 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 type assignment Γ is nonlinear if all its bindings are of nonlinear aspect.

Two type assignments 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 type assignments, expressions, and types is defined inductively by the rules in Figure 1. We suppose that all type assignments, 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]. Their syntactic metatheory is easier, however, they are a little more difficult to justify semantically. Our purpose in introducing the truly linear system in *op. cit.* was to demonstrate that the type checking algorithm also works in this more difficult situation. The main result of [10] is that typing is decidable by a syntax-directed procedure in the following sense.

From a type assignment Γ and a term e we can compute a type assignment Δ and a type A such that

i. $\Delta\vdash e : A$

ii. $\mathrm{dom}(\Delta)\subseteq\mathrm{dom}(\Gamma)$ and $\Delta(x)=\Gamma(x)$ for each $x\in\mathrm{dom}(\Delta)$

iii. Whenever $\Theta\vdash e : B$ and $\mathrm{dom}(\Theta)\subseteq\mathrm{dom}(\Gamma)$ and $\Theta(x)=\Gamma(x)$ for each $x\in\mathrm{dom}(\Theta)$ then $A\leq B$ and $\mathrm{dom}(\Delta)\subseteq\mathrm{dom}(\Theta)$ and $\Theta((x))\leq\Delta((x))$ for each $x\in\mathrm{dom}(\Theta)$.

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

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

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

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

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : \mathsf{N} \\ \Gamma, \Delta_2 \vdash e_2 : A \\ \Gamma, \Delta_2 \vdash e_3 : \mathsf{N} \to A \\ \Gamma, \Delta_2 \vdash e_4 : \mathsf{N} \to A \\ \Gamma \text{ nonlinear}\end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{case}_A \, e_1 \ \mathsf{zero} \ e_2 \ \mathsf{even} \ e_3 \ \mathsf{odd} \ e_4 : A} \qquad \text{(T-Case)}$$

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

Figure 1: Typing rules

The last statement means that given the typing of the variables prescribed by Γ the typing $\Delta \vdash e : A$ is optimal in the sense that any other typing yields a weaker type under stronger assumptions.

2.2 Set-theoretic semantics

The calculus SLR has an intended set-theoretic interpretation which in particular associates a function $\mathbb{N} \to \mathbb{N}$ 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.

To each type A we associate a set $[\![A]\!]$ by $[\![\mathsf{N}]\!] = \mathbb{N}$ and $[\![A \xrightarrow{a} B]\!] = [\![A]\!] \to [\![B]\!]$ where $\to$ denotes set-theoretic function space. To each constant c we associate an element $[\![c]\!] \in [\![\tau(c)]\!]$ by

$$\begin{array}{lcl} [\![n]\!] & = & n \\ [\![\mathsf{S}_0]\!](v) & = & 2v \\ [\![\mathsf{S}_1]\!](v) & = & 2v+1 \\ [\![\mathsf{linrec}_A]\!](x,g,h) & = & f \end{array}$$

where

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

The interpretation of terms is w.r.t. an environment η which assigns values to variables:

$$\begin{array}{lcl} [\![x]\!]\eta & = & \eta(x) \\ [\![\lambda x{:}A.e]\!]\eta & = & \lambda v {\in} [\![A]\!].[\![e]\!]\eta[x \mapsto v] \\ [\![e_1\ e_2]\!]\eta & = & [\![e_1]\!]\eta([\![e_2]\!]\eta) \\ [\![\mathsf{case}_A\ e_1\ \mathsf{zero}\ e_2\ \mathsf{even}\ e_3\ \mathsf{odd}\ e_4]\!]\eta & = & \begin{cases} [\![e_2]\!]\eta, & \text{if } [\![e_1]\!]\eta = 0 \\ [\![e_3]\!]\eta(v), & \text{if } [\![e_1]\!]\eta = 2v \\ [\![e_4]\!]\eta(v), & \text{if } [\![e_1]\!]\eta = 2v+1 \end{cases} \\ [\![c]\!]\eta & = & [\![c]\!] \end{array}$$

3 Review of functor categories over concrete categories

Let $\mathbb{C}$ be a small concrete category with cartesian products (written $\times$) and terminal object (written $\top$). That $\mathbb{C}$ is concrete means that the functor $\mathcal{G} : \mathbb{C} \to \mathbf{Sets}$ defined by $\mathcal{G}(X) = \mathbb{C}(\top, X)$ and $\mathcal{G}(f)(x) = f \circ x$ for $f \in \mathbb{C}(X,Y)$, $x \in \mathcal{G}(X)$ is faithful; i.e. every morphism in $\mathbb{C}$ is uniquely determined by its functional action on global elements. Notice that $\mathcal{G}$ preserves cartesian products and terminal object up to isomorphism.

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 $m \in \widehat{\mathbb{C}}(F,G)$ assigns a function $m_X : F_X \to G_X$ to each $X \in \mathbb{C}$ in such a way that $m_Y(F_u(x)) = G_u(m_X(x))$ 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 $m : \mathcal{Y}(X) \to \mathcal{Y}(Y)$ then $f =_{\mathrm{def}} m_X(\mathrm{id}_X) \in \mathbb{C}(X, Y)$, and we have $m = \mathcal{Y}(f)$. Notice that since $m_Z(g) = f \circ g$ we have $\mathcal{G}(f) = m_\top$. 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 $m : \mathcal{Y}(X) \to F$ then $m_X(\mathrm{id}_X) \in F_X$ and $m_Y(f \in \mathbb{C}(Y, X)) = F_f(m_X(\mathrm{id}_X))$ 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 exponential 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 $\top_X = \{0\}$.

It will greatly simplify the subsequent calculations if we exploit these properties of $\widehat{\mathbb{C}}$ by using an informal typed lambda calculus with the objects and morphisms of $\widehat{\mathbb{C}}$ as base types and constants in order to denote particular morphisms in $\widehat{\mathbb{C}}$. E.g. if $f : A \to B$ and $g : C \to D$ then

$$h(x{:}\,C) =_{\mathrm{def}} \lambda y{:}\,A.\langle f(y), g(x)\rangle$$

defines the evident morphism from C to $A \Rightarrow (B \times D)$ constructed from f and g.

We will also use set-theoretic notation for equalisers, e.g., if $f, g : A \to B$ then $\{x{:}\,A | f(x){=}g(x)\}$ denotes the equalizer of f and g in $\widehat{\mathbb{C}}$ given explicitly by

$$\{x{:}\,A | f(x){=}g(x)\}_X = \{x{\in}A_X | f_X(x) = g_X(x)\}$$

The inclusion map from the equalizer into A is not written.

So, in fact, we use a *dependently typed* lambda calculus in the spirit of Martin-Löf's extensional type theories [16]. This lambda calculus is referred to as the *internal language* of $\widehat{\mathbb{C}}$. Definitions and verifications in the internal language are always with respect to some ambient "context presheaf" Γ. In the beginning this context is the terminal object. Using phrases like "let $x{:}\,A$" the current context Γ may be replaced by $\Gamma \times A$. Or, if A depends on Γ then $\Sigma(\Gamma, A)$ is used instead of $\Gamma \times A$ where the elements of $\Sigma(\Gamma, A)_X$ are pairs (γ, a) where $\gamma \in \Gamma_X$ and $a \in A_X(\gamma)$. Such temporary extensions can be discharged by phrases like "we have thus defined an object of $A \Rightarrow B$" or "we have thus shown $\forall x{:}\,A. \ldots$".

If X is an object of $\mathbb{C}$ and $F \in \widehat{\mathbb{C}}$ then we can define $F^X \in \widehat{\mathbb{C}}$ by $F^X_Y = F_{Y \times X}$, $F^X_f = F_{f \times \mathrm{id}_X}$.

Lemma 3.1 *Let $F \in \widehat{\mathbb{C}}$ and $X \in \mathbb{C}$. The presheaf F^X defined by $F^X_Y = F_{Y \times X}$ and $F^X_f = F_{f \times \mathrm{id}_X}$ is isomorphic to the function space $\mathcal{Y}(X) \Rightarrow F$.*

Proof. We have $(\mathcal{Y}(X) \Rightarrow F)_Y \cong \widehat{\mathbb{C}}(\mathcal{Y}(Y) \times \mathcal{Y}(X), F) \cong \widehat{\mathbb{C}}(\mathcal{Y}(Y \times X), F) \cong F_{Y \times X} \cong F^X_Y$. □

In order to simplify notation we will treat the isomorphisms guaranteed by the above as identities. Similarly, we will identify the set of global elements $\widehat{\mathbb{C}}(\top, F)$ with the set $F_\top$ in view of $\mathcal{Y}(\top) \cong \top$. So, for example, in order to define a global element of presheaf $\mathcal{Y}(X) \Rightarrow \mathcal{Y}(Y) \Rightarrow F$ we may simply provide an element of $F_{Y \times X}$.

One could formally achieve equality between $\mathcal{Y}(X) \Rightarrow F$ and F^X by defining function spaces $G \Rightarrow F$ in $\widehat{\mathbb{C}}$ by case distinction on whether G is equal to $\mathcal{Y}(X)$ for some X or not, taking F^X in the former case. We prefer, however, not to do this and to view the convention as a shorthand for more verbose definitions in which isomorphisms are inserted in various places.

4 The category of polymax functions

Definition 4.1 *A function $f : \mathbb{N}^m \times \mathbb{N}^n \rightarrow \mathbb{N}$ is (m,n)-polymax if it is in PTIME and there exists an m-variate polynomial p such that*

$$|f(\vec{x},\vec{y})| \leq p(|\vec{x}|) + \max(|\vec{y}|)$$

for each $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$.

The category $\mathbb{B}$ has pairs (m,n) of natural numbers as objects; a $\mathbb{B}$-morphism from (m,n) to $(1,0)$ consists of an m-ary PTIME-function; a $\mathbb{B}$-morphism from (m,n) to $(0,1)$ consists of an (m,n)-polymax function. A morphism from (m,n) to (m',n') consists of m' morphisms from (m,n) to $(1,0)$ and n' morphisms from (m,n) to $(0,1)$. We view such a morphism as a function from $\mathbb{N}^{m+n}$ to $\mathbb{N}^{m'+n'}$.

It follows by an easy calculation that this is indeed a category, i.e., that the set-theoretic composition of two $\mathbb{B}$-morphisms is a $\mathbb{B}$-morphism again. We may write a morphism in $\mathbb{B}((m,n)\,,\,(m',n'))$ in the form $(\vec{u},\vec{v})$ where $\vec{u}$ consists of m' *PTIME*-functions of arity m and $\vec{v}$ consists of n' (m,n)-polymax functions. We write $\langle\rangle$ for the empty vector, thus in the above situation $(\vec{u},\langle\rangle)$ is a morphism from (m,n) to $(m',0)$. It is also easy to see that $\mathbb{B}$ has a terminal object, viz. $(0,0)$ and cartesian products given on objects by $(m,n)\times(m',n') = (m+m',n+n')$.

Notice also that since morphisms of $\mathbb{B}$ are particular functions the category $\mathbb{B}$ is concrete.

The following shows that polymax functions are closed under simultaneous safe recursion on notation and thus provides a slight generalisation of one direction of the central result in [4].

Proposition 4.2 *Let m,n,k be natural numbers and let $\vec{g} \in \mathbb{B}((m,n)\,,\,(0,k))$, $\vec{h} \in \mathbb{B}((m+1,n+k)\,,\,(0,k))$. The function(s) $\vec{f} : \mathbb{N}^{m+1}\times\mathbb{N}^n \rightarrow \mathbb{N}^k$ defined by*

$$\begin{aligned} \vec{f}(\vec{x},0;\vec{y}) &= \vec{g}(\vec{x};\vec{y}) \\ \vec{f}(\vec{x},x;\vec{y}) &= \vec{h}(\vec{x},x;\vec{y},\vec{f}(\vec{x},\lfloor\tfrac{x}{2}\rfloor\,;\vec{y}))\ \textit{if}\ x>0 \end{aligned}$$

is in $\mathbb{B}((m+1,n)\,,\,(0,k))$.

Proof. Let p_g and p_h be polynomials such that $\max|\vec{g}(\vec{x};\vec{y})| \leq p_g(|\vec{x}|) + \max|\vec{y}|$ and $\max|\vec{h}(\vec{x},x;\vec{y},\vec{u})| \leq p_h(|\vec{x}|,|x|) + \max(|\vec{y}|,|\vec{u}|)$.

We have

$$\max|\vec{f}(\vec{x},x;\vec{y})| \leq |x|\cdot p_h(|\vec{x}|,|x|) + p_g(|\vec{x}|) + \max(|\vec{y}|)$$

so the obvious Turing machine computing f runs in polynomial time and f is polymax bounded. □

The following proposition shows closure of polymax functions under simultaneous safe recursion with safe parameter substitutions. The case $k=1$ appears also in [2].

Proposition 4.3 *Let m,n,k be natural numbers and let $\vec{g} \in \mathbb{B}((m,n)\,,\,(0,k))$, $\vec{h} \in \mathbb{B}((m+1,n+k)\,,\,(0,k))$, and $\vec{u} \in \mathbb{B}((m+1,n)\,,(0,n))$. The function(s) $\vec{f} : \mathbb{N}^{m+1}\times \mathbb{N}^n \longrightarrow \mathbb{N}^k$ defined by*

$$\begin{aligned} \vec{f}(\vec{x},0;\vec{y}) &= \vec{g}(\vec{x};\vec{y}) \\ \vec{f}(\vec{x},x;\vec{y}) &= \vec{h}(\vec{x},x;\vec{y},\vec{f}(\vec{x},\lfloor\tfrac{x}{2}\rfloor\,;\vec{u}(\vec{x},x;\vec{y})))\ \textit{if}\ x>0 \end{aligned}$$

is in $\mathbb{B}((m+1,n)\,,\,(0,k))$.

Proof. Define $\vec{v} \in \mathbb{B}((m+2,n)\,,\,(0,n))$ (using Prop. 4.2) by

$$\begin{array}{rcl} \vec{v}(\vec{x},x,0;\vec{y}) & = & \vec{y} \\ \vec{v}(\vec{x},x,y;\vec{y}) & = & \vec{u}(\vec{x},\left\lfloor \frac{x}{2^{|y|}} \right\rfloor; \vec{v}(\vec{x},x,\left\lfloor \frac{y}{2} \right\rfloor; \vec{y})) \text{ if } y > 0 \end{array}$$

As y varies from 0 to x the function $\vec{v}(\vec{x},x,y;\vec{y})$ takes on the parameter values on which $\vec{f}$ is called recursively in the course of the computation of $\vec{f}(\vec{x},x;\vec{y})$.

Now consider the function(s)

$$\vec{F}(\vec{x},x,y;\vec{y}) = \vec{f}(\vec{x},\left\lfloor \frac{x}{2^{|x|-|y|}} \right\rfloor; \vec{v}(\vec{x},x,\left\lfloor \frac{x}{2^{|y|}} \right\rfloor; \vec{y}))$$

Routine calculations now show that

i. if $|y| \geq |x|$ then $F(\vec{x},x,y;\vec{y}) = f(\vec{x},x;\vec{y})$,

ii. $\vec{F}(\vec{x},x,0;\vec{y}) = \vec{g}(\vec{x};\vec{v}(\vec{x},x,x;\vec{y}))$

iii. $F(\vec{x},0,y;\vec{y}) = g(\vec{x};\vec{y})$

iv. if $|y| \leq |x|$ then $F(\vec{x},x,y;\vec{y}) = \vec{h}(\vec{x},\left\lfloor \frac{x}{2^{|x|-|y|}} \right\rfloor; \vec{v}(\vec{x},x,\left\lfloor \frac{x}{2^{|y|}} \right\rfloor; \vec{y}), \vec{F}(\vec{x},x,\left\lfloor \frac{y}{2} \right\rfloor; \vec{y}))$.

v. if $|y| > |x|$ then $F(\vec{x},x,y;\vec{y}) = F(\vec{x},x,\left\lfloor \frac{y}{2} \right\rfloor; \vec{y})$

Notice that, (v) follows directly from (i).

Now, using the fact that characteristic functions for the relations $|y| \geq |x|$ and $|y| > |x|$ with x, y normal are polymax we find that clauses (ii)–(v) give rise to a definition of F by safe recursion on notation on y (without any substitution of parameters. Once we have defined F we can recover f using the special case of (i) where $y = x$. □

5 Presheaves over polymax functions

Our aim is to interpret the calculus SLR in the functor category $\widehat{\mathbb{B}}$ in such a way that terms of type $\Box\mathsf{N}\to\mathsf{N}$, say, will be interpreted as morphisms from $\mathcal{Y}(1,0)$ to $\mathcal{Y}(0,1)$ hence—by the Yoneda lemma—as *PTIME* functions. (Notice here that the functions in $\mathbb{B}((1,0)\,,\,(0,1))$ are in 1-1 correspondence with the *PTIME*-functions.) For the linearity-free fragment this has already been achieved in [9] so we will only briefly review this construction and then focus on the issue of linearity.

We will write $\mathbf{N}$ for the functor $\mathcal{Y}(0,1)$. Notice that $\mathbf{N}_{(m,n)}$ consists of the (m,n)-polymax functions. Since $\mathbb{B}$ has cartesian products the characterisation of function spaces in Lemma 3.1 applies to the present situation and we obtain in particular that the function space $\mathbf{N}\Rightarrow F$ in $\widehat{\mathbb{B}}$ is isomorphic to the functor $F^{\mathbf{N}}$ defined by $F^{\mathbf{N}}_{(m,n)} = F_{(m,n+1)}$.

5.1 Definition by cases

The category of all presheaves is not quite yet suitable for our purposes as it does not behave well w.r.t. case distinction. Consider the constant presheaf $\mathbf{2}$ given by $\mathbf{2}_{(m,n)} = \{0,1\}$. As follows from naturality every morphism in $\widehat{\mathbb{B}}$ from $\mathbf{N}$ to $\mathbf{2}$ must be constant. In particular, there can be no morphism which would perform case distinction, e.g. be 0 on even numbers and 1 on the odd ones. Intuitively speaking, the reason for this is that $\mathbf{N}$ does not consist of natural numbers, but rather of $\mathbb{N}$-valued functions defined on $\mathbb{N}^m \times \mathbb{N}^n$ where (m,n) is the "current stage". The existence of a morphism parity : $\mathbf{N} \longrightarrow \mathbf{2}$ which is 0 on even numbers and 1 on the odd ones would mean that every such function either yields even values only or yields odd values only. This is obviously not the case. What we can say, however, is that for every such function there exists a finite partition of its domain such that in each patch it yields either even or odd values.

A well-known category-theoretic concept, namely the notion of *sheaf*, allows us to cater for such local case distinctions.

Definition 5.1 *Let (m,n) be an object of $\mathbb{B}$. A* cover *of (m,n) consists of a morphism $t \in \mathbb{B}((m,n)\,,\,(0,1))$ with range $\{0,1\}$. In other words, every $m+n$-ary PTIME-computable function with range $\{0,1\}$ is a cover.*

A presheaf $F \in \widehat{\mathbb{B}}$ is a sheaf *if for each (m,n) and every cover $t \in \mathbb{B}((m,n)\,,\,(0,1))$ and elements $f_0, f_1 \in F_{(m,n)}$ there exists a unique element $f \in F_{(m,n)}$ such that for every $u : \mathbb{B}((m,n)\,,\,(m,n))$ such that $t \circ u$ is constant, we have $F_u(f) = F_u(f_i)$.*

Notice that it would suffice to require the latter property for those u which are the identity on some $t^{-1}(i)$ for $i = 0,1$ and constant outside. The general case would then follow by the functor laws.

We say that f is obtained by *pasting* f_0, f_1.

We could define a more general notion of cover which uses functions with a range of the form $\{0,\dots,n-1\}$ and accordingly, n elements $f_0,\dots,f_{n-1}$ as "input" to pasting. However, it is easy to see that the resulting notion of sheaf agrees with the present one. This is for the same reason as generalised case distinction can be defined from binary if-then-else.

One can show that a sheaf in our sense is a sheaf for a suitable Grothendieck topology on $\mathbb{B}$ (see [17] for an accessible account of Grothendieck topologies.) It follows from this fact that the subcategory $\mathit{Sh}(\mathbb{B})$ of $\widehat{\mathbb{B}}$ consisting of the sheaves is closed under products, function, spaces, and equalisers and that these are constructed in the same way as for sheaves.

Proposition 5.2 *The representable presheaf $\mathbf{N} = \mathcal{Y}((0,1))$ is a sheaf.*

Proof. Let $t \in \mathbb{B}((m,n),(0,1))$ be a cover and let $f_0, f_1 \in \mathbf{N}_{(m,n)}$ be polymax-functions. We define $f \in \mathbb{B}((m,n),(0,1))$ by

$$f(\vec{x};\vec{y}) = f_{t(\vec{x};\vec{y})}(\vec{x};\vec{y})$$

The verification is left to the reader. $\square$

The reason for the introduction of sheaves is the following.

Proposition 5.3 *Let C be a sheaf. There exists a morphism*

$$\mathrm{ifz}_C : \mathbf{N} \times C \times C \longrightarrow C$$

such that

$$\mathrm{ifz}_C(0, c_0, c_1) = c_0$$

and such that for every nonzero global element $x : \top \longrightarrow \mathbf{N}$ we have

$$\mathrm{ifz}_C(x, c_0, c_1) = c_1$$

Moreover, ifz_C is natural in C in the sense that if $f : C \longrightarrow D$ is a morphism between sheaves then

$$f(\mathrm{ifz}_C(n, c_0, c_1)) = \mathrm{ifz}_D(n, f(c_0), f(c_1))$$

Proof. Notice that we only specify the behaviour of ifz_C when the first argument is a global element. Let us now define a morphism ifz_C with the required properties. Assume (m,n) and $f \in \mathbf{N}_{(m,n)}$ and $c_0, c_1 \in C_{(m,n)}$. The function $t \in \mathbb{B}((m,n)\,,\,(0,1))$ defined by

$$t(\vec{x};\vec{y}) = \begin{cases} 0, \text{ if } f(\vec{x};\vec{y}) = 0 \\ 1, \text{otherwise} \end{cases}$$

defines a cover on (m,n). We define $(\mathrm{ifz}_C)_{(m,n)}(f, c_0, c_1)$ as the unique element $c \in C_{(m,n)}$ such that $t \circ u = i \in \{0,1\}$ implies $C_u(c) = c_i$.

Naturality of the thus defined family of maps $(\mathrm{ifz}_C)_{(m,n)}$ is a direct consequence of uniqueness of pasting. (Going both sides of the naturality square yields elements satisfying the requirement of a pasting.) The desired property of ifz_C is also a consequence of uniqueness: if f is constantly 0 so is t and then c_0 itself is a pasting thus equal to c. Similarly, if f is constantly 1. Finally, this line of argument also establishes naturality of ifz_C in C. $\Box$

5.2 A comonad on $\widehat{\mathbb{B}}$

We have a functor $\Box : \widehat{\mathbb{B}} \to \widehat{\mathbb{B}}$ defined by $(\Box F)_{(m,n)} = F_{(m,0)}$ and $(\Box F)_{(\vec{u},\vec{v})} = F_{(\vec{u},0)}$. If $f : F \to G$ is a morphism in $\widehat{\mathbb{B}}$ then $\Box f : \Box F \to \Box G$ is given by $(\Box f)_{(m,n)} = f_{(m,0)}$.

It is easily seen that this is indeed a functor. More specially, it extends to a comonad on $\widehat{\mathbb{B}}$ and restricts to $\mathit{Sh}(\mathbb{B})$. The notion of comonad is dual to the more well-known concept of a monad defined e.g. in [18]. All comonads arising in this paper are of a very special form, namely an endofunctor $F : \mathbb{C} \to \mathbb{C}$ on some category $\mathbb{C}$ such that $F \circ F = F$ together with a natural transformation $\varepsilon : F \to \mathit{Id}$ called counit such that $F(\varepsilon) = \mathrm{id}$.

In the particular case at hand the counit is the natural transformation $\mathsf{unbox} : \Box \to \mathrm{Id}$ given by $(\mathsf{unbox}_F)_{(m,n)} = F_\pi$ where $\pi \in \mathbb{B}((m,n)\,,\,(m,0))$ is the projection on the first component. Obviously, we have $\Box \circ \Box = \Box$ and $\Box(\mathsf{unbox}) = \mathrm{id}$.

The fact that $\Box$ is a comonad allows us to lift a morphism $f : \Box F \to G$ to a morphism $f^\Box : \Box F \to \Box G$, in fact, we have $f^\Box = \Box(f)$ in view of $\Box\Box = \Box$. Notice that $\Box$ is *not* a strong comonad, i.e., there is in general no way of lifting a morphism $f : H \times \Box F \to G$ to a morphism from $H \times \Box F$ to $\Box G$. Also notice how the lifting operation corresponds to the rule of necessitation found in modal logic.

The comonad $\Box$ has the further property that it commutes with cartesian products in the sense that $\Box(A \times B) = \Box A \times \Box B$ and $\Box \pi_i = \pi_i$ for $i = 1, 2$.

The reader not familiar with the notion of comonad need only remember the definition of $\Box$ and the definition and typing of unbox.

Now consider the sheaf $\Box\mathbf{N}$. We have $\Box\mathbf{N}_{(m,n)} = \mathbf{N}_{(m,0)} = \mathbb{B}((m,0)\,,\,(0,1)) \cong \mathbb{B}((m,n)\,,\,(1,0))$; where the last step follows from the definition of polymax functions: a $(m,0)$-polymax function is the same as an m-ary *PTIME*-function. This means that $\Box\mathbf{N}$ is isomorphic to the sheaf $\mathcal{Y}(1,0)$ and Lemma 3.1 gives $(\Box\mathbf{N} \Rightarrow F)_{(m,n)} \cong F_{(m+1,n)}$. Again, we will treat this isomorphism as an identity.

This allows us to lift recursion on notation (Prop. 4.2) and recursion on notation with substitution of parameters to global elements

$$\mathsf{saferec}^{(k)} : \quad \top \longrightarrow \Box\mathbf{N} \Rightarrow A \Rightarrow (\Box\mathbf{N} \Rightarrow A \Rightarrow A) \Rightarrow A$$

$$\begin{aligned} \mathsf{substrec}^{(k)} : \quad \top \longrightarrow\ & \Box\mathbf{N} \Rightarrow \\ & (A \Rightarrow \mathbf{N}) \Rightarrow \\ & (\Box\mathbf{N} \Rightarrow A \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \\ & (A \Rightarrow A) \Rightarrow A \Rightarrow \mathbf{N} \end{aligned}$$

where $A = \mathbf{N}^k$.

We have shown in [9] that in this way one obtains a model of the linearity-free fragment of SLR which can be used to show that all definable functions of type $\Box\mathsf{N}^n \to \mathsf{N}$ are in *PTIME*.

6 Chu spaces

The notion of Chu space has been introduced by Michael Barr as a canonical example of a $*$-autonomous category (a certain model of linear logic). Afterwards, it has been recognised (notably by Vaughan Pratt) that Chu spaces provide an abstract model of duality between objects and attributes into which many concepts like topological spaces and lattices of open sets or vector spaces and linear forms can be embedded in a natural way.

Here we want to put forward the use of Chu spaces as a generalised continuation-passing-style (CPS) transformation. Although we do this with a very specific application in mind, we hope that by introducing this concept into the functional programming community more applications will be found.

The starting question here is the following.

"What benefit can we extract from the knowledge that a function(al) is linear?"

One answer will be provided by the Chu space interpretation we are going to give. Namely, it will allow us to deduce that if we are given a linear functional $F : (A \multimap \mathsf{N}) \multimap \mathsf{N}$ where A is arbitrary and where F possibly depends on parameters then we can effectively come up with an element $\mathsf{arg} : A$ and a "continuation" $\mathsf{rest} : \mathsf{N} \to \mathsf{N}$ (both also depending on possible parameters of course) such that for every $u : A \to \mathsf{N}$ it holds that

$$F(u) = \mathsf{rest}(u(\mathsf{arg}(u)))$$

As stated in the introduction this will allow us to reduce higher type recursion with linear step function to first-order recursion with parameter substitution.

The idea behind the Chu space interpretation is that every type A gets interpreted as a "space of values" or denotations $|[\![A]\!]|$ and a "space of continuations" $|[\![A]\!]|^*$ and, finally, a function $\delta_A : |[\![A]\!]| \times [\![A]\!]^* \to R$ where R is some object of "responses". The analogy with CPS transformation stems from the observation that there we translate every type either into a set of denotations $[\![A]\!]$ and define the set of continuations as $[\![A]\!] \to R$, or we translate every type A into a set of continuations and recover denotations as functions from continuations to responses. In the Chu space interpretation we can as it were arbitrarily choose both continuations and denotations as long as we say how to apply a continuation to a denotation to yield a response (that's the purpose of the map δ.)

Such Chu spaces are usually formed w.r.t. the category of sets, i.e. $|[\![A]\!]|$ and $[\![A]\!]^*$ are sets; δ_A is a function. However, they can be formed w.r.t. other categories as well as long as these support cartesian products, function spaces, and equalisers. The last requirement (equalisers) hampers a view of an interpretation in a category of Chu spaces as a syntactic translation. Notice, conversely, that the CPS translation is usually presented as a syntactic translation can also be seen as interpretation in an appropriate model.

If one insists on using syntax to construct Chu spaces one would first have to conservatively embed the target calculus into a system with equationally defined subset types.

We could adopt here such a strategy with the target language being the a version of SLR with first-order safe recursion with parameter substitution; a system of which we already know that it captures *PTIME* by interpretation in the functor category $\widehat{\mathbb{B}}$.

It is more direct, however, to use Chu spaces over the category $Sh(\mathbb{B})$ directly, since it supports all the required structure. The object of responses we use is $R = \mathbf{N}$.

Definition 6.1 *A Chu-space (over $Sh(\mathbb{B})$) is a triple $A = (|A|, A^*, \delta_A)$ where $|A|, A^*$ are objects of $Sh(\mathbb{B})$ and $\delta_A : |A| \times A^* \to \mathbf{N}$. We call $|A|$ and A^* the value space and continuation space of A. The map δ_A is called the evaluation map.*

If A, B are Chu-spaces then a morphism from A to B is a pair $f = (|f|, f^*)$ where $|f| : |A| \to |B|$ and $f^* : B^* \to A^*$ such that the following diagram called adjointness condition commutes:

$$\begin{tikzcd}
|A| \times B^* \arrow[r, "|f| \times B^*"] \arrow[d, "|A| \times f^*"'] & |B| \times B^* \arrow[d, "\delta_B"] \\
|A| \times A^* \arrow[r, "\delta_A"'] & \mathbf{N}
\end{tikzcd}$$

Here—following common practice—we denote the identity at some object X by that object itself rather than by id_X. Again, $|f|$ is called the value map; f^* is called the continuation map of f.

The composition of morphisms is given componentwise by $|f \circ g| = |f| \circ |g|$ and $(f \circ g)^* = g^* \circ f^*$.

Note that we can express the adjointness condition using the internal language by requiring that for all $a{:}\,|A|$ and $\beta{:}\, B^*$ we have $\delta_B(|f|(a), \beta) = \delta_A(a, f^*(\beta))$.

6.1 Examples of Chu spaces

It is known that Chu spaces form a symmetric monoidal closed category [20]. We only give here the constructions on objects associated with this fact; the definition of the morphism parts such as currying, as well as the verifications are then routine and left to the reader.

In order to define particular Chu spaces we make use of the informal typed lambda calculus described above. To ease understanding the first few examples will, however, be given together with explicit definitions employing "categorical combinators".

The Chu space $\mathbf{N}$ is defined by $|\mathbf{N}| = \mathbf{N}$ and $\mathbf{N}^* = (\mathbf{N} \Rightarrow \mathbf{N})$ where $\mathbf{N}$ on the right hand side refers to $Sh(\mathbb{B})$ of course. The mapping $\delta_{\mathbf{N}}$ is simply the evaluation map $\mathbf{N} \times (\mathbf{N} \Rightarrow \mathbf{N}) \to \mathbf{N}$. Alternatively, we can define $\delta_{\mathbf{N}}$ using the internal language of $Sh(\mathbb{B})$ by $\delta_{\mathbf{N}}(n, \nu) = \nu(n)$.

The linear function space $A \multimap B$ is a bit more complicated: Its value space is defined in the internal language as the following subset type.

$$\begin{aligned}|A \multimap B| = \{(|f|, f^*) \mid\, & |f| : |A| \Rightarrow |B|, f^* : B^* \to A^*, \\ & \forall a{:}\,|A|.\forall \beta{:}\,B^*.\delta_B(|f|(a), \beta) = \delta_A(a, f^*(\beta))\}\end{aligned}$$

Explicitly, we define it as the following pullback in $Sh(\mathbb{B})$:

$$\begin{tikzcd}
|A \multimap B| \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & |A| \Rightarrow |B| \arrow[d, "g"] \\
B^* \Rightarrow A^* \arrow[r, "f"'] & (|A| \times B^*) \Rightarrow \mathbf{N}
\end{tikzcd}$$

where f and g are maps constructed in the obvious way from δ_A and δ_B by currying and some wiring.

Intuitively, $|A \multimap B|$ is the set of morphisms from A to B. This intuition is somewhat misleading, though, since the homset $\mathbf{Chu}(A, B)$ is an actual set, whereas $|A \multimap B|$ is a sheaf.

The space of continuations for $A \multimap B$ is $|A| \times B^*$. The evaluation map is given by

$$\delta_{A \multimap B}((|f|, f^*), (a, \beta)) = \delta_B(|f|(a), \beta) \quad (= \delta_A(a, f^*(\beta)))$$

From now on, we omit the explicit definitions.

The tensor product $A \otimes B$ is the most complicated construction we will encounter. We have

$$\begin{aligned}
&|A \otimes B| = |A| \times |B| \\
&(A \otimes B)^* = \{(\alpha, \beta) \mid \alpha \colon |B| \Rightarrow A^*, \beta \colon |A| \Rightarrow B^*, \forall a \colon |A|. \\
&\qquad\qquad \forall b \colon |B|. \delta_A(a, \alpha(b)) = \delta_B(b, \beta(a))\} \\
&\delta_{A \otimes B}((a, b), (\alpha, \beta)) = \delta_A(a, \alpha(b)) \quad (= \delta_B(b, \beta(a)))
\end{aligned}$$

If $f : A \to C$ and $g : B \to D$ are morphisms between Chu spaces then we define $f \otimes g : A \otimes B \to C \otimes D$ by $|f \otimes g|(a, b) = (|f|(a), |g|(b))$ and $(f \otimes g)^*(\gamma, \delta) = (f^* \circ \gamma \circ |g|, g^* \circ \delta \circ |f|)$. The verifications are by straightforward equality reasoning.

There is also a neutral element for the tensor product: the Chu space I defined by $|I| = \{0\}$, $I^* = \mathbf{N}$, $\delta_I(0, n) = n$. We have $I \otimes A \cong A \otimes I \cong A$.

Proposition 6.2 *These settings endow* ***Chu*** *with the structure of a symmetric monoidal closed category in the sense that there are coherent isomorphisms* $I \otimes A \cong A$, $A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$, $A \otimes B \cong B \otimes A$, *and* $\boldsymbol{Chu}(A \otimes B, C) \cong \boldsymbol{Chu}(A, B \multimap C)$.

6.2 Nonlinearisation

For Chu space A we define its nonlinearisation $!A$ by $|!A| = |A|$ and $(!A)^* = |A| \Rightarrow \mathbf{N}$. The evaluation map $\delta_{!A}$ is given by ordinary evaluation, i.e., $\delta_{!A}(a, \alpha) = \alpha(a)$. Notice that $!\mathbf{N} = \mathbf{N}$. Nonlinearisation is actually a functor with morphism part given by $|!f| = |f|$ and $(!f)^*(\alpha) = \alpha \circ |f|$. It extends to a comonad on $\mathbf{Chu}$ with counit $\mathsf{derelict}_A : !A \to A$ given by $|\mathsf{derelict}_A|(a) = a$ and $\mathsf{derelict}^*_A(\alpha) = \lambda a. \delta_A(a, \alpha)$. As with $\Box$ we have $!!A = !A$ and $!\mathsf{derelict}_A = \mathrm{id}_{!A}$. Let A, B be Chu spaces and consider the function space $!A \multimap B$. An element of $|!A \multimap B|$ consists of a function $|f| : |A| \to |B|$ and a function $f^* : B^* \to (|A| \Rightarrow \mathbf{N})$ such that

$$f^*(\beta)(a) = \delta_B(|f|(a), \beta)$$

for all $a : |A|$ and $\beta : B^*$. But now, f^* is uniquely determined by this requirement so that the only relevant component is $|f|$. The space of continuations is the same as for $A \multimap B$, namely $(!A \multimap B)^* = |A| \times B^*$. Thus, $!A \multimap B$ is canonically isomorphic to the Chu space $A \Rightarrow B$ defined by

$$\begin{aligned}
&|A \Rightarrow B| = |A| \Rightarrow |B| \\
&(A \Rightarrow B)^* = |A| \times B^* \\
&\delta_{A \Rightarrow B}(f, (a, \beta)) = \delta_B(f(a), \beta)
\end{aligned}$$

We remark that $A \Rightarrow B$ is not cartesian function space. Indeed, the category of Chu spaces has cartesian products given by $|A \times B| = |A| \times |B|$ and $(A \times B)^* = A^* + B^*$. These, however, lack a right adjoint.

6.3 Modality

Finally, we extend the comonad $\Box$ to Chu spaces by

$$\begin{aligned} |\Box A| &= \Box|A| \\ (\Box A)^* &= \Box A \Rightarrow \mathbf{N} \\ \delta_{\Box A}(a, \alpha) &= \alpha(a) \end{aligned}$$

If $f : A \to B$ then $|\Box f| = \Box|f|$ and $(\Box f)^* = f^*$.

The counit $\mathsf{unbox}_A : \Box A \to A$ is given by $|\mathsf{unbox}_A|(a) = \mathsf{unbox}_{|A|}(a)$ and $\mathsf{unbox}^*_A(\alpha) = \lambda a.\delta_A(\mathsf{unbox}_A(a), \alpha)$. As in $\widehat{\mathbb{B}}$ we have $\Box\Box = \Box$ and $\Box\mathsf{unbox}_A = \mathsf{unbox}_A$ so $\Box$ is again a comonad. We also notice that $!\Box A = \Box !A = \Box A$.

Proposition 6.3 *Let A, B be Chu spaces.*

i. If $A = !A$ then $\boldsymbol{Chu}(A, B) \cong \widehat{\mathbb{B}}(|A|, |B|)$, moreover, $|A \multimap B| \cong |A| \Rightarrow |B|$ in this case.

ii. If $!A = A$ and $!B = B$ then $A \otimes B \cong !(A \otimes B)$.

iii. The set of global elements of Chu space A, i.e., the Chu space morphisms from I to A are in 1-1 correspondence with global elements of $|A|$, i.e., elements of $A_{(0,0)}$.

Proof. Ad i) The continuation part of a morphism $f : !A \longrightarrow B$ is uniquely determined by its value part $|f| : |A| \longrightarrow |B|$ by the adjointness condition. Namely, for $\beta : B^*$ we have $f^*(\beta) = \lambda x{:}\,|A|.\delta_B(|f|(x), \beta)$.

By the same argument the first projection $|A \multimap B| \longrightarrow |A \Rightarrow B|$ extends to an isomorphism whose inverse sends $f : |A| \Rightarrow |B|$ to the pair (f, f^*) where $f^*(\beta) = \lambda x{:}\,|A|.\delta_B(f(x), \beta)$.

Ad ii) Here we notice that if $!A = A$ and $!B = B$ then $(A \otimes B)^*$ consists of two functions $u : |A| \Rightarrow (|B| \Rightarrow \mathbf{N})$ and $v : |B| \Rightarrow (|A| \Rightarrow \mathbf{N})$ such that for each $a : |A|$ and $b : |B|$ we have $u(a)(b) = v(b)(a)$. Hence, u and v are one and the same function and thus $(A \otimes B) \cong !(A \otimes B)$.

Ad iii) This follows from part i) in view of $I = !I$. $\Box$

6.4 Affine Chu spaces

Let A be a Chu space. In order to define projections $A \otimes B \to A$ for each B it is sufficient to have a morphism $\mathsf{discard}_A : A \to I$. We will see that all the Chu spaces which are of interest to us admit such a morphism. In order to define it inductively we also need global elements of the Chu spaces of interest. This motivates the following definition.

Definition 6.4 *A Chu space A is called affine if it has I as a retract, i.e., if there are morphisms $\mathsf{elem}_A : I \to A$ and $\mathsf{discard}_A : A \to I$ such that $\mathsf{discard}_A \circ \mathsf{elem}_A = \mathrm{id}_I$.*

If B is affine and A is arbitrary then we have a projection $\pi_{A,B} : A \otimes B \to A$ obtained by composing $\mathrm{id}_A \otimes \mathsf{discard}_B : B \to I$ with the isomorphism $A \otimes I \cong A$. Similarly, we obtain a second projection $\pi'_{B,A} : B \otimes A \to A$.

Unfortunately, these projections do not necessarily form a natural transformation, i.e., if both B and B' are affine, $f : A \to A'$ and $g : B \to B'$ then it need not be the case that $\pi_{A',B'} \circ (f \otimes g) = f \circ \pi_{A,B}$. Notice, however, that $|\pi_{A,B}|(a, b) = a$, so the naturality equation holds for the respective value maps.

Proposition 6.5 *The Chu space $\mathbf{N}$ is affine. If A, B are affine so are $A \multimap B$, $A \Rightarrow B$, $\Box A$, $A \otimes B$, $!A$.*

Proof. Define $|\mathsf{discard}_{\mathbf{N}}| = 0$ and $(\mathsf{discard}_{\mathbf{N}})^*(n) = \lambda x.n$; define $|\mathsf{elem}_{\mathbf{N}}| = 0$ and $(\mathsf{elem}_{\mathbf{N}})^*(\nu) = \nu(0)$. This establishes that $\mathbf{N}$ is affine. The other claims follow from $I \cong I \multimap I \cong I \otimes I \cong {!I} \cong \Box I$. $\square$

6.5 Linear N-valued functionals

The whole point about Chu spaces is that they make precise the intuition that a (affine) linear function uses its argument (at most) once. In particular, if $F : (A \Rightarrow \mathbf{N}) \longrightarrow (A \Rightarrow \mathbf{N})$ is a map between Chu spaces then there are $\widehat{\mathbb{B}}$-morphisms $\mathsf{arg} : |A| \rightarrow |A|$ and $\mathsf{rest} : |A| \rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ such that

$$|F|(u)(a) = \mathsf{rest}(a)(u(\mathsf{arg}(a)))$$

Indeed, these functions can be obtained as the two components of $\lambda a{:}|A|.F^*(a, \mathrm{id})$ where id is the identity function viewed as an element of $\mathbf{N}^* = (\mathbf{N} \Rightarrow \mathbf{N})$.

More generally, we can characterise the linear function space $(A \Rightarrow \mathbf{N}) \multimap \mathbf{N}$ as follows.

Definition 6.6 *Let A be a Chu space. The Chu space $\mathbf{Lin}(A)$ of linear $\mathbf{N}$-valued functionals on $A \multimap \mathbf{N}$ is defined by*

$$\begin{array}{lcl} |\mathbf{Lin}(A)| & = & |A| \times (\mathbf{N} \Rightarrow \mathbf{N}) \\ \mathbf{Lin}(A)^* & = & |A \multimap \mathbf{N}| \times (\mathbf{N} \Rightarrow \mathbf{N}) \quad (= ((A \multimap \mathbf{N}) \multimap \mathbf{N})^*) \\ \delta_{\mathbf{Lin}(A)}((\mathsf{arg}, \mathsf{rest})\,,\,(u, \nu)) & = & \nu(\mathsf{rest}(u.1(\mathsf{arg}))) \end{array}$$

Here $u.1$ refers to the first component of $u : |A \multimap \mathbf{N}|$ hence $u.1 : |A| \Rightarrow \mathbf{N}$.

Proposition 6.7 *The Chu space $\mathbf{Lin}(A)$ is functorial in A and as such naturally isomorphic to $(A \multimap \mathbf{N}) \multimap \mathbf{N}$. The value part of the isomorphism sends $(\mathsf{arg}, \mathsf{rest})$ to $\lambda u.\mathsf{rest}(u.1(\mathsf{arg}))$.*

Proof. The continuation part of the isomorphism is simply the identity. It is obvious from the definition that this defines a morphism between Chu spaces. To see that it is an isomorphism we define a morphism $\psi : ((A \multimap \mathbf{N}) \multimap \mathbf{N}) \longrightarrow \mathbf{Lin}(A)$ as follows.

Let $(|F|, F^*)$ be an "element" (i.e. a formal variable towards the definition of a morphism) of $(A \multimap \mathbf{N}) \multimap \mathbf{N}$. This means that $|F| : |A \multimap \mathbf{N}| \Rightarrow \mathbf{N}$ and $F^* : \mathbf{N}^* \Rightarrow (A \multimap \mathbf{N})^* = (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (|A| \times (\mathbf{N} \Rightarrow \mathbf{N}))$ and

$$\nu(|F|(u)) = \delta_{A \multimap \mathbf{N}}(F^*(\nu)\,,\,u) = F^*(\nu).2(u.1(F^*(\nu).1))$$

for $u : |A \multimap \mathbf{N}|$ and $\nu : \mathbf{N}^* = \mathbf{N} \Rightarrow \mathbf{N}$. This follows from the definition of linear function space. Now setting ν equal to the identity gives us

$$|F|(u) = \mathsf{rest}(u.1(\mathsf{arg}))$$

where $\mathsf{rest} = F^*(\mathrm{id}).2$ and $\mathsf{arg} = F^*(\mathrm{id}).1$.

Therefore, putting

$$\begin{array}{l} |\psi|((|F|, F^*)) = F^*(\mathrm{id}) \\ \psi^*(u, \nu) = (u, \nu) \end{array}$$

yields the desired inverse. $\square$

7 Interpretation of SLR in the category of Chu spaces

We are now ready to define an interpretation of SLR judgements in the category **Chu**. To each aspect a we associate a functor $F_a : \mathbf{Chu} \to \mathbf{Chu}$ by

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

We also define a natural transformation $\varepsilon_a : F_a \to Id$ as an appropriate composition of unbox and $\mathsf{derelict}$.

To each type A we associate a Chu space $[\![A]\!]$ by

$$\begin{array}{lcl} [\![\mathsf{N}]\!] & = & \mathbf{N} \\ [\![A_1 \xrightarrow{a} A_2]\!] & = & F_a([\![A_1]\!]) \multimap [\![A_2]\!] \end{array}$$

Notice that $[\![A \Rightarrow B]\!] \cong [\![A]\!] \Rightarrow [\![B]\!]$ and $[\![\Box A \Rightarrow B]\!] \cong \Box[\![A]\!] \Rightarrow [\![B]\!]$ in view of Prop. 6.3.

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

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

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

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

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 foresee no serious obstacle against establishing such results; the reason is merely that we do not need them.

All we are interested in is to establish a relationship between the set-theoretic semantics and the Chu space interpretation which establishes that the set-theoretic semantics stays within polynomial time.

7.1 Constants

An integer constant n (under some type assignment Γ) is interpreted as the composition of $\mathsf{discard}_{[\![\Gamma]\!]} : [\![\Gamma]\!] \to I$ with the global element of $\mathbf{N}$ corresponding to the constant using $\mathbf{Chu}(I, \mathbf{N}) \cong \mathbf{N}_{(0,0)}$.

In order to interpret the functional constants S_0 and S_1 we compose $\mathsf{discard}_{[\![\Gamma]\!]}$ with the set-theoretic functions $\lambda x {\in} \mathbb{N}.2x$ and $\lambda x {\in} \mathbb{N}.2x + 1$ lifted to global elements of $\mathbf{N} \Rightarrow \mathbf{N}$ using the isomorphisms

$$\mathbf{Chu}(I, \mathbf{N} \Rightarrow \mathbf{N}) \cong \widehat{\mathbb{B}}(1, \mathbf{N} \Rightarrow \mathbf{N}) \cong \mathbf{N}_{(0,1)}$$

7.2 Variables

A variable $\Gamma \vdash x : \Gamma(x)$ where $x \in \text{dom}(\Gamma)$ is interpreted as the appropriate projection function, followed by a counit. More precisely, if

$$[\![\Gamma]\!] = [\![\Gamma_1]\!] \otimes F_{\Gamma((x))}([\![\Gamma(x)]\!]) \otimes [\![\Gamma_2]\!]$$

then $[\![\Gamma \vdash x : \Gamma(x)]\!]$ is obtained as follows

$$[\![\Gamma_1]\!] \otimes F_{\Gamma((x))}([\![\Gamma(x)]\!]) \otimes [\![\Gamma_2]\!] \xrightarrow{\mathsf{discard}_{[\![\Gamma_1]\!]} \otimes \mathsf{id}_{[\![\Gamma(x)]\!]} \otimes \mathsf{discard}_{[\![\Gamma_2]\!]}} I \otimes F_{\Gamma((x))}([\![\Gamma(x)]\!]) \otimes I \cong F_{\Gamma((x))}([\![\Gamma(x)]\!]) \xrightarrow{\varepsilon_a} [\![\Gamma(x)]\!]$$

7.3 Subsumption

By induction on the definition of subtyping we define coercion morphisms $\iota_{A,A'} : [\![A]\!] \to [\![A']\!]$ for any two types A, A' with $A \leq A'$. The definition of $\iota_{A,A'}$ is by induction on the derivation of $A \leq A'$ using as basic ingredients identity, composition, the isomorphism $!X \multimap Y \cong X \Rightarrow Y$, the counits unbox and $\mathsf{derelict}$, and the morphism part of the $\multimap$-functor. For example, the coercion from $[\![\mathsf{N} \Rightarrow A]\!]$ to $[\![\square\mathsf{N} \multimap B]\!]$ when $A \leq B$ is defined as the composition

$$[\![\mathsf{N} \Rightarrow A]\!] = \mathbf{N} \Rightarrow [\![A]\!] = \mathbf{N} \multimap [\![A]\!] \xrightarrow{\mathsf{unbox}_{\mathsf{N}} \multimap \iota_{A,B}} \square\mathbf{N} \multimap [\![B]\!] = [\![\square\mathbf{N} \multimap A]\!]$$

7.4 Abstraction

If $\Gamma, x \overset{a}{:} A \vdash e : B$ then

$$f := [\![\Gamma, x \overset{a}{:} A \vdash e : B]\!] : [\![\Gamma]\!] \otimes [\![x \overset{a}{:} A]\!] \longrightarrow [\![B]\!]$$

The transpose of f along the adjunction $\otimes \dashv \multimap$ yields a morphism from $[\![\Gamma]\!]$ to $[\![A \xrightarrow{a} B]\!]$ which serves as the interpretation of $\lambda x{:}A.e$.

7.5 Application

Suppose that $f_1 : [\![\Gamma]\!] \otimes [\![\Delta_1]\!] \longrightarrow [\![A \xrightarrow{a} B]\!]$ and that $f_2 : [\![\Gamma]\!] \otimes [\![\Delta_2]\!] \longrightarrow [\![A]\!]$ are the interpretations of $\Gamma, \Delta_1 \vdash e_1 : A \xrightarrow{a} B$ and $\Gamma, \Delta_2 \vdash e_2 : A$ respectively. We may assume that Γ is nonlinear and that all bindings in Γ, Δ_2 have an aspect smaller than a. This last requirement shows that $F_a([\![\Gamma, \Delta]\!]) \cong [\![\Gamma, \Delta]\!]$ by Prop. 6.3. Since Γ is nonlinear we have $![\![\Gamma]\!] \cong [\![\Gamma]\!]$ and the diagonal map $|[\![\Gamma]\!]| \longrightarrow |[\![\Gamma]\!] \otimes [\![\Gamma]\!]|$ extends to a Chu space morphism by Prop. 6.3 thus allowing us to define a wiring map

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

We obtain the interpretation of $e_1\ e_2$ as the following morphism

$$\begin{aligned} &[\![\Gamma, \Delta_1, \Delta_2]\!] \xrightarrow{w} [\![\Gamma, \Delta_1]\!] \otimes [\![\Gamma, \Delta_2]\!] \xrightarrow{f_1 \otimes F_a(f_2)} [\![A \xrightarrow{a} B]\!] \otimes F_a[\![A]\!] \cong \\ &\quad \cong (F_a[\![A]\!] \multimap [\![B]\!]) \otimes F_a[\![A]\!] \xrightarrow{\mathrm{eval}} [\![B]\!] \end{aligned}$$

where eval is the evaluation map.

7.6 Case distinction

We could interpret case distinction directly, but it considerably simplifies the notation if we merely show the interpretation of the following three special cases from which all instances of case distinction are obviously definable.

- The conditional defined explicitly by

$$\mathsf{cond}(e_1, e_2, e_3) =_{\mathrm{def}} \mathsf{case}_A e_1\ \mathsf{zero}\ e_2\ \mathsf{even}\ \lambda x{:}\mathsf{N}.e_3\ \mathsf{odd}\ \lambda x{:}\mathsf{N}.e_3$$

- The division by two $\mathsf{divtwo} : \mathsf{N} \to \mathsf{N}$ from Section 1.3.
- The parity function

$$\mathsf{parity} : \mathsf{N} \to \mathsf{N}$$

  defined by

$$\mathsf{parity} = \lambda x{:}\mathsf{N}.\mathsf{case}_{\mathsf{N}} x\ \mathsf{zero}\ 0\ \mathsf{even}\ 0\ \mathsf{odd}\ 1$$

The interpretation of the latter two functions is analogous to the interpretation of the constructors $\mathsf{S}_0, \mathsf{S}_1$ and follows immediately from the fact that $\mathbf{Chu}(\mathbf{N},\mathbf{N}) \cong \mathbb{B}((0,1),(0,1))$.

In order to interpret the conditional we assume an arbitrary instance, i.e.,

$$\begin{array}{c} \Gamma, \Delta_1 \vdash e_1 : \mathsf{N} \\ \Gamma, \Delta_2 \vdash e_2 : A \\ \Gamma, \Delta_2 \vdash e_3 : A \end{array}$$

where the common type assignment Γ is nonlinear.

Let f_1, f_2, f_3 be the interpretations of e_1, e_2, e_3.

Let us use the notation

$$f : [\![\Gamma]\!] \otimes [\![\Delta_1]\!] \otimes [\![\Delta_2]\!] \longrightarrow \mathbf{N}$$

for the interpretation of $\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{Cond}(e_1, e_2, e_3) : \mathsf{N}$ which we are going to construct now.

The value component $|f|$ of f is to be a $Sh(\mathbb{B})$-morphism from $|[\![\Gamma]\!]| \times |[\![\Delta_1]\!]| \times |[\![\Delta_2]\!]|$ to $|[\![A]\!]|$.

We obtain it by plugging the value components of f_1, f_2, f_3 into an appropriate instance of the morphism ifz from Proposition 5.3. More precisely,

$$|f|(\gamma, \delta_1, \delta_2) = \mathrm{ifz}_{|[\![A]\!]|}(|f_1|(\gamma, \delta_1)\,,\ |f_2|(\gamma, \delta_2)\,,\ |f_3|(\gamma, \delta_2))$$

It remains to define the continuation component f^*. It takes the form of a $Sh(\mathbb{B})$-morphism from $[\![A]\!]^*$ to $([\![\Gamma]\!] \otimes [\![\Delta_1]\!] \otimes [\![\Delta_2]\!])^*$.

According to the definition of continuation parts of tensor products this means that we have to define three morphisms

$$\begin{array}{rl} f^*_\Gamma : [\![A]\!]^* \times |[\![\Delta_1]\!]| \times |[\![\Delta_2]\!]| & \longrightarrow [\![\Gamma]\!]^* \\ f^*_{\Delta_1} : [\![A]\!]^* \times |[\![\Gamma]\!]| \times |[\![\Delta_2]\!]| & \longrightarrow [\![\Delta_1]\!]^* \\ f^*_{\Delta_2} : [\![A]\!]^* \times |[\![\Gamma]\!]| \times |[\![\Delta_1]\!]| & \longrightarrow [\![\Delta_2]\!]^* \end{array}$$

These are to satisfy

$$\begin{array}{cl} & \delta_\Gamma(f^*_\Gamma(\alpha, d_1, d_2)\,,\ g) = f^*_\Gamma(\alpha, d_1, d_2)(g) \\ \overset{(I)}{=} & \delta_{\Delta_1}(f^*_{\Delta_1}(\alpha, g, d_2)\,,\ d_1) \\ \overset{(II)}{=} & \delta_A(\alpha\,,\ |f|(g, d_1, d_2)) \\ \overset{(III)}{=} & \delta_{\Delta_2}(f^*_{\Delta_2}(\alpha, g, d_1)\,,\ d_2) \end{array}$$

for $g : |[\![\Gamma]\!]|, d_1 : |[\![\Delta_1]\!]|, d_2 : |[\![\Delta_2]\!]|, \alpha : [\![A]\!]^*$. Notice that the non-supercsripted equality follows from the fact that Γ is nonlinear.

We have to our disposal the continuation parts of f_1, f_2, f_3 which take the form

$$\begin{array}{rl} f^*_1 : |[\![\Gamma]\!]| \times [\![\mathbf{N}]\!]^* & \longrightarrow [\![\Delta_1]\!]^* \\ f^*_2 : |[\![\Gamma]\!]| \times [\![A]\!]^* & \longrightarrow [\![\Delta_2]\!]^* \\ f^*_3 : |[\![\Gamma]\!]| \times [\![A]\!]^* & \longrightarrow [\![\Delta_2]\!]^* \end{array}$$

and satisfy the following equations:

$$\begin{array}{rcl} \delta_{\Delta_1}(f^*_1(g, \nu)\,,\ d_1) & \overset{(A)}{=} & \nu(|f_1|(g, d_1)) \\ \delta_{\Delta_2}(f^*_2(g, \alpha)\,,\ d_2) & \overset{(B)}{=} & \delta_A(\alpha, |f_2|(g, d_2)) \\ \delta_{\Delta_2}(f^*_3(g, \alpha)\,,\ d_2) & \overset{(C)}{=} & \delta_A(\alpha, |f_3|(g, d_2)) \end{array}$$

We make the following definitions:

$$
\begin{aligned}
f^*_\Gamma(\alpha, d_1, d_2) &= \lambda g{:}\,|\llbracket\Gamma\rrbracket|.\delta_{\Delta_1}(f^*_{\Delta_1}(\alpha, g, d_1)\,,\, d_1)\\
f^*_{\Delta_1}(\alpha, g, d_2) &= f^*_1(g, \lambda n.\delta_{\llbracket A\rrbracket^*}(\alpha, \mathrm{ifz}_{|\llbracket A\rrbracket|}(n, |f_2|(g, d_2), |f_3|(g, d_2))))\\
f^*_{\Delta_2}(\alpha, g, d_1)) &= \mathrm{ifz}_{\llbracket\Delta_2\rrbracket^*}(|f_1|(g, d_1), f^*_2(g, \alpha), f^*_3(g, \alpha))
\end{aligned}
$$

Now equation (I) is direct from the definition of f^*_1. Equation (II) follows from equation (A). For equation (III), the most complicated one, we calculate as follows:

$$
\begin{array}{cll}
 & \delta_{\Delta_2}(f^*_{\Delta_2}(\alpha, g, d_1)\,,\, d_2) & \\
= & \delta_{\Delta_2}(\mathrm{ifz}_{\llbracket\Delta_2\rrbracket^*}(|f_1|(g, d_1), f^*_2(g, \alpha), f^*_3(g, \alpha))\,,\, d_2) & \\
= & \mathrm{ifz}_{\mathbf{N}}(|f_1|(g, d_1), \delta_{\llbracket\Delta_2\rrbracket}(f^*_2(g, \alpha), d_2), \delta_{\llbracket\Delta_2\rrbracket}(f^*_3(g, \alpha), d_2)) & \text{Nat'ty of ifz}\\
= & \mathrm{ifz}_{\mathbf{N}}(|f_1|(g, d_1), \delta_{\llbracket A\rrbracket}(\alpha, |f_2|(g, d_2)), \delta_{\llbracket A\rrbracket}(\alpha, |f_3|(g, d_2))) & \text{Eqn. } (B) \text{ and } (C)\\
= & \delta_{\llbracket A\rrbracket}(\alpha, \mathrm{ifz}_{|\llbracket A\rrbracket|}(|f_1|(g, d_1), |f_2|(g, d_2), |f_3|(g, d_3))) & \text{Nat'ty of ifz}\\
= & \delta_{\llbracket A\rrbracket}(\alpha, |f|(g, d_1, d_2)) &
\end{array}
$$

7.7 Linear recursion

Let A be $\mathsf{N}^k{\rightarrow}\mathsf{N}$. In order to interpret linrec_A we seek a global element of the Chu space

$$\Box\mathbf{N} \Rightarrow \llbracket A\rrbracket \Rightarrow (\Box\mathbf{N} \Rightarrow \llbracket A\rrbracket {\multimap} \llbracket A\rrbracket) \Rightarrow \llbracket A\rrbracket$$

Now Proposition 6.7 $(\mathbf{N}^k \Rightarrow \mathbf{N}){\multimap}(\mathbf{N}^k \Rightarrow \mathbf{N}) \cong \mathbf{N}^k \Rightarrow \mathbf{Lin}(\mathbf{N}^k)$
gives this is isomorphic to

$$\Box\mathbf{N} \Rightarrow \llbracket A\rrbracket \Rightarrow (\Box\mathbf{N} \Rightarrow \mathbf{N}^k \Rightarrow \mathbf{Lin}(\mathbf{N}^k)) \Rightarrow \llbracket A\rrbracket$$

Expanding the definitions a global element of the above Chu space amounts to a global element of the following sheaf in $\mathit{Sh}(\mathbb{B})$:

$$\Box\mathbf{N} \Rightarrow (\mathbf{N}^k \Rightarrow \mathbf{N}) \Rightarrow (\Box\mathbf{N} \Rightarrow \mathbf{N}^k \Rightarrow (\mathbf{N}^k \times (\mathbf{N} \Rightarrow \mathbf{N})) \Rightarrow \mathbf{N}^k \Rightarrow \mathbf{N}$$

Now up to isomorphism this coincides with the "type" of the recursor with parameter substitution $\mathsf{substrec}^{(k)}$ from Section 5.2 and it is this recursor composed with the described chain of isomorphisms which we use as interpretation for linrec_A.

8 Relating the interpretations

In order to finally deduce the desired result that all functions definable in SLR are *PTIME* we must relate the Chu space interpretation of SLR to its intended set-theoretic meaning. Let us introduce the notations $\llbracket - \rrbracket^{\mathbf{Chu}}$ and $\llbracket - \rrbracket^{\mathbf{Sets}}$ for these interpretations.

Let $\mathcal{G} : \mathbf{Chu}{\rightarrow}\mathbf{Sets}$ be the functor which sends Chu-space A to $|A|_{(0,0)}$ and similarly morphism $f : A{\rightarrow}B$ to $|f|_{(0,0)}$. Note that $\mathcal{G}(A) \cong \mathbf{Chu}(I, A)$.

If $f \in \mathcal{G}(A{\multimap}B)$ and $a \in \mathcal{G}(A)$ we define $app(f, a)$ as the application of the first component of f to a. More precisely, f comes as a pair $(|f|, f^*)$ where $|f| \in (|A| \Rightarrow |B|)_{(0,0)}$. We obtain $app(f, a)$ by applying $|f|$ to a.

We have $\mathcal{G}(A) = \mathcal{G}(\Box A) = \mathcal{G}(!A)$ hence $\mathcal{G}(\llbracket A \xrightarrow{a} B\rrbracket^{\mathbf{Chu}}) = \mathcal{G}(\llbracket A\rrbracket^{\mathbf{Chu}} {\multimap} \llbracket B\rrbracket^{\mathbf{Chu}})$ and $\mathcal{G}(A \otimes B) = \mathcal{G}(A) \times \mathcal{G}(B)$. Furthermore $\mathcal{G}(\mathbf{N}) \cong \mathbb{N}$.

Now, for each SLR-type A we define a relation

$$R^A \subseteq \mathcal{G}(\llbracket A\rrbracket^{\mathbf{Chu}}) \times \llbracket A\rrbracket^{\mathbf{Sets}}$$

by

$$xR^{\mathsf{N}}y \iff x = y$$
$$fR^{A\overset{a}{\to}B}g \iff \forall x{\in}\mathcal{G}([\![A]\!]^{\mathbf{Chu}}).\forall y{\in}[\![A]\!]^{\mathbf{Sets}}.\ xR^Ay \Rightarrow app(f,x)R^Bg(y)$$

Theorem 8.1 *Let* $\Gamma = x_1 \overset{a_1}{:} A_1, \dots, x_n \overset{a_n}{:} A_n$ *be a type assignment and assume* $\Gamma \vdash e : A$. *If* $x_i \in \mathcal{G}([\![A_i]\!]^{\mathbf{Chu}})$ *and* $y_i \in [\![A_i]\!]^{\mathbf{Sets}}$ *are such that* $x_i R^{A_i} y_i$ *for each* $i = 1 \dots n$ *then*

$$\mathcal{G}([\![e]\!]^{\mathbf{Chu}})(x_1, \dots, x_n)\ R^A\ [\![e]\!]^{\mathbf{Sets}}(y_1, \dots, y_n)$$

Proof. By induction on the derivation of $\Gamma \vdash e : A$. All cases except $e = \mathsf{linrec}_A$ are immediate. So let us prove that $[\![\mathsf{linrec}_A]\!]^{\mathbf{Chu}} R^{\tau(\mathsf{linrec}_A)} [\![\mathsf{linrec}_A]\!]^{\mathbf{Sets}}$ where

$$\tau(\mathsf{linrec}_A) = \Box\mathsf{N} \Rightarrow A \Rightarrow (\Box\mathsf{N} \Rightarrow A \multimap A) \Rightarrow A$$

and $A = \mathsf{N}^k \Rightarrow \mathsf{N}$. Assume that

$$\begin{array}{rcl} x & \in & \mathcal{G}(\Box\mathbf{N}) \cong \mathbf{N}_{(0,0)} \cong \mathbb{N} \\ g & \in & \mathcal{G}(A) \cong \mathbf{N}_{(0,k)} \\ h & \in & \mathcal{G}(\Box\mathbf{N} \Rightarrow A \multimap A) \cong \\ & & \mathcal{G}(\Box\mathbf{N} \Rightarrow \mathbf{N}^k \Rightarrow \mathbf{Lin}(\mathbf{N}^k)) \cong \\ & & \mathcal{G}(\Box\mathbf{N} \Rightarrow \mathbf{N}^k \Rightarrow (\mathbf{N}^k \times \mathbf{N} \Rightarrow \mathbf{N})) \cong \\ & & \mathbf{N}^k_{(1,k)} \times \mathbf{N}_{(1,k+1)} \\ y & \in & \mathcal{G}(\mathbf{N}^k) \cong \mathbb{N}^k \end{array}$$

and assume furthermore that

$$\begin{array}{rcl} x' & \in & \mathbb{N} \\ g' & \in & \mathbb{N}^k \to \mathbb{N} \\ h' & \in & \mathbb{N} \to (\mathbb{N}^k \to \mathbb{N}) \to (\mathbb{N}^k \to \mathbb{N}) \\ y' & \in & \mathbb{N}^k \end{array}$$

such that xRx', gRg', hRh', yRy' with the appropriate superscripts to R. Replacing x, g, h, y by their transpositions along the isomorphisms indicated above and denoting the components of h by $\vec{h}_1, h_2$ these assumptions amount to $x = x', g = g', y = y'$, and

$$h'(z, u, \vec{w}) = h_2(z, \vec{w}, u(\vec{h}_1(z, \vec{w})))$$

for each $z \in \mathbb{N}$, $u \in \mathbf{N}_{(0,k)}$, and $\vec{w} \in \mathbb{N}^k$.

We must show that $f(x) = f'(x')$ where $f, f' : \mathbb{N} \to \mathbb{N}^k \to \mathbb{N}$ are given by

$$\begin{array}{l} f(0, \vec{w}) = g(\vec{w}) \\ f(z, \vec{w}) = h_2(z, \vec{w}, f(\lfloor \frac{z}{2} \rfloor, \vec{h}_1(z, \vec{w}))),\ \text{if } x > 0 \\ f'(0, \vec{w}) = g'(\vec{w}) \\ f'(z, \vec{w}) = h'(z, \lambda\vec{w}.f(\lfloor \frac{z}{2} \rfloor, \vec{w}), \vec{w}) \end{array}$$

This follows by induction on z while maintaining the additional invariant that $\lambda\vec{w}.f(z, \vec{w}) \in \mathbf{N}_{(0,k)}$. □

Corollary 8.2 *If* $\vec{x}{:}\,\Box\mathsf{N}^m, \vec{y}{:}\,\mathsf{N}^n \vdash e : \mathsf{N}$ *then its set-theoretic interpretation is a polynomial time computable function.*

Proof. We have $f = |[\![\vdash e : \Box\mathsf{N}^m{\to}\mathsf{N}^n{\to}\mathsf{N}]\!]| : \Box\mathbf{N}^m \times \mathbf{N}^n \longrightarrow \mathbf{N}$. From the Yoneda Lemma we know that $f_{(0,0)} : \mathbb{N}^m \times \mathbb{N}^n{\to}\mathbb{N}$ is an (m,n)-polymax function (namely $f_{(m,n)}$ applied to the identity function). Theorem 8.1 on the other hand, tells us that $[\![e]\!]^{\mathbf{Sets}} = f$. $\Box$

9 Related work

There are several related approaches from some of which this work draws inspiration and over some of which it improves. The most important ones apart from Bellantoni-Cook's work are Leivant-Marion's work on tiered recursion, Caseiro's "don't double criticals" systems, and Girard's light linear logic. We discuss these works in order further down. Also related in the sense that category-theoretic methods are used to characterise complexity classes is Otto's work [19]. The difference to the present work is that categories are employed to describe the syntax as opposed to the semantics of systems of safe or tiered recursion. It appears that using Otto's presentation or an appropriate generalisation to higher order our results could be phrased more directly, e.g. without going through the rather laborious definition of an interpretation function. The disadvantage would then be that categories are then needed in the statement not only in the proof of the main result.

9.1 Tiered predicative recursion

Leivant-Marion's work [14] is based on a hierarchy of copies of natural number types N_0, N_1, N_2, ... If $x : \mathsf{N}_k$ then x is said to have tier k. In a primitive recursion the output of the function to be defined must be of a lower tier than the argument on which one recurs. It is shown that already two tiers suffice to represent all *PTIME*-functions, but natural definitions which are based on a composition of several auxiliary functions may require arbitrarily high tiers. So, when writing an auxiliary function one has to decide in advance which tier to assign to the arguments. It may well be that one could design a "tier inference scheme" for the Leivant-Marion system which would translate Bellantoni-Cook types to Leivant-Marion types, but the details remain to be worked out. Leivant and Marion have also studied primitive recursion with first-order functional result type [15] but reach the class PSPACE in this way. The reason is that the restrictions they impose rule out nested applications like in example 1.1 but still allow to call the recursive argument more than once so that typical PSPACE-complete functions such as evaluation of quantified boolean formulas (encoded as integers) can be programmed. It appears that by adding linearity to the Leivant-Marion framework one can also obtain a *PTIME* primitive recursion with first-order result type.

9.2 Caseiro's systems

Caseiro [5] studies an extension of the Bellantoni-Cook framework to arbitrary first-order data structures such as lists or trees. She notices that in the presence of binary constructors (like node in the case of binary trees) arbitrary duplication of safe arguments must be avoided. Accordingly several sets of conditions to avoid dangerous duplication of arguments are designed and it is shown that the resulting systems yield *PTIME* definitions. Unlike the present work Caseiro's systems are purely first-order. A disadvantage of her systems is that the syntactic conditions are fairly complex to state and look rather ad hoc. We believe that through the use of a type system based on modality and linear logic one could obtain a smoother formulation of Caseiro's work and integrate parts of it with the present work. See [11] for an attempt in this direction.

9.3 Light Linear Logic

Girard's Light Linear Logic (LLL) [8, 1] is a modification of his linear logic with restricted nonlinearisation ($!A \multimap A$ is no longer derivable) and an extra modality §. It is shown that this system admits cut elimination in *PTIME* and therefore all lambda terms typeable in this system can be reduced to normal form (by some strategy) in polynomial time. In particular it is shown that certain functions on Church numerals can be typed: letting int be the type $\forall X.!(X \multimap X) \multimap \S(X \multimap X)$, i.e. a linear logic decoration of the usual type of polymorphic tally integers in system F, then multiplication can be given the type $\mathsf{int} \multimap \mathsf{int} \multimap \S\S\mathsf{int}$ and more generally, for every (unary) *PTIME* function there exists a term of type $\mathsf{int} \multimap \S^k \mathsf{int}$ for some k. A similar characterisation exists for integers in binary notation. It is not clear, however, whether natural algorithms such as bitwise addition with carry can be represented in LLL.

The major advantage of LLL is that it contains polymorphic functions and—via impredicative encoding—arbitrary inductive datatypes like lists or trees. Its main disadvantage is its rather complex syntax and the lack of a semantic justification. Furthermore, although all polynomial time functions are expressible in LLL, the pragmatics, 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 systems based on safe recursion.

As LLL matures it might, however, supersede the present approach and also the work of Caseiro and Leivant-Marion. It remains to be seen whether the methods described in this paper and [10] could be used for such further development of LLL.

9.4 Safe recursion in all finite types

Since the submission of the first version of this paper considerable progress has been made on the matter of higher-order extensions of safe recursion both by the author [12] and (independently) by Bellantoni-Niggl-Schwichtenberg [3]. In these works, the main result of this paper is strengthened to linear recursion with arbitrary result types built up from $\mathbb{N}$ by $\multimap$. In [12] even other inductive datatypes like lists and binary trees together with the corresponding induction principles are added to the calculus.

The proof methods (linear combinatory algebras in [12] and a proof-theoretic analysis [3]) used in those works are very different from the one based on Chu spaces which we are going to present here and the Chu space method does not seem to scale up to those more general results. The advantage of the Chu space method lies in the fact that it provides an effective translation of higher-order linear recursion to first order recursion. We believe that this can serve as the basis of a semantics-based optimisation technique for compilation of higher-order recursive functional programs. Details remain to be worked out.

References

[1] Andrea Asperti. Light affine logic. In *Proc. Symp. Logic in Comp. Sci. (LICS)*. IEEE, 1998.

[2] S. Bellantoni. *Predicative recursion and computational complexity*. PhD thesis, University of Toronto, 192. Technical Report 264/92.

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

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

[5] 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.

[6] Peter Clote. Computation models and function algebras. available electronically under http://thelonius.tcs.informatik.uni-muenchen.de/~clote/Survey.ps.gz, 1996.

[7] 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.

[8] Jean-Yves Girard. Light Linear Logic. Manuscript. Available via http://hypatia.dcs.qmw.ac.uk/, 1995.

[9] 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.

[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. Linear types and non-size-increasing polynomial time computation. Draft. Available from www.dcs.ed.ac.uk/home/mxh/papers/icfp.ps.gz, October 1998.

[12] Martin Hofmann. Typed lambda calculi for polynomial-time computation, Forthcoming 1998. Habilitation thesis in preparation, TU Darmstadt, Germany.

[13] Yves Lafont and Thomas Streicher. Game semantics for linear logic. In Proc. 6th Annual IEEE Symp. on Logic in Computer Science, editor, *Logic in Computer Science (LICS)*, pages 43–49, 1991.

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

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

[16] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[17] Ieke Moerdijk and Saunders Mac Lane. *Sheaves in Geometry and Logic. A First Introduction to Topos Theory*. Springer, 1992.

[18] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[19] Jim Otto. *Complexity Doctrines*. PhD thesis, McGill University, 1995.

[20] V.R. Pratt. Chu spaces and their interpretation as concurrent objects. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Springer LNCS*, pages 392–405. Springer-Verlag, 1995.