# The Church-Scott representation of inductive and coinductive data

**Herman Geuvers**[1]

1 **ICIS, Radboud University**
**Heyendaalseweg 135 6525 AJ Nijmegen**
**The Netherlands**
**and**
**Faculty of Mathematics and Computer Science**
**Eindhoven University of Technology**
**The Netherlands**
herman@cs.ru.nl

## Abstract

Data in the lambda calculus is usually represented using the "Church encoding", which gives closed terms for the constructors and which naturally allows to define functions by *iteration*. An additional nice feature is that in system F (polymorphically typed lambda calculus) one can define types for this data and the iteration scheme is well-typed. A problem is that primitive recursion is not directly available: it can be coded in terms of iteration at the cost of inefficiency (e.g. a predecessor with linear run-time). The much less well-known Scott encoding has *case distinction* as a primitive. The terms are not typable in system F and there is no iteration scheme, but there is a constant time destructor (e.g. predecessor).

We present a unification of the Church and Scott definition of data types, the Church-Scott encoding, which has primitive recursion as basic. For the numerals, these are also known as 'Parigot numerals'. We show how these can be typed in the polymorphic lambda calculus extended with recursive types. We show that this also works for the dual case, co-inductive data types with primitive co-recursion.

We single out the major schemes for defining functions over data-types: the iteration scheme, the case scheme and the primitive recursion scheme, and we show how they are tightly connected to the Church, the Scott and the Church-Scott representation. We also single out the duals of these schemes: co-iteration, co-case and primitive co-recursion as schemes for defining function to a co-data-type.

A major advantage is that all these are encodings in pure untyped $\lambda$-calculus and that strong normalization is guaranteed, because the terms are typable in the polymorphic lambda calculus extended with recursive types.

**1998 ACM Subject Classification** F.1.1 Models of Computation, D.1.1 Applicative (Functional) Programming, D.3.3 Language Constructs and Features, F.4.1 Mathematical Logic, F.3.3 Studies of Program Constructs

**Keywords and phrases** lambda calculus, data types, (co)inductive types, iteration, primitive recursion

**Digital Object Identifier** 10.4230/LIPIcs.xxx.yyy.p

## 1 Introduction

There are various way to represent natural numbers in $\lambda$-calculus, the most well-known one being the Church representation, but there are various other, sometimes very eccentric, ones,


Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

e.g. as described in [2, 3, 23]. A notably very simple encoding of the natural numbers is know as the *Scott encoding*, which has been presented in 1963 by Scott [19], but was never published by Scott and only became known through [7]. It has been reinvented independently by [17] and it has recently regained interest in the functional programming community [20, 12] and also among researchers that use $\lambda$-calculus representations for the complexity analysis of functions [14, 5].

The Church representation can be generalized to many other data types, and these can be typed in system $F$ (polymorphic $\lambda$-calculus) and already quite a few functions can be well-typed [10, 4]. Already in simple type theory, the basic functions like addition and multiplication can be typed on the Church numerals [6]. On the other hand, the Scott representation of data can also be generalized, and the encoding is clearly simpler, but it's not very well-known and seldom used. Probably the main reason is that Scott data (e.g. the Scott numerals) are not readily typable in system $F$.

In the present paper we look at the various numeral systems from the point of view of *function definition schemes*. The most well-known function definition schemes are *iteration* and *primitive recursion* over the natural numbers.

The *iteration property for natural numbers* states that, if $d \in \mathbf{N}$ and $f : \mathbf{N} \to \mathbf{N}$, then there is a total function $h : \mathbf{N} \to \mathbf{N}$ that satisfies

$$\begin{aligned} h(0) &= d \\ h(n+1) &= f(h(n)) \end{aligned}$$

It can easily be proven that such $h$ indeed exists and is computable if $d$ and $f$ are. Actually, the equations give an algorithm for computing $h$ (given algorithms for computing $d$ and $f$). So, in a functional programming language, and especially in a strongly typed language where one wants all functions to be total, it is natural to add this iteration property as a primitive scheme: given a data-type for natural numbers nat, with constructors Zero and Succ we have the following *iteration scheme* for defining functions on nat

$$\frac{d : \mathrm{nat} \quad f : \mathrm{nat} \to \mathrm{nat}}{\mathrm{It}\, d\, f : \mathrm{nat} \to \mathrm{nat}}$$

with reduction rules

$$\begin{aligned} \mathrm{It}\, d\, f\, \mathrm{Zero} &\twoheadrightarrow_\beta d \\ \mathrm{It}\, d\, f\, (\mathrm{Succ}\, t) &\twoheadrightarrow_\beta f\, (\mathrm{It}\, d\, f\, t) \end{aligned}$$

The nice feature of the Church numerals, and Church data-types in general, that they basically are iterators: the $n$-th Church numeral just states: "take a value and a function and iterate the function $n$ times on the value". This is worked out in detail in [4].

The other well-known scheme is primitive recursion, which is, in the same computational phrasing

$$\frac{d : \mathrm{nat} \quad f : \mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}}{\mathrm{Rec}\, d\, f : \mathrm{nat} \to \mathrm{nat}}$$

with reduction rules

$$\begin{aligned} \mathrm{Rec}\, d\, f\, \mathrm{Zero} &\twoheadrightarrow_\beta d \\ \mathrm{Rec}\, d\, f\, (\mathrm{Succ}\, t) &\twoheadrightarrow_\beta f\, t\, (\mathrm{It}\, d\, f\, t) \end{aligned}$$

It is well-known that this scheme can be coded in terms of iteration in a slightly cumbersome way, so it would be desirable to have it as a primitive. In [8], it was argued that

the primitive recursion scheme is the most desirable to have, and the concept of *primitive recursion scheme* was extended to general inductive data types, and also to co-inductive data types, with a scheme of *primitive co-recursion*. To have these schemes, the simply typed $\lambda$-calculus, $\lambda\!\!\rightarrow$, and the polymorphically typed $\lambda$-calculus, system $F$, were extended with schemes for defining inductive (and co-inductive) types with primitive recursion (and primitive co-recursion), and it was shown that this system is strongly normalizing and has the same expressive power as a similar system by Mendler [16]. The (implicit) question was left open whether it would be possible to define data types with primitive recursion (and data-types with primitive co-recursion) directly in $\lambda$-calculus, typed or untyped.

Later, in [9], the scheme for primitive recursion was further broken down and it was shown that it is the combination of a *case scheme*, allowing simple pattern matching over a data-type, and the iteration scheme.

Here we discuss the three schemes: iteration, case and primitive recursion, and also their duals (co-iteration, co-case and primitive co-recursion) and we show that Church data-types support iteration, Scott data-types support case. We also introduce the new notion of *Church-Scott data-type* as a kind of union of the two that supports primitive recursion. Then we define Church, Scott and Church-Scott encodings of the dual: co-data-types and we show that they support co-iteration, co-case and primitive co-recursion, respectively.

An important feature of this work is that primitive recursion (and its dual, primitive co-recursion) can be encoded directly in pure untyped $\lambda$-calculus, without needing any additional constants or constructs. This simplifies the study of these schemes.

In addition, there is a typed $\lambda$-calculus, $\lambda 2\mu$ (see Appendix A), in which these data-types can be defined and these schemes are well-typed. The system $\lambda 2\mu$ extends system $F$ with recursive type definitions for positive type schemes, and is inspired from [1], where it was shown how to type the Scott numerals. The system $\lambda 2\mu$ can be shown to be strongly normalizing, by applying the standard saturated sets technique. This has been done for a slightly different system of recursive types in [16], but the approach is basically the same, so we don't repeat the proof here. This strongly normalization result guarantees the termination of all functions defined by the schemes we consider.

## 2 Natural Numbers

### 2.1 Church natural numbers

We look at the well-known definitions of data in the untyped $\lambda$-calculus. The most well-known are the Church numerals [6]:

$$\begin{array}{lcl@{\qquad}lcl} \overline{0} & := & \lambda x\, f.x & \overline{p} & := & \lambda x\, f.f^p\,(x) \\ \overline{1} & := & \lambda x\, f.f\, x & \overline{S} & := & \lambda n.\lambda x\, f.f\,(n\, x\, f) \\ \overline{2} & := & \lambda x\, f.f\,(f\, x) & & & \end{array}$$

The Church numerals have *iteration* as basis: the numerals are iterators. This means that for $c$ and $f$ terms, there is a term $\mathrm{It}\, c\, f$ satisfying

$$\begin{array}{rcl} \mathrm{It}\, d\, f\, \overline{0} & =_\beta & d \\ \mathrm{It}\, d\, f\, \overline{n+1} & =_\beta & f\,(\mathrm{It}\, d\, f\, n) \end{array}$$

Ideally, one would have $\twoheadrightarrow_\beta$ instead of $=_\beta$, which we could call a *reducing iterator*.

The Church numerals *are* the iterators themselves: $\mathrm{It}\, d\, f := \lambda n.n\, d\, f$ does the job. This doesn't give a reducing iterator, because one $\beta$-expansion is needed: $(\lambda n.n\, d\, f)\overline{p+1} \twoheadrightarrow_\beta$

$f\,(\overline{p}\,d\,f)$, which $\beta$-expands to $f\,(\mathrm{It}\,d\,f\,\overline{p})$. Also, one would like the iterator to not just work for closed values (actual numbers), but for any term of the shape $\overline{S}\,t$. We can make these requirements explicit by defining the iteration scheme for natural number as follows.

▶ **Definition 1.** The *iteration scheme for natural numbers* in untyped $\lambda$-calculus consists of closed terms Zero and Succ, and for all terms $d, f, t$ a term $\mathrm{It}\,d\,f\,t$ satisfying the following properties

$$\begin{aligned} \mathrm{It}\,d\,f\,\mathrm{Zero} &\twoheadrightarrow_\beta d \\ \mathrm{It}\,d\,f\,(\mathrm{Succ}\,t) &\twoheadrightarrow_\beta f\,(\mathrm{It}\,d\,f\,t) \end{aligned}$$

It is now easy to check that the Church numerals with Zero $:=$, Succ $:=$ and $\mathrm{It}\,d\,f\,t := t\,d\,f$ form an iteration scheme for the natural numbers. In a typed setting, this can be turned into a *typed iteration scheme* by requiring the terms to be well-typed.

▶ **Definition 2.** The *typed iteration scheme for natural numbers* in some ambient typed $\lambda$-calculus extends the notion of iteration scheme for the natural numbers by a type nat and the following typing conditions: Zero : nat, Succ : nat $\to$ nat and

$$\frac{d : D \quad f : D \to D \quad n : \mathrm{nat}}{\mathrm{It}\,d\,f\,n : D}$$

for any type $D$.

and terms $d$ and $f$ of the appropriate types, $\mathrm{It}\,d\,f$ is of type nat $\to D$, where nat is a type for the numerals $\overline{n}$.

The advantage of the iteration scheme is that one gets quite a bit of "well-founded recursion" for free: many well-known functions (addition, multiplication, exponentiation) fit in the iteration scheme. Also the Ackermann function can be defined using the iteration scheme, in case one also allows higher types (types containing arrows) for $D$. In a typed setting where the defining equations ($=_\beta$) for the iteration scheme are actually reductions ($\twoheadrightarrow_\beta$), the normalization property for the type system implies the termination of algorithms defined using the iteration scheme.

A well-known result of Girard [10] is that in system F (the polymorphic lambda calculus), there is a typed iteration scheme for natural numbers: nat can be defined as $\forall\alpha.\alpha\to(\alpha\to\alpha)\to\alpha$ and then the terms above are all well-typed. Böhm and Berarducci [4] have shown how to extend this to arbitrary inductive data types, giving a general procedure for defining a closed system $F$ type for a data type, closed terms for its constructors and an iteration scheme.

A disadvantage of the iteration scheme is that there is no pattern matching built in, so the predecessor is hard to define. This was already known to Church, who seems to have been a bit dissatisfied with these numerals. Kleene was the first to discover a definition for the predecessor on the Church numerals, which he found during a visit of his dentist. The idea is to define by considering the function $F : \mathrm{nat} \times \mathrm{nat} \to \mathrm{nat} \times \mathrm{nat}$, with $F(\langle x, y\rangle) = \langle \mathrm{Succ}\,x, x\rangle$. If one takes the $n$-times iteration of $F$ on $\langle \mathrm{Zero}, \mathrm{Zero}\rangle$ and then the second component of the pair, one obtains $\overline{n-1}$ (and Zero if $n = 0$). So $\mathrm{Pred} := \lambda n.(\mathrm{It}\,\langle \mathrm{Zero}, \mathrm{Zero}\rangle\,\lambda z.\langle \mathrm{Succ}\,z_1, z_1\rangle)_2$, where $(-)_1$ and $(-)_2$ denote first and second projection from a pair. This trick applies generally to define *destructors* (the inverse of the constructors) for data types in the (typed) $\lambda$-calculus, as has been shown, e.g. in [8]. The problem is:

- The predecessor function only works for *closed values*: $\mathrm{Pred}\,\overline{n+1} =_\beta \overline{n}$, but not $\mathrm{Pred}\,(\mathrm{Succ}\,x) =_\beta x$.

- The predecessor does not compute in constant time: $\mathrm{Pred}\,\overline{n+1} \twoheadrightarrow_\beta \overline{n}$ in a number of steps linear in $n$. (Basically, Pred counts down to 0, keeping track of the number of steps it takes to do that, and then it counts up one step less.)

These problems are the same for all inductive data type definitions in typed or untyped $\lambda$-calculus that follow the "Church style approach". See [8] for details.

## 2.2 Scott natural numbers

The Scott numerals haven't actually been published by Scott himself, but they appear in [7], where they are attributed to Scott [19]. The definition of the numerals is as follows.

$$\begin{array}{rcl} \underline{0} & := & \lambda x\, f.x \\ \underline{1} & := & \lambda x\, f.f\,\underline{0} \\ \underline{2} & := & \lambda x\, f.f\,\underline{1} \end{array} \qquad\qquad \begin{array}{rcl} \underline{p+1} & := & \lambda x\, f.f\,\underline{p} \\ \underline{S} & := & \lambda n.\lambda x\, f.f\,n \end{array}$$

The Scott numerals have *case distinction* as a basis: the numerals are *case distinctors*. For clarity and comparison, we single out case distinction as a function definition scheme, just like iteration in Definition 1.

▶ **Definition 3.** The *case scheme for natural numbers* in untyped $\lambda$-calculus consists of closed terms Zero and Succ, and for all terms $d, f, t$ a term $\mathrm{It}\,d\,f\,t$ satisfying the following properties

$$\begin{array}{rcl} \mathrm{Case}\,d\,f\,\mathrm{Zero} & \twoheadrightarrow_\beta & d \\ \mathrm{Case}\,d\,f\,(\mathrm{Succ}\,t) & \twoheadrightarrow_\beta & f\,t \end{array}$$

An advantage is that the predecessor can now immediately be defined, by taking $d = \mathrm{Zero}$ and $f = \lambda x.x$. However one has to get recursion from somewhere else, because the case scheme doesn't provide a recursive call to the function $\mathrm{Case}\,d\,f$. In untyped $\lambda$-calculus, recursion can be obtained from a fixed point-combinator.

In analogy with Definition 2, we also define a typed case scheme.

▶ **Definition 4.** The *typed case scheme for natural numbers* in some ambient typed $\lambda$-calculus extends the notion of case scheme by a type nat and the following typing conditions: $\mathrm{Zero} : \mathrm{nat}$, $\mathrm{Succ} : \mathrm{nat} \to \mathrm{nat}$ and

$$\frac{d : D \quad f : \mathrm{nat} \to D \quad n : \mathrm{nat}}{\mathrm{Case}\,d\,f\,n : D}$$

for any type $D$.

## 2.3 Combined Church-Scott natural numbers

The thing missing from the Church numerals is that they do not directly provide definitions by *primitive recursion*, as one has, e.g. in Gödel's system $T$. To make this explicit, we define the scheme for primitive recursion.

▶ **Definition 5.** The *primitive recursion scheme for natural numbers* in untyped $\lambda$-calculus consists of closed terms Zero and Succ, and for all terms $d, f, t$ a term $\mathrm{Rec}\,d\,f\,t$ satisfying the following properties

$$\begin{array}{rcl} \mathrm{Rec}\,d\,f\,\mathrm{Zero} & \twoheadrightarrow_\beta & d \\ \mathrm{Rec}\,d\,f\,(\mathrm{Succ}\,t) & \twoheadrightarrow_\beta & f\,t\,(\mathrm{Rec}\,d\,f\,t) \end{array}$$

The *typed primitive recursion scheme for natural numbers* in some ambient typed $\lambda$-calculus extends this by a type nat and the following typing conditions: Zero : nat, Succ : nat $\to$ nat and

$$\frac{d : D \quad f : \mathrm{nat} \to D \to D \quad n : \mathrm{nat}}{\mathrm{Rec}\, d\, f\, n : D}$$

The primitive recursion scheme captures both the iteration scheme and the case scheme, so both the predecessor and the iterative functions like addition and multiplication can be defined directly.

The following definition of the natural numbers also occurs in [18] and is therefore also called the *Parigot numerals*, e.g. in [20].

▶ **Definition 6.** The combined *Church-Scott numerals* in the untyped $\lambda$-calculus are defined as follows.

$$\begin{array}{rcl} \underline{\overline{0}} & := & \lambda x\, f.x \\ \underline{\overline{1}} & := & \lambda x\, f.f\, \underline{\overline{0}}\, (\underline{\overline{0}}\, x\, f) \\ \underline{\overline{2}} & := & \lambda x\, f.f\, \underline{\overline{1}}\, (\underline{\overline{1}}\, x\, f) \end{array} \qquad \begin{array}{rcl} \underline{\overline{p+1}} & := & \lambda x\, f.f\, \underline{\overline{p}}\, (\underline{\overline{p}}\, x\, f) \\ \underline{\overline{S}} & := & \lambda n.\lambda x\, f.f\, n\, (n\, x\, f) \end{array}$$

These numerals can be typed if one allows *recursive types*: Suppose we extend system $F$ with the possibility to define types by recursive equations $\sigma := \mu X.\Phi(X)$, under the condition that $X$ should occur positively in $\Phi(X)$. This definition amounts to a type equality $\sigma = \Phi(\sigma)$. Then we can define

$$\mathrm{nat} := \mu Y.\forall X.X \to (Y \to X \to X) \to X.$$

Now the Church-Scott numerals can be typed with type nat. As illustration we give the typing of the successor.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{n : \mathrm{nat}, x : X, f : \mathrm{nat} \to X \to X \vdash n : X \to (\mathrm{nat} \to X \to X) \to X}{n : \mathrm{nat}, x : X, f : \mathrm{nat} \to X \to X \vdash n\, x\, f : X}}{n : \mathrm{nat}, x : X, f : \mathrm{nat} \to X \to X \vdash f\, n\, (n\, x\, f) : X}}{n : \mathrm{nat} \vdash \lambda x\, f.f\, n\, (n\, x\, f) : X \to (\mathrm{nat} \to X \to X) \to X}}{n : \mathrm{nat} \vdash \lambda x\, f.f\, n\, (n\, x\, f) : \forall X.X \to (\mathrm{nat} \to X \to X) \to X}}{n : \mathrm{nat} \vdash \lambda x\, f.f\, n\, (n\, x\, f) : \mathrm{nat}}}{\vdash \lambda n.\lambda x\, f.f\, n\, (n\, x\, f) : \mathrm{nat} \to \mathrm{nat}}$$

▶ **Lemma 7.** *The Church-Scott numerals admit a primitive recursion scheme, given by* $\mathrm{Rec}\, n\, d\, f := n\, d\, f$. *In system F with positive recursive types, there is is a typed primitive recursion scheme.*

**Proof.** For the reduction requirements:

$$\begin{array}{rcl} \mathrm{Rec}\, d\, f\, \mathrm{Zero} & := & \mathrm{Zero}\, d\, f \twoheadrightarrow_\beta d \\ \mathrm{Rec}\, d\, f\, (\mathrm{Succ}\, t) & := & \mathrm{Succ}\, t\, d\, f \twoheadrightarrow_\beta f\, t\, (t\, d\, f) \equiv f\, t\, (\mathrm{Rec}\, d\, f\, t) \end{array}$$

For the typing:

$$\frac{d : D, f : \mathrm{nat} \to D \to D, n : \mathrm{nat} \vdash n : \forall X.X \to (\mathrm{nat} \to X \to X) \to X}{d : D, f : \mathrm{nat} \to D \to D, n : \mathrm{nat} \vdash n\, d\, f : D}$$

◀

## 3 Categorical data types

Data types in functional languages are often derived from ideas in categorical semantics. The first to make that precise is Hagino ([11]) who derives from the notions of initial algebra and terminal co-algebra an extension of simply typed lambda calculus, which he calls *categorical data types*. This amounts to two schemes for defining a new type from a co-variant functor from types to types. These come together with constants and reduction rules.

Here we repeat this general definition, using the presentation of [8]. This amounts to *inductive data types* that enjoy an iteration scheme and is therefore not completely satisfactory from a computational point of view. For example, the destructor (the inverse of the constructors of the data type) is hard to define. So, we follow [8] and we introduce the notion of *recursive algebra*, which is a data type that enjoys primitive recursion. In parallel, we give the categorical diagrams that serve as a motivation and we also introduce the dual notions of *co-inductive data type* that enjoys *co-iteration* and *co-recursive co-algebra* that enjoys *primitive co-recursion*.

We start of with some preliminaries.

A co-variant functor from simple types to simple types is a *positive type scheme* $\Phi(\alpha)$, that is a type $\Phi$ in which the free variable $\alpha$ occurs *positively*. (The type variable $\alpha$ occurs *positively* in the type $\Phi$ if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi \equiv \alpha$ or if $\Phi \equiv \Phi_1 \to \Phi_2$ and $\alpha$ occurs negatively in $\Phi_1$, positively in $\Phi_2$. The type variable $\alpha$ occurs *negatively* in $\Phi$ if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi \equiv \alpha$ or if $\Phi \equiv \Phi_1 \to \Phi_2$ and $\alpha$ occurs negatively in $\Phi_2$, positively in $\Phi_1$.) If $\Phi(\alpha)$ is a type scheme, with $\Phi(\tau)$ we mean the type $\Phi$ with $\tau$ substituted for $\alpha$. If there's no ambiguity to which type variable $\alpha$ we're referring, we just write $\Phi$ in stead of $\Phi(\alpha)$.

A positive, respectively negative type scheme $\Phi$ can be applied to a function $f : \tau \to \rho$, obtaining $\Phi(f) : \Phi(\rho) \to \Phi(\tau)$, respectively $\Phi(f) : \Phi(\tau) \to \Phi(\rho)$ by *lifting*: $\alpha(f) \equiv f$, if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi(f) \equiv id_\Phi$ and if $\alpha$ occurs negatively in $\Phi_1$, positively in $\Phi_2$ then

$$\begin{aligned}(\Phi_1 \to \Phi_2)(f) &\equiv \lambda x : \Phi_1(\rho) \to \Phi_2(\rho).\lambda y : \Phi_1(\tau).\Phi_2(f)(x(\Phi_1(f)y)),\\ (\Phi_2 \to \Phi_1)(f) &\equiv \lambda x : \Phi_2(\tau) \to \Phi_1(\tau).\lambda y : \Phi_2(\rho).\Phi_1(f)(x(\Phi_2(f)y)).\end{aligned}$$

These notions extend naturally to other type theories, like system $F$ or $\lambda 2\mu$ that we will consider.

▶ **Definition 8.** Let $\Phi_1(\alpha), \Phi_2(\alpha), \ldots, \Phi_n(\alpha)$ be positive type schemes. The *inductive type scheme* for constructing data types is the following.

$\sigma :=$ **inductive** $\alpha$ **with constructors**

$c_1 : \Phi_1(\alpha) \to \alpha$

$c_2 : \Phi_2(\alpha) \to \alpha$

$\vdots$

$c_n : \Phi_n(\alpha) \to \alpha$

**end**

This declaration gives rise to the following extension of the system.

1. a closed type $\sigma$
2. constants $c_i : \Phi_i(\sigma) \to \sigma$ for $1 \leq i \leq n$,
3. terms $\mathrm{It}\, M_1\, M_2 \ldots M_n$ for every $M_1, M_2, \ldots, M_n$ such that
4. for every $\tau$,

$$\frac{M_1 : \Phi_1(\tau) \to \tau \quad M_2 : \Phi_2(\tau) \to \tau \quad \ldots \quad M_n : \Phi_n(\tau) \to \tau}{\mathrm{It}\, M_1\, M_2 \ldots M_n : \sigma \to \tau}$$

5. The reduction relation is extended with the rule

$$\mathrm{It}\, M_1\, M_2 \ldots M_n\, (c_i t) \twoheadrightarrow M_i\, (\Phi_i(\mathrm{It}\, M_1 \ldots M_n)\, t)$$

The iterator is also called *eliminator*, as it eliminates an object of an inductive type and its computation rule matches on a constructor of the inductive type.

The natural numbers are an example of an inductive data-type, if we also allow $\Phi(\alpha)$ to be absent, or if we have a unity type in our language. We go for the first option and then we have

nat := **inductive** $\alpha$ **with constructors**
$c_1$ : $\alpha$
$c_2$ : $\alpha \to \alpha$
**end**

For $d : \tau$, $f : \tau \to \tau$, the reduction rule for $\mathrm{It}\, d\, f$ is

$$\begin{array}{rcl} \mathrm{It}\, M_1\, M_2\, c_1 & \twoheadrightarrow & M_1 \\ \mathrm{It}\, M_1\, M_2\, (c_2\, t) & \twoheadrightarrow & M_2\, (\mathrm{It}\, M_1 \ldots M_n\, t) \end{array}$$

which are the rules of the iteration scheme for nat.

▶ **Definition 9.** Let $\Phi_1(\alpha), \Phi_2(\alpha), \ldots, \Phi_n(\alpha)$ be positive type schemes. The *co-inductive type scheme* for constructing data types is the following.

$\sigma$ := **inductive** $\alpha$ **with constructors**
$d_1$ : $\alpha \to \Phi_1(\alpha)$
$d_2$ : $\alpha \to \Phi_1(\alpha)$
⋮
$d_n$ : $\alpha \to \Phi_1(\alpha)$
**end**

This declaration gives rise to the following extension of the system.

1. a closed type $\sigma$
2. constants $c_i : \sigma \to \Phi_i(\sigma)$ for $1 \leq i \leq n$,
3. terms $\mathrm{CoIt}\, M_1\, M_2 \ldots M_n\, d$ for every $M_1, M_2, \ldots, M_n$ such that
4. for every $\tau$,

$$\frac{M_1 : \tau \to \Phi_1(\tau) \quad M_2 : \tau \to \Phi_2(\tau) \quad \ldots \quad M_n : \tau \to \Phi_n(\tau)}{\mathrm{CoIt}\, M_1\, M_2 \ldots M_n : \tau \to \sigma}$$

5. The reduction relation is extended with the rule

$$d\, (\mathrm{CoIt}\, M_1\, M_2 \ldots M_n\, t) \twoheadrightarrow \Phi_i(\mathrm{CoIt}\, M_1 \ldots M_n)\, (M_i\, t)$$

An interesting example of a co-inductive type is the type of *streams over some base type* $A$, $\mathrm{Str}_A$. It is given by.

$\mathrm{Str}_A$ := **co-inductive** $\alpha$ **with destructors**
hd : $\alpha \to A$
tl : $\alpha \to \alpha$
**end**

Then $\mathrm{hd} : \mathrm{Str}_A \to A$ and $\mathrm{tl} : \mathrm{Str}_A \to \mathrm{Str}_A$ and

$$\frac{M_1 : \tau \to A \quad M_2 : \tau \to \tau}{\mathrm{CoIt}\, M_1\, M_2 : \tau \to \mathrm{Str}_A}$$

with

$$\begin{aligned} \mathrm{hd}(\mathrm{CoIt}\, M_1\, M_2\, t) &\twoheadrightarrow M_1 t \\ \mathrm{tl}(\mathrm{CoIt}\, M_1\, M_2\, t) &\twoheadrightarrow \mathrm{CoIt}\, M_1\, M_2\, (M_2\, t) \end{aligned}$$

We single this out as a definition scheme for functions *to* streams.

▶ **Definition 10.** A type $\mathrm{Str}_A$ enjoys the *co-iteration scheme for streams* if there are terms $\mathrm{hd} : \mathrm{Str}_A \to A$, $\mathrm{tl} : \mathrm{Str}_A \to \mathrm{Str}_A$ and $\mathrm{CoIt}\, M_1\, M_2\, t$ (for any $M_1$, $M_2$ and $t$) such that

$$\frac{M_1 : \tau \to A \quad M_2 : \tau \to \tau \quad t : \tau}{\mathrm{CoIt}\, M_1\, M_2\, t : \mathrm{Str}_A}$$

with reductions

$$\begin{aligned} \mathrm{hd}(\mathrm{CoIt}\, M_1\, M_2\, t) &\twoheadrightarrow M_1 t \\ \mathrm{tl}(\mathrm{CoIt}\, M_1\, M_2\, t) &\twoheadrightarrow \mathrm{CoIt}\, M_1\, M_2\, (M_2\, t) \end{aligned}$$

This allows to define, for example, the stream of all $a$'s (for some $a : A$) as $\mathrm{CoIt}\, (\lambda x.a)\, (\lambda x.a)\, a$, or if we take $A := \mathrm{nat}$, $\lambda n.\mathrm{CoIt}\, (\lambda x.x)\, \mathrm{Succ}\, n$ define the function that, given $n : \mathrm{nat}$, returns the stream $n, n+1, n+2, \ldots$.

▶ Remark. The type $\sigma$ defined by the inductive type scheme from $\Phi_1(\alpha), \ldots \Phi_n(\alpha)$ should be read as a (weakly) initial algebra of $T(X) = \Phi_1(X) + \ldots + \Phi_n(X)$. The type $\sigma$ defined by the co-inductive type scheme from $\Phi_1(\alpha), \ldots \Phi_n(\alpha)$ should be read and (weakly) terminal co-algebras of $T(X) = \Phi_1(X) \times \ldots \times \Phi_n(X)$. So dualising is of course not the same as reversing all the arrows in a sum scheme to obtain a product scheme!

The inductive type scheme and the co-inductive type scheme correspond to the categorical notions of initial algebra and terminal co-algebra. In [8], these have been refined to *recursive algebra* and *co-recursive co-algebra*. We introduce these notions, because those are the one we wish to capture syntactically. For further discussion we refer to [8].

In the following, we assume we are in some category $C$ that has products and co-products, and that $T$ is a functor in this category.

▶ **Definition 11.** 1. A $T$-*algebra* is a pair $(A, f)$, with $A$ an object and $f : TA \to A$.

2. If $(A, f)$ and $(B, g)$ are $T$-algebra's, *a morphism from* $(A, f)$ *to* $(B, g)$ is a morphism $h : A \to B$ such that the following diagram commutes.

$$\begin{array}{ccc} T(A) & \xrightarrow{f} & A \\ {\scriptstyle T(h)}\downarrow & = & \downarrow{\scriptstyle h} \\ T(B) & \xrightarrow[g]{} & B \end{array}$$

3. A $T$-algebra $(A, f)$ is *initial* if it is initial in the category of $T$-algebras, i.e. for every $T$-algebra $(B, g)$ there's a unique $h$ which makes the above diagram commute.

The initial algebra of the functor $T(X) = 1 + X$ is the natural numbers object. The initial algebra of $T(X) = 1 + (A \times X)$ is the object of finite lists over $A$. In a typed $\lambda$-calculus like system $F$ one can define *weakly initial algebras* for a positive type scheme $T(\alpha)$, that is: we have $T$-algebra $(A, f)$ such that for every $T$-algebra $(B, g)$ there is a map $h$ which makes the diagram of Definition 11 commute, but this $h$ is not necessarily unique. As a matter of fact, for an inductive type, $\mathrm{It}\, g$ is an $h$ that makes the diagram commute.

▶ **Definition 12.** 1. A *$T$-co-algebra* is a pair $(A, f)$, with $A$ an object and $f : A \to TA$.
2. If $(A, f)$ and $(B, g)$ are $T$-co-algebras, *a morphism from* $(B, g)$ *to* $(A, f)$ is a morphism $h : B \to A$ such that the following diagram commutes.

$$\begin{array}{ccc} B & \xrightarrow{g} & T(B) \\ {\scriptstyle h}\downarrow & = & \downarrow{\scriptstyle T(h)} \\ A & \xrightarrow[f]{} & T(A) \end{array}$$

3. A $T$-co-algebra $(A, f)$ is *terminal* if it is terminal in the category of $T$-co-algebras, i.e. for every co-algebra $(B, g)$ there's a unique $h$ which makes the above diagram commute.

Our pet example for terminal co-algebras is the one for $T(X) = A \times X$, the object of streams over $A$.

Again, in a typed $\lambda$-calculus like system $F$, one can only define *weakly terminal co-algebras*, where for every $T$-co-algebra, $(B, g)$ there exists an arrow $h$ that makes the diagram in 12 commute, but this need not be unique. A co-inductive type is a weakly terminal co-algebra: $\mathrm{CoIt}\, g$ is an $h$ that makes the diagram commute.

Let in the following $C$ be a category with weak products and weak co-products and $T$ a functor from $C$ to $C$.

▶ **Definition 13.** $(A, f)$ is a *recursive $T$-algebra* if $(A, f)$ is a $T$-algebra and for every $g : T(X \times A) \to X$ there exists an $h : A \to X$ such that the following diagram commutes.

$$\begin{array}{ccc} TA & \xrightarrow{f} & A \\ {\scriptstyle T(\langle h, \mathrm{id}\rangle)}\downarrow & = & \downarrow{\scriptstyle h} \\ T(X \times A) & \xrightarrow[g]{} & X \end{array}$$

▶ **Definition 14.** $(A, f)$ is a *co-recursive $T$-co-algebra* if $(A, f)$ is a $T$-co-algebra and for every $g : X \to T(X + A)$ there exists an $h : X \to A$ such that the following diagram commutes.

$$\begin{array}{ccc} X & \xrightarrow{g} & T(X + A) \\ {\scriptstyle h}\downarrow & = & \downarrow{\scriptstyle T([h, \mathrm{id}])} \\ A & \xrightarrow[f]{} & TA \end{array}$$

In [9], the notions of *algebra with case* and co-algebra with co-case have been introduced, to further refine the notions of (co-)recursive (co-)algebra.

# 4 Church and Scott data types in type theory

## 4.1 Inductive data types

We now give a general representation of inductive data types in $\lambda$-calculus, in Church, Scott and Church-Scott style. The first is well-known from [4], but we give it for completeness.

Church style data types enjoy the iteration scheme, while Scott style data types enjoy the case scheme and Church-Scott data types enjoy the primitive recursion scheme.

To keep the presentation simple, we limit our attention to first order data types in this paper, so there are no nested arrows.

▶ **Definition 15.** A *first order data type* will be written as

$$\begin{array}{lll} \textbf{data-type} \quad D \quad \text{with constructors} & & \\ & \mathbf{C}_1^D & : \quad T_1^1(D) \to \ldots \to T_{\mathsf{ar}(1)}^1(D) \to D \\ & & \ldots \\ & \mathbf{C}_n^D & : \quad T_1^n(D) \to \ldots \to T_{\mathsf{ar}(n)}^n(D) \to D \end{array}$$

where each of the $T_j^i(X)$ is either $X$ or a type expression that does not contain $X$. If $D$ is clear from the context, we will omit it as a superscript and write $\mathbf{C}_i$ in stead of $\mathbf{C}_i^D$.

This defines an algebraic data-type $D$ with $n$ constructors with names $\mathbf{C}_1, \ldots, \mathbf{C}_n$. Each constructor $\mathbf{C}_i$ has arity $\mathsf{ar}(i)$, which can also be 0, and then the constructor is a constant.

▶ **Convention 16.** To simplify notation, we abbreviate the list of argument types of a constructor, writing $T^1(X)$ for $T_1^1(X) \ldots T_{\mathsf{ar}(1)}^1(X)$ etc. As a matter of fact, this is a kind of uncurrying, replacing $\mathbf{C}_1 : T_1^1(D) \to \ldots \to T_{\mathsf{ar}(1)}^1(D) \to D$ by $\mathbf{C}_1 : T_1^1(D) \times \ldots \times T_{\mathsf{ar}(1)}^1(D) \to D$, and then we abbreviate $T_1^1(X) \times \ldots \times T_{\mathsf{ar}(1)}^1(X)$ to $T^1(X)$. So then a first order data type looks like

$$\begin{array}{lll} \textbf{data-type} \quad D \quad \text{with constructors} & & \\ & \mathbf{C}_1 & : \quad T^1(D) \to D \\ & & \ldots \\ & \mathbf{C}_n & : \quad T^n(D) \to D \end{array}$$

In semantical terms, an inductive data type should capture the least fixed point of $X \mapsto T^1(X) + \ldots + T^n(X)$, or equivalently the initial algebra of the functor $\Phi$, where $\Phi(X) = T^1(X) + \ldots + T^n(X)$.

We can now give a general definition of Church, Scott and Church-Scott data types in the typed $\lambda$-calculus.

▶ **Definition 17.** Given a data-type $D$ as in Definition 15, with constructors $\mathbf{C}_1, \ldots, \mathbf{C}_n$, the *Church encoding* of $D$ in system $F$ is given by

$$\overline{D} := \forall\alpha.(T^1(\alpha){\to}\alpha){\to}\ldots{\to}(T^n(\alpha){\to}\alpha){\to}\alpha.$$

The constructors are encoded by

$$\begin{array}{rcll} \overline{\mathbf{C}_i} & := & \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\, t_1 \ldots t_{\mathsf{ar}(i)} & \\ \text{with } t_j & := & x_j & \text{if} \quad \alpha \notin \mathrm{FV}(T_j^i(\alpha)) \\ t_j & := & x_j\, \vec{c} & \text{if} \quad T_j^i(\alpha) = \alpha \end{array}$$

The simplest instance is the natural numbers: $\overline{0} := \lambda x\, f.x$ and $\overline{S} := \lambda n.\lambda x\, f.f\,(n\, x\, f)$. It is well-known that the Church encoding enjoys the iteration scheme, which is just the It rule that we have given in general in 15.

We can encode Scott data types in $\lambda 2\mu$. (See Appendix A for a description of the system $\lambda 2\mu$.)

▶ **Definition 18.** Given a data-type $D$ as in Definition 15, with constructors $\mathbf{C}_1, \ldots, \mathbf{C}_n$, the *Scott encoding* of $D$ in system $\lambda 2\mu$ is given by

$$\underline{D} := \mu\beta.\forall\alpha.(T^1(\beta)\to\alpha)\to\ldots\to(T^n(\beta)\to\alpha)\to\alpha.$$

The constructors are encoded by

$$\underline{\mathbf{C}_i} \quad := \quad \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_n.c_i\, x_1 \ldots x_{\mathsf{ar}(i)}$$

The most well-known instance is again the natural numbers: $\underline{0} := \lambda x\, f.x$, $\underline{S} := \lambda n.\lambda x\, f.f\, n$. The Scott data types enjoy a case definition scheme. An important instance of this is that one can define the inverse of the constructors, a *destructor*. We single this out in a separate Lemma.

▶ **Definition 19.** Given the Scott encoding of a data-type $D$, the *destructor* is defined by

$$\begin{array}{lcl} \mathrm{dest} & : & \underline{D} \to T^1(\underline{D}) + \ldots + T^n(\underline{D}) \\ & := & \lambda d.d\,\widehat{\mathrm{in}}_1 \ldots \widehat{\mathrm{in}}_n \end{array}$$

where

$$\widehat{\mathrm{in}}_i : T_1^i(\underline{D}) \to \ldots \to T_{\mathsf{ar}(i)}^i(\underline{D}) \to T^1(\underline{D}) + \ldots + T^n(\underline{D})$$

is the curried version of $\mathrm{in}_i : T^i(\underline{D}) \to T^1(\underline{D}) + \ldots + T^n(\underline{D})$,

$$\widehat{\mathrm{in}}_i := \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\mathrm{in}_i\langle x_1, \ldots, x_{\mathsf{ar}(i)}\rangle.$$

▶ **Lemma 20.** *Given the Scott encoding of a data-type $D$, the destructor satisfies the property*

$$\mathrm{dest}(\underline{\mathbf{C}_i}\, x_1 \ldots x_{\mathsf{ar}(i)}) \twoheadrightarrow_\beta \mathrm{in}_i\langle x_1 \ldots x_{\mathsf{ar}(i)}\rangle$$

**Proof.**

$$\mathrm{dest}(\underline{\mathbf{C}_i}\, x_1 \ldots x_{\mathsf{ar}(i)}) \twoheadrightarrow_\beta \widehat{\mathrm{in}}_i\, x_1 \ldots x_{\mathsf{ar}(i)} \twoheadrightarrow_\beta \mathrm{in}_i\langle x_1 \ldots x_{\mathsf{ar}(i)}\rangle$$

◀

▶ **Lemma 21.** *The Scott encoding of a data-type $D$ enjoys the* case scheme*, given by*

$$\frac{f_1 : T^1(\underline{D})\to\tau \qquad f_n : T^n(\underline{D})\to\tau \qquad d : \underline{D}}{\mathrm{Case}\, f_1 \ldots f_n\, d : \tau}$$

*and the following reduction rule*

$$\mathrm{Case}\, f_1 \ldots f_n\, (\underline{\mathbf{C}_i}\, x_1 \ldots x_{\mathsf{ar}(i)}) \twoheadrightarrow_\beta f_i\, x_1 \ldots x_{\mathsf{ar}(i)}$$

**Proof.** Define $\mathrm{Case}\, f_1 \ldots f_n\, d := d\, f_1 \ldots f_n$. Then the reduction is immediate. ◀

The Scott encoding is sometimes also called the Mogensen-Scott encoding, because Mogensen [17] used it to give an (amazingly simple and perspicuous) encoding of untyped $\lambda$-calculus in itself. We will refer to it as the Scott-encoding.

We can even do better, by defining a Church-Scott encoding of a data-type.

▶ **Definition 22.** Given a data-type $D$ as in Definition 15, with constructors $\mathbf{C}_1, \ldots, \mathbf{C}_n$, the *Church-Scott encoding* of $D$ in system $\lambda 2\mu$ is given by

$$\overline{\underline{D}} := \mu\beta.\forall\alpha.(T^1(\beta\times\alpha)\to\alpha)\to\ldots\to(T^n(\beta\times\alpha)\to\alpha)\to\alpha.$$

The constructors are encoded by

$$\begin{array}{rcll} \overline{\underline{\mathbf{C}_i}} & := & \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\, t_1 \ldots t_{\mathsf{ar}(i)} & \\ \text{with } t_j & := & x_j & \text{if} \quad \alpha \notin \mathrm{FV}(T_j^i) \\ t_j & := & \langle x_j, x_j\, \vec{c}\rangle & \text{if} \quad T_j^i = \alpha \end{array}$$

The most well-known instance is again the natural numbers that we have seen before with constructors

$$\begin{array}{lcl} \overline{\underline{0}} & := & \lambda x\, f.x \\ \overline{\underline{S}} & := & \lambda n.\lambda x\, f.f\, n\, (n\, x\, f). \end{array}$$

The Church-Scott data types enjoy a primitive recursion scheme. An important instance of this is that one can define the inverse of the constructors, a *destructor*. The reason is that the case scheme and the iteration scheme are both instances of the primitive recursion scheme.

▶ **Definition 23.** Given the Church-Scott encoding of a data-type $D$, the *destructor* is defined as follows. We abbreviate $A := T^1(\overline{\underline{D}}) + \ldots + T^n(\overline{\underline{D}})$.

$$\begin{array}{lcll} \mathrm{dest} & : & \overline{\underline{D}} \to T^1(\overline{\underline{D}}) + \ldots + T^n(\overline{\underline{D}}) & \\ & := & \lambda d.d\, p_1 \ldots p_n & \\ & & \text{where } p_j & : \quad T^j(\overline{\underline{D}} \times A) \to A \\ & & & := \quad \mathrm{in}_j \circ T^j(\pi_1) \end{array}$$

▶ **Lemma 24.** *Given the Church-Scott encoding of a data-type $D$, the destructor satisfies the property*

$$\mathrm{dest}(\overline{\underline{\mathbf{C}_i}}\, x_1 \ldots x_{\mathsf{ar}(i)}) \twoheadrightarrow_\beta \mathrm{in}_i\langle x_1 \ldots x_{\mathsf{ar}(i)}\rangle$$

**Proof.**

$$\begin{array}{lll} \mathrm{dest}(\overline{\underline{\mathbf{C}_i}}\, x_1 \ldots x_{\mathsf{ar}(i)}) & \twoheadrightarrow_\beta & p_i\, t_1^* \ldots t_{\mathsf{ar}(i)}^* \\ & & \quad \text{where} \quad t_j^* = x_j \qquad \text{if } \alpha \notin \mathrm{FV}(T_j^i(\alpha)) \\ & & \qquad\qquad\; t_j^* = \langle x_j, x_j \vec{p}\rangle \quad \text{if } T_j^i(\alpha) = \alpha \\ & \twoheadrightarrow_\beta & \mathrm{in}_i(T^j(\pi_1)\, t_1^* \ldots t_{\mathsf{ar}(i)}^*) \\ & \twoheadrightarrow_\beta & \mathrm{in}_i\langle x_1 \ldots x_{\mathsf{ar}(i)}\rangle \end{array}$$

◀

▶ **Lemma 25.** *The Church-Scott encoding of a data-type $D$ enjoys the* primitive recursion scheme*, given by*

$$\frac{f_1 : T^1(\overline{\underline{D}} \times \tau) \to \tau \qquad f_n : T^n(\overline{\underline{D}} \times \tau) \to \tau \qquad d : \overline{\underline{D}}}{\mathrm{Rec}\, f_1 \ldots f_n\, d : \tau}$$

*and the following reduction rule (where we abbreviate $f_1 \ldots f_n$ to $\vec{f}$)*

$$\mathrm{Rec}\, f_1 \ldots f_n\, (\overline{\underline{\mathbf{C}_i}}\, x_1 \ldots x_{\mathsf{ar}(i)}) \twoheadrightarrow_\beta f_i\, (T_1^i\langle \mathrm{id}, \mathrm{Rec}\, \vec{f}\rangle x_1) \ldots (T_{\mathsf{ar}(i)}^i\langle \mathrm{id}, \mathrm{Rec}\, \vec{f}\rangle x_{\mathsf{ar}(i)})$$

*Here, $T_j^i\langle \mathrm{id}, \mathrm{Rec}\, \vec{f}\rangle x$ denotes*

$$\begin{cases} x & \text{if} \quad \alpha \notin FV(T_j^i(\alpha)) \\ \langle x, \mathrm{Rec}\, \vec{f}\, x\rangle & \text{if} \quad T_j^i(\alpha) = \alpha \end{cases}$$

**Proof.** If we define $\mathrm{Rec}\, f_1 \dots f_n\, d := d\, f_1 \dots f_n$, the typing is immediate. For the reduction rule

$$\begin{array}{rcl} \mathrm{Rec}\, f_1 \dots f_n\, (\overline{\mathbf{C}_i}\, x_1 \dots x_{\mathsf{ar}(i)}) & = & \overline{\mathbf{C}_i}\, x_1 \dots x_{\mathsf{ar}(i)}\, f_1 \dots f_n \\ & \twoheadrightarrow_\beta & f_i\, t_1^* \dots t_{\mathsf{ar}(i)}^* \\ & & \quad \text{where} \quad \begin{array}{ll} t_j^* = x_j & \text{if } \alpha \notin \mathrm{FV}(T_j^i(\alpha)) \\ t_j^* = \langle x_j, x_j \vec{f} \rangle & \text{if } T_j^i(\alpha) = \alpha \end{array} \\ & = & f_i\, (T_1^i \langle \mathrm{id}, \mathrm{Rec}\, \vec{f} \rangle x_1) \dots (T_{\mathsf{ar}(i)}^i \langle \mathrm{id}, \mathrm{Rec}\, \vec{f} \rangle x_{\mathsf{ar}(i)}) \end{array}$$

◀

That all this works is due to that fact that we can define *recursive algebras* (in the sense of [8], see Definition 13) inside $\lambda 2\mu$ in a generic way. We have shown here how to do that for functors of the form $\Phi(\alpha) = T^1(\alpha) + \ldots + T^n(\alpha)$ where each $T^i(\alpha)$ is a product of $\alpha$'s and objects that do not depend on $\alpha$. It would be interesting to extend this further to all polynomial functors.

## 4.2 Co-inductive data types

It is well-known that streams over a base type $A$ can also be defined in $\lambda 2$:

$$\mathrm{Str}_A := \exists \alpha.\alpha \times (\alpha \to A \times \alpha)$$

This could be called the *Church encoding of co-data-types* even though Church never used it. It is most likely due to [24]. We use the same type of implicit Curry-style typing for $\exists$ that we use for the $\forall$, and we use $\langle -, - \rangle$ for pairing and $\pi_1$ and $\pi_2$ for the projections. See Appendix A for the precise rules. Then the definitions of head and tail are as follows.

$$\begin{array}{rcl} \mathrm{hd}\, s & := & s\, (\lambda p.\pi_1 (\pi_2\, p\, (\pi_1 p))) \\ \mathrm{tl}\, s & := & s\, (\lambda p.\lambda f.f\, \langle \pi_2 (\pi_2\, p\, (\pi_1 p)), \pi_2\, p \rangle) \end{array}$$

This gives a weakly terminal co-algebra for the functor $T(\alpha) = A \times \alpha$. In general, given a type scheme $\Phi(\alpha)$, a weakly terminal co-algebra for $\Phi$ can be defined as $\exists \alpha.\alpha \times (\alpha \to \Phi(\alpha)$.

To get a general definition of co-algebraic data that we will define in $\lambda 2\mu$, we take the destructors apart.

▶ **Definition 26.** A *first order co-data-type* will be written as

$$\begin{array}{lll} \textbf{co-data-type} \quad D \quad \text{with destructors} & & \\ & \mathbf{D}_1^D & : \quad D \to T^1(D) \\ & & \dots \\ & \mathbf{D}_n^D & : \quad D \to T^n(D) \end{array}$$

where each of the $T^i(X)$ is either $X$ or a type expression that does not contain $X$. If $D$ is clear from the context, we will omit it as a superscript and write $\mathbf{D}_i$ in stead of $\mathbf{D}_i^D$.

This defines a co-algebraic data-type $D$ with $n$ destructors with names $\mathbf{D}_1, \dots, \mathbf{D}_n$. Each constructor $\mathbf{D}_i$ has arity 1. This is a limitation, as in general we would like a destructor to have a higher arity: $\mathbf{D} : D \to T_1(D) + \ldots + T_k(D)$. This can be done, but the syntax becomes very heavy, so we leave the precise treatment of the more general case for the future.

Using this definition of co-data-type, we obtain for streams over a base type $A$:

$$\begin{array}{lllll} \textbf{co-data-type} & \mathrm{Str}_A & \text{with destructors} & & \\ & & \mathrm{hd} & : & \mathrm{Str}_A \to A \\ & & \mathrm{tl} & : & \mathrm{Str}_A \to \mathrm{Str}_A \end{array}$$

This is encoded in system $F$ as

$$\overline{\mathrm{Str}_A} := \exists\alpha.\alpha \times (\alpha \to A) \times (\alpha \to \alpha)$$

The definitions of head and tail then are as follows, where we use $\langle -\rangle$ to denote a tuple of arbitrary length (so we avoid nested pairs as much as possible) and $\pi_j$ denotes projection on the $j$-th component of a tuple. We also define $\mathrm{CoIt}\, M_1\, M_2\, t$

$$\begin{array}{rcl} \overline{\mathrm{hd}}\, s & := & s\,(\lambda p.\pi_2\, p\,(\pi_1 p)) \\ \overline{\mathrm{tl}}\, s & := & s\,(\lambda p.\lambda f.f\,\langle \pi_3\, p\,(\pi_1 p), \pi_2\, p, \pi_3\, p\rangle) \\ \mathrm{CoIt}\, M_1\, M_2\, t & := & \lambda f.f\,\langle f, M_1, M_2\rangle \end{array}$$

It is a simple check that $\overline{\mathrm{Str}_A}$ thus defined enjoys the co-iteration scheme for streams of Definition 10. Now it is hard to define the cons operator, that takes an $a : A$ and an $s : \overline{\mathrm{Str}_A}$ to create $\mathrm{cons}\, a\, s : \overline{\mathrm{Str}_A}$. However, in $\lambda 2\mu$ we can define a *Scott encoding of streams* that does enable the definition of the constructor, because it enjoys a *co-case scheme*.

▶ **Definition 27.** A type $\mathrm{Str}_A$ enjoys the *co-case scheme for streams* if there are terms $\mathrm{hd} : \mathrm{Str}_A \to A$, $\mathrm{tl} : \mathrm{Str}_A \to \mathrm{Str}_A$ and $\mathrm{CoCase}\, M_1\, M_2\, t$ (for any $M_1$, $M_2$ and $t$) such that

$$\frac{M_1 : \tau \to A \quad M_2 : \tau \to \mathrm{Str}_A \quad t : \tau}{\mathrm{CoCase}\, M_1\, M_2\, t : \mathrm{Str}_A}$$

with reductions

$$\begin{array}{rcl} \mathrm{hd}(\mathrm{CoCase}\, M_1\, M_2\, t) & \twoheadrightarrow & M_1 t \\ \mathrm{tl}(\mathrm{CoCase}\, M_1\, M_2\, t) & \twoheadrightarrow & M_2\, t \end{array}$$

▶ **Definition 28.** The *Scott encoding of streams in* $\lambda 2\mu$ is given by

$$\begin{array}{rcl} \underline{\mathrm{Str}_A} & := & \mu\beta.\exists\alpha.\alpha \times (\alpha \to A) \times (\alpha \to \beta) \\ \underline{\mathrm{hd}}\, s & := & s\,(\lambda p.\pi_2\, p\,(\pi_1 p)) \\ \underline{\mathrm{tl}}\, s & := & s\,(\lambda p.\pi_3\, p\,(\pi_1 p)) \end{array}$$

So we have the type equation $\underline{\mathrm{Str}_A} = \exists\alpha.\alpha \times (\alpha \to A) \times (\alpha \to \underline{\mathrm{Str}_A})$, which makes the terms $\underline{\mathrm{hd}}$ and $\underline{\mathrm{tl}}$ well-typed.

▶ **Lemma 29.** *The* Scott encoding of streams in $\lambda 2\mu$*, enjoys a co-case scheme, where*

$$\mathrm{CoCase}\, M_1\, M_2\, t \quad := \quad \lambda f.f\,\langle t, M_1, M_2\rangle$$

The proof is an immediate check of the properties of Definition 27. We can now also define the constructor easily. For $a : A$ and $s : \underline{\mathrm{Str}_A}$,

$$\mathrm{cons}\, a\, s \quad := \quad \mathrm{CoCase}\,(\lambda z.a)\,(\lambda z.s)\, a$$

and one immediately verifies that $\underline{\mathrm{hd}}\,(\mathrm{cons}\, a\, s) \twoheadrightarrow_\beta a$, $\underline{\mathrm{tl}}\,(\mathrm{cons}\, a\, s) \twoheadrightarrow_\beta s$.

We can generalize this further and define *co-recursive co-algebras* (Definition 14) in $\lambda 2\mu$ for the co-data-type of Definition 26. For streams we get the following.

▶ **Definition 30.** A type $\mathrm{Str}_A$ enjoys the *co-recursion scheme for streams* if there are terms $\mathrm{hd} : \mathrm{Str}_A \to A$, $\mathrm{tl} : \mathrm{Str}_A \to \mathrm{Str}_A$ and $\mathrm{CoRec}\, M_1\, M_2\, t$ (for any $M_1$, $M_2$ and $t$) such that

$$\frac{M_1 : \tau \to A \quad M_2 : \tau \to \mathrm{Str}_A + \tau \quad t : \tau}{\mathrm{CoRec}\, M_1\, M_2\, t : \mathrm{Str}_A}$$

with reductions

$$\begin{array}{rcl} \mathrm{hd}(\mathrm{CoRec}\, M_2\, t) & \twoheadrightarrow & M_1 t \\ \mathrm{tl}(\mathrm{CoRec}\, M_1\, M_2\, t) & \twoheadrightarrow & \mathrm{case}\ \ M_2\, t\ \ \mathrm{of}\ (\mathrm{inl}\, x \Rightarrow x)\ (\mathrm{inr}\, x \Rightarrow \mathrm{CoRec}\, M_1\, M_2\, x) \end{array}$$

▶ **Definition 31.** The *Church-Scott encoding of streams in* $\lambda 2\mu$ is given by

$$\begin{array}{rcl} \overline{\underline{\mathrm{Str}_A}} & := & \mu\beta.\exists\alpha.\alpha \times (\alpha \to A) \times (\alpha \to \beta + \alpha) \\ \overline{\underline{\mathrm{hd}}}\, s & := & s\, (\lambda p.\pi_2\, p\, (\pi_1 p)) \\ \overline{\underline{\mathrm{tl}}}\, s & := & s\, (\lambda p.\mathrm{case}\ \ \pi_3\, p\, (\pi_1 p)\ \ \mathrm{of}\ (\mathrm{inl}\, x \Rightarrow x)\ (\mathrm{inr}\, x \Rightarrow \lambda f.f\, \langle x,\, \pi_2 p, \pi_3\, p\rangle)) \end{array}$$

▶ **Lemma 32.** *The* Church-Scott encoding of streams in $\lambda 2\mu$*, enjoys a co-recursion scheme, where*

$$\mathrm{CoRec}\, M_1\, M_2\, t \quad := \quad \lambda f.f\, \langle t, M_1, M_2\rangle$$

The proof is immediate by unfolding the definitions.

The approach taken above for streams works for more general co-inductive data-types. It gives nice finite representations of infinitary data in the untyped $\lambda$-calculus. (E.g. the stream of natural numbers is a term in normal form.) We now show how to define Church, Scott and Church-Scott co-data-types in general. Let a co-data-type $D$ as in Definition 26 be given.

▶ **Definition 33.** The Church encoding of $D$ in system $F$ is as follows.

$$\overline{D} := \exists\alpha.\alpha \times (\alpha \to T^1(\alpha)) \times \ldots \times (\alpha \to T^n(\alpha))$$

with the destructors $\mathbf{D}_i : D \to T^i(D)$ defined by

$$\begin{array}{rcll} \overline{\mathbf{D}_i}\, s & := & s\, (\lambda p.\pi_{i+1}\, p\, (\pi_1 p)) & \text{if} \quad \alpha \notin \mathrm{FV}(T^i(\alpha)) \\ \overline{\mathbf{D}_i}\, s & := & s\, (\lambda p.\lambda f.f\, \langle \pi_{i+1}\, p\, (\pi_1 p), \pi_2\, p, \ldots, \pi_{n+1}\, p\rangle) & \text{if} \quad T^i(\alpha) = \alpha \end{array}$$

A Church encoded co-data-type enjoys a co-iteration scheme, that we define now.

▶ **Definition 34.** A type $S$ enjoys the *co-iteration scheme* for co-date-type $D$ if there are destructor terms and terms $\mathrm{CoIt}\, M_1 \ldots M_n\, t$ (for any $M_1, \ldots M_n$ and $t$) such that

$$\frac{M_1 : \tau \to T^1(\tau) \quad M_n : \tau \to T^n(\tau) \quad t : \tau}{\mathrm{CoIt}\, M_1 \ldots M_n\, t : S}$$

with reductions

$$\begin{array}{rcll} \mathbf{D}_i\, (\mathrm{CoIt}\, M_1 \ldots M_2\, t) & \twoheadrightarrow & M_i t & \text{if} \quad \alpha \notin \mathrm{FV}(T^i(\alpha)) \\ \mathbf{D}_i\, (\mathrm{CoIt}\, M_1 \ldots M_2\, t) & \twoheadrightarrow & \mathrm{CoIt}\, M_1 \ldots M_2\, (M_i\, t) & \text{if} \quad T^i(\alpha) = \alpha \end{array}$$

▶ **Lemma 35.** *The Church encoded co-data-type enjoys a co-iteration scheme, if we take*

$$\mathrm{CoIt}\, M_1 \ldots M_n\, t := \lambda f.f\, \langle t, M_1 \ldots M_n\rangle$$

**Proof.** The reduction properties are checked immediately. ◀

In $\lambda 2\mu$ we can define a *Scott encoding of co-data-types* that enables the definition of the constructor right away, because it enjoys a *co-case scheme*.

▶ **Definition 36.** A type $S$ enjoys the *co-case scheme* for co-date-type $D$ if there are destructor terms and $\mathrm{CoCase}\, M_1 \dots M_n\, t$ (for any $M_1, \dots, M_n$ and $t$) such that

$$\frac{M_1 : \tau \to T^1(S) \quad M_n : \tau \to T^n(S) \quad t : \tau}{\mathrm{CoCase}\, M_1 \dots M_n\, t : S}$$

with reductions

$$\mathbf{D}_i(\mathrm{CoCase}\, M_1 \dots M_n\, t) \quad \twoheadrightarrow \quad M_i\, t$$

▶ **Definition 37.** The Scott encoding of co-data-type $D$ in $\lambda 2\mu$ is as follows.

$$\underline{D} := \mu\beta.\exists\alpha.\alpha \times (\alpha \to T^1(\beta)) \times \dots \times (\alpha \to T^n(\beta))$$

with the destructors $\underline{\mathbf{D}_i} : \underline{D} \to T^i(\underline{D})$ encoded by

$$\underline{\mathbf{D}_i}\, s \quad := \quad s\, (\lambda p.\pi_{i+1}\, p\, (\pi_1 p))$$

▶ **Lemma 38.** *The Scott encoding of co-data-type $D$ enjoys a co-case scheme, where*

$$\mathrm{CoCase}\, M_1 \dots M_n\, t \quad := \quad \lambda f.f\, \langle t, M_1, \dots, M_n \rangle$$

The proof is an immediate check of the properties of Definition 36. Now we can also define the constructor, $\mathrm{cons} : T^1(\underline{D}) \to \dots \to T^n(\underline{D}) \to \underline{D}$, using CoCase or directly. We define it directly:

$$\mathrm{cons} := \lambda x_1, \dots, x_n.\lambda f.f\, \langle x_1, \lambda z.x_1, \dots, \lambda z.x_n \rangle$$

Then

$$\underline{\mathbf{D}_i}\, (\mathrm{cons}\, x_1, \dots, x_n) \twoheadrightarrow_\beta x_i$$

Note that the choice of $x_1$ as first element in the tuple in the definition of cons is completely arbitrary.

We now define the Church-Scott encoding of co-data-types.

▶ **Definition 39.** The Church-Scott encoding of co-data-type $D$ in $\lambda 2\mu$ is as follows.

$$\overline{D} := \mu\beta.\exists\alpha.\alpha \times (\alpha \to T^1(\beta + \alpha)) \times \dots \times (\alpha \to T^n(\beta + \alpha))$$

with the destructors $\overline{\mathbf{D}_i} : \overline{D} \to T^i(\overline{D})$ defined by

$$\begin{array}{lcll} \overline{\mathbf{D}_i}\, s & := & s\, (\lambda p.\pi_{i+1}\, p\, (\pi_1 p)) & \text{if} \quad \alpha \notin \mathrm{FV}(T^i(\alpha)) \\ \overline{\overline{\mathbf{D}_i}}\, s & := & s\, (\lambda p.\mathrm{case}\ \ \pi_{i+1}\, p\, (\pi_1 p)\ \ \mathrm{of} & \\ & & \quad (\mathrm{inl}\, x \Rightarrow x)\ (\mathrm{inr}\, x \Rightarrow \lambda f.f\, \langle x, \pi_2 p, \dots, \pi_{n+1}\, p \rangle) & \text{if} \quad T^i(\alpha) = \alpha \end{array}$$

▶ **Definition 40.** The type $S$ enjoys a *co-recursion scheme* for co-data-type $D$ in case there are destructor terms and $\mathrm{CoRec}\, M_1 \dots M_n\, t$ (for any $M_1, \dots, M_n$ and $t$) such that

$$\frac{M_1 : \tau \to T^1(S + \tau) \quad M_n : \tau \to T^n(S + \tau) \quad t : \tau}{\mathrm{CoRec}\, M_1 \dots M_n\, t : S}$$

with reductions

$$\begin{array}{lcll} \mathbf{D}_i(\mathrm{CoRec}\, M_1 \dots M_n\, t) & \twoheadrightarrow & M_i\, t & \\ & & \text{if } \alpha \notin \mathrm{FV}(T^i(\alpha)) & \\ \mathbf{D}_i(\mathrm{CoRec}\, M_1 \dots M_n\, t) & \twoheadrightarrow & \mathrm{case}\ \ M_i\, t\ \ \mathrm{of}\ (\mathrm{inl}\, x \Rightarrow x)\ (\mathrm{inr}\, x \Rightarrow \mathrm{CoRec}\, M_1 \dots M_n\, x) & \\ & & \text{if } T^i(\alpha) = \alpha & \end{array}$$

▶ **Lemma 41.** *The Church-Scott encoding of co-data-types in $\lambda 2\mu$, enjoys a co-recursion scheme, where*

$$\mathrm{CoRec}\, M_1 \dots M_n\, t \quad := \quad \lambda f.f\, \langle t, M_1, \dots, M_n \rangle$$

**Proof.** The requirements in Definition 40 are verified right away. ◀

## 5 Conclusion and Future Work

We have shown how to represent data-types and co-data-types in $\lambda 2\mu$, the extension of system $F$ with recursive types, in such a way that we support the function definitions schemes of (co-)iteration, (co-)case and primitive (co-)recursion. This also allows for new generic definitions of data in pure untyped $\lambda$-calculus, so without using additional constants or constructs, like e.g. pattern matching, guarded recursion. The data-types we have considered are still limited, so we want to extend this further. The further reaching goal is to be able to translate all computational language constructs of a powerful language like CIC (Calculus of Inductive Constrictions) to the untyped $\lambda$-calculus, to create a simple computational basis for the proof assistant Coq.

It should be pointed out – as was already remarked by Parigot [18] for the numerals – that the Church-Scott data-types have an inefficient encoding: the size of $\overline{n}$ is exponential in $n$, which is clearly undesirable. So, this is something to resolve.

The Church-Scott data-types have another deficit compared to the Church and Scott data-types: there are closed terms of type nat that are not the encoding of a natural number. For example, $\lambda x\, f.f\, \underline{0}\, (\underline{2}\, x\, f)$ is such a term. However, if we use more refined types, like TTR of [18], such a term is not well-typed, and only the terms $\overline{n}$ are closed terms of type nat. The system TTR, is second order predicate logic extended with recursive definitions of predicates (like $\lambda 2\mu$ is an extension of system $F$) and the idea is that formulas are specifications and programs can be extracted from the proofs of specifications. This is an approach that has been advocated and studied by Krivine, Parigot and Leivant ([13, 18]) and the latter has recently made a first extension to co-inductive data-types [15].

Another point for further research is to see how these data-types behave for the scheme of *course of value recursion* as defined in [22]. Also, the *Mendler style* inductive (and co-inductive) data types are worth comparing with the approach that we have chosen [16, 21].

## A Typing rules for $\lambda 2\mu$

We now want to make precise what the type theory is that we are working with. We assume that simple type theory and polymorphic type theory (syetm $F$) are familiar. We look at typing *à la Curry*, so we give types to untyped $\lambda$-terms. We define $\lambda 2\mu$ as an extension of system $F$ à la Curry.

▶ **Definition 42.** The types of $\lambda 2\mu$ are the ones of system $F$ extended with types of the form

$$\mu\beta.\Phi(\beta)$$

where $\Phi(\beta)$ is a system $F$ type in which the type variable $\beta$ occurs positively. $\mu\beta$ binds the type variable $\beta$ in $\Phi(\beta)$, just like the $\forall$ does.

The equational theory on types is the contextual closure of the equations $\mu\beta.\Phi(\beta) = \Phi(\mu\beta.\Phi(\beta))$.

So, we don't have nested occurrences of $\mu$. To make notation simpler, we introduce a name for every type $\mu\beta.\Phi(\beta)$ that we want to talk about. This is what we do in the paper. Then we have:

$$\frac{A := \mu\beta.\Phi(\beta)}{A = \Phi(A)}$$

The typing rules are the usual ones of system $F$ à la Curry, extended with

$$\frac{\Gamma \vdash M : A \quad A = B}{\Gamma \vdash M : B}$$

We assume that we have product and sum types. These are definable, so we don't need to add them. To fix notation we give the rules here.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M : B}{\Gamma \vdash \langle M, N\rangle : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1\, M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2\, M : B}$$

$$\frac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N : C \quad \Gamma, x : B \vdash P : C}{\Gamma \vdash \text{case } M \text{ of } (\text{inl}\, x \Rightarrow N)\,(\text{inr}\, x \Rightarrow P) : C} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl}\, M : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inr}\, M : A + B}$$

With reduction rules

$$\begin{array}{rcl} \pi_i\, \langle M_1, M_2\rangle & \twoheadrightarrow_\beta & M_i \\ \text{case } (\text{inl}\, Q) \text{ of } (\text{inl}\, x \Rightarrow N)\,(\text{inr}\, x \Rightarrow P) & \twoheadrightarrow_\beta & N[Q/x] \\ \text{case } (\text{inr}\, Q) \text{ of } (\text{inl}\, x \Rightarrow N)\,(\text{inr}\, x \Rightarrow P) & \twoheadrightarrow_\beta & P[Q/x] \end{array}$$

We will avoid iterated pairs and projections as much as possible and similarly for iterated injections. So we will just have

$$\frac{\Gamma \vdash M : A_i}{\Gamma \vdash \text{in}_i M : A_1 + \ldots + A_n} \qquad \frac{\Gamma \vdash M : A_1 \times \ldots \times A_n}{\Gamma \vdash \pi_i M : A_i}$$

We pay some special attention to the rules for $\forall$ and $\exists$, because we are in Curry style, so types are implicit. First the rules for $\forall$

$$\frac{\Gamma \vdash M : \forall\alpha.A}{\Gamma \vdash M : A[B/\alpha]} \text{ for } B \text{ a type} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall\alpha.A} \text{ if } \alpha \notin \text{FV}(\Gamma)$$

The $\exists$-quantifier is defined in terms of the $\forall$:

$$\exists\beta.A := \forall\alpha.(\forall\beta.A \to \alpha) \to \alpha$$

This imples that we have the following derived rules for $\exists$. These are the ones we actually use.

$$\frac{\Gamma \vdash M : \exists\alpha.A \quad \Gamma, p : A \vdash N : B}{\Gamma \vdash M\,(\lambda p.N) : B} \text{ if } \alpha \notin \text{FV}(\Gamma, B) \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \lambda f.f\, M : \exists\alpha.A}$$

## References


1 M. Abadi, L. Cardelli, and G. Plotkin. Types for the Scott numerals, 1993. `http://lucacardelli.name/Papers/Notes/scott2.pdf`.

2 H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, revised edition, 1984.

3 Erik Barendsen. An unsolvable numeral system in lambda calculus. *J. Funct. Program.*, 1(3):367–372, 1991.

4 Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

5 Aloïs Brunel and Kazushige Terui. Church => Scott = Ptime: an application of resource sensitive realizability. In Patrick Baillot, editor, *Proceedings International Workshop on Developments in Implicit Computational complExity, DICE 2010, Paphos, Cyprus, 27-28th March 2010.*, volume 23 of *EPTCS*, pages 31–46, 2010.

6 Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

7 H. Curry, J. Hindley, and J. Seldin. *Combinatory Logic*, volume 2. North Holland Publishing Company, 1972.

8 H. Geuvers. Inductive and coinductive types with iteration and recursion. In *Proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden*, pages 183–207, 1992. `http://www.cs.ru.nl/~herman/PUBS/BRABasInf_RecTyp.ps.gz`.

9 H. Geuvers and E. Poll. Iteration and primitive recursion in categorical terms. In E. Barendsen, H. Geuvers, V. Capretta, and M. Niqui, editors, *Reflections on Type Theory, Lambda Calculus, and the Mind, Essays Dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, pages 101–114. Radboud Universiteit Nijmegen, 2007.

10 J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.

11 Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science, Edinburgh, UK, September 7-9, 1987, Proceedings*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer, 1987.

12 Jan Martin Jansen. Programming in the $\lambda$-calculus: From Church to Scott and back. In Peter Achten and Pieter W. M. Koopman, editors, *The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, volume 8106 of *Lecture Notes in Computer Science*, pages 168–180. Springer, 2013.

13 Jean-Louis Krivine and Michel Parigot. Programming with proofs. *Elektronische Informationsverarbeitung und Kybernetik*, 26(3):149–167, 1990.

14 Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science*, 8(3), 2012.

15 Daniel Leivant. Global semantic typing for inductive and coinductive computing. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 469–483. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

16 N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic*, 51(1-2):159–172, 1991.

17 Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

18 Michel Parigot. Recursive programming with proofs. *Theor. Comput. Sci.*, 94(2):335–336, 1992.

**19** D. Scott. A system of functional abstraction, 1963. Lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University, September 1963, furnished by author in 1968 (note in [7]).

**20** Aaron Stump. Directly reflective meta-programming. *Higher-Order and Symbolic Computation*, 22(2):115–144, 2009.

**21** Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nord. J. Comput.*, 6(3):343, 1999.

**22** Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica, Lith. Acad. Sci.*, 10(1):5–26, 1999.

**23** Ch. Wadsworth. Some unusual $\lambda$-calculus numeral systems. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

**24** G. C. Wraith. A note on categorical datatypes. In David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, editors, *Category Theory and Computer Science, Manchester, UK, September 5-8, 1989, Proceedings*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer, 1989.