A TYPED λ-CALCULUS WITH CALL-BY-NAME AND CALL-BY-VALUE ITERATION

HERMAN GEUVERS

Abstract. We define a typed λ-calculus where data-types are represented as *Scott data types* (as opposed to the more well-known *Church data-types*), and where computation is done via continuations. Scott data types don't have any recursion built in, so we add an iteration scheme to define functions by well-founded structural recursion. As a matter of fact, we add two schemes: one for *call-by-value iteration* and one for *call-by-name iteration*. The advantage is that we can control the computation behavior of functions via their definition, and not via the evaluation relation we choose: evaluation is done via weak head reduction.

In the paper we show that our calculus is strongly normalizing and we define a notion of model for the calculus. Our calculus has polymorphism and recursive types, so the strong normalization proof is somewhat like the one of Mendler [16]. However, we have some additional features, like iterators over Scott data types, so we can not just copy or reuse the proof of Mendler. We first define a saturated-sets model, in the style of Krivine [13], and then show that the SN proof is almost an instance of it.

We also show how we can switch from call-by-value to call-by-name , via a kind of double negation translation and we show how to define storage operators (in the sense of Krivine [12, 14]).

Keywords: typed lambda calculus, Scott data-types, call-by-name, call-by-value, continuations.

1. Introduction

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 the polymorphic λ calculus one can define closed data types for this data, the iteration scheme is well-typed and β-reduction is always terminating. The much less well-known *Scott encoding* has *case distinction* as a primitive. These data types were introduced by Scott in 1963 by Scott [21], but were never published and only became known later through Curry, Hindley and Seldin [4] and Wadsworth [24]. They have been reinvented independently by Mogensen [17] and it has regained interest among functional programmers [11, 22] and among researchers that use λ-calculus for the complexity analysis of functions [3, 15]. In [8] we give an overview of different approaches to data types, also presenting combined *Church-Scott data types*. For the natural numbers, these are also known as *Parigot numerals* [6, 19]. For Scott data types, the terms are not typable in polymorphic λ calculus and there is no iteration scheme, but there is a constant time *destructor* (e.g. predecessor). In [1] it is shown how to type Scott data types using recursive types.

As said, Scott data types encode a definition *by cases*. We will shed a different light on the Scott data types, by viewing the different cases as *continuations*. For natural numbers this means that there is a continuation for the 0-case and a continuation for the $n+1$-case (which takes n as input). As Scott data types do not have iteration (or recursion) built in, we add as primitives a *call-by-name iterator* and a *call-by-value iterator*. The reduction rules for the iterators expose the expected evaluation behavior if one does weak head reduction in the calculus. So, to get a call-by-name or call-by-value behavior we do not choose an evaluation order, as it is done, e.g. in the original work of Plotkin [20], but we choose the proper iteration scheme.

To represent Scott data types we need (at least) simple types plus positive recursive types. In the present paper we use polymorphic types plus positive recursive types, as the addition of polymorphism does not add any fundamental difficulty. A fundamental extension is that we add two iterators (call-by-value and call-by-name) as additional constants with special reduction rules. This is needed, because the Scott data types do not provide a lot of expressivity themselves in terms of definability of recursive functions. We show that the calculus is strongly normalizing (SN) by adapting the well-known saturated sets proof in a non-trivial way.

The interest of the calculus lies in the fact that one can *choose* for a call-by-name or call-by-value function by choosing an iteration scheme. The work is partly motivated by our studies into Continuation Calculus, [7, 9], which aims at being a simplified model of functional computation. The proof of SN of $\lambda 2\mu$It also implies SN of the typed Continuation Calculus of [7, 9].

1.1. **Scott and Church Data-types.** We now give some details of the data types in the system.

Definition 1.1. *The Scott numerals are defined as follows.*

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

The Scott numerals have *case* as a basis: the numerals are *case distinctors*: $\underline{n}\, q\, t = q$ if $n = 0$ and $\underline{n}\, q\, t = t\underline{m}$ if $n = m + 1$. An advantage is that the predecessor can immediately be defined: $\mathrm{pred} := \lambda n.n\, \underline{0}\, (\lambda x.x)$. On the other hand, one has to get "recursion" from somewhere else (e.g. in the untyped λ-calculus by using a fixed point-combinator).

In contrast, we also have the Church numerals.

Definition 1.2. *The Church numerals are defined as follows.*

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

The Church numerals have *iteration* as basic concept: the numerals are iterators, in the sense that $\overline{n}\, x\, f$ iterates the function f on x, for n-times.

To type Scott numerals we need $\mathsf{N} = A \to (\mathsf{N} \to A) \to A$. In $\lambda 2$, we cannot do this, unless we extend it with (positive) recursive types, obtaining $\lambda 2\mu$:

$$\mathsf{N} := \mu Y.\forall X.X \to (Y \to X) \to X.$$

Using this approach, which is explained in [1], one still does not get any well-founded recursion "for free", so we still have to add iterators.

Church numerals can be typed in $\lambda 2$ in the well-known way:

$$\mathsf{N} := \forall X.X \to (X \to X) \to X$$

and it can be shown that many functions on the numbers can be defined as closed terms of type $\mathsf{N} \to \mathsf{N}$ in $\lambda 2$ (e.g. see [2, 10]).

In the present article we consider general first order data types in the Scott encoding. We now fix the data types we talk about and show how these data are represented as untyped λ-terms.

Definition 1.3. *A* first order data type *will be written as*

$$\begin{array}{lll} \textbf{data-type} \quad D \quad \textit{with constructors} & & \\ & \mathbf{c}_1^D \;:\; & D_1^1 \to \ldots \to D_{\mathsf{ar}1}^1 \to D \\ & & \ldots \\ & \mathbf{c}_n^D \;:\; & D_1^n \to \ldots \to D_{\mathsf{ar}n}^n \to D \end{array}$$

where each of the D^i_j is either D or a type expression that does not contain D. If D is clear from the context, we will omit it as a superscript and write $\mathbf{c}_i$ instead of $\mathbf{c}^D_i$.

Example 1.4. *Examples of data types are*

$$\begin{array}{rrcl} \textbf{data-type} & \textbf{Bool} & \textit{with constructors} & \\ & & \textbf{true} & : \ \textbf{Bool} \\ & & \textbf{false} & : \ \textbf{Bool} \end{array}$$

$$\begin{array}{rrcl} \textbf{data-type} & \textbf{Nat} & \textit{with constructors} & \\ & & \textbf{zero} & : \ \textbf{Nat} \\ & & \textbf{succ} & : \ \textbf{Nat} \to \textbf{Nat} \end{array}$$

And, for A a data type,

$$\begin{array}{rrcl} \textbf{data-type} & \textbf{List}_A & \textit{with constructors} & \\ & & \textbf{nil} & : \ \textbf{List}_A \\ & & \textbf{cons} & : \ A \to \textbf{List}_A \to \textbf{List}_A \end{array}$$

$$\begin{array}{rrcl} \textbf{data-type} & \textbf{BinTree}_A & \textit{with constructors} & \\ & & \textbf{leaf} & : \ A \to \textbf{BinTree}_A \\ & & \textbf{join} & : \ \textbf{BinTree}_A \to \textbf{BinTree}_A \to \textbf{BinTree}_A \end{array}$$

Given a first-order data type, we can directly interpret the data elements in the untyped λ-calculus, either in Church or in Scott style.

Definition 1.5. *Let a data type D be given with constructors $\mathbf{c}_1, \ldots, \mathbf{c}_k$. Let the arity of constructor $\mathbf{c}_i$ be $\mathsf{ar}(i)$ and let the type of $\mathbf{c}_i$ be $D^i_1 \to \ldots \to D^i_{\mathsf{ar}(i)} \to D$. We say that the j-th* argument of $\mathbf{c}_i$ is recursive *in case D^i_j is D.*

The Scott encoding *of a constructor is:*

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

The Church encoding *of a constructor is:*

$$\overline{\mathbf{c}_i} := \lambda x_1 \ldots x_{\mathsf{ar}(i)}.\lambda c_1 \ldots c_k.c_i\, d_1 \ldots d_{\mathsf{ar}(i)},$$

where $d_j = x_j$ if the j-th argument of $\mathbf{c}_i$ is not recursive and $d_j = x_j c_1 \ldots c_k$ if the j-th argument of $\mathbf{c}_i$ is recursive.

We see that the Scott encoding is much simpler and it gives a direct definition of the the 'destructor': given $M = \underline{\mathbf{c}_i} t_1 \ldots t_{\mathsf{ar}(i)}$, we get t_j by applying M to k arguments with the i-th argument being $\lambda y_1 \ldots y_{\mathsf{ar}(i)}.y_j$. On the other hand it is less convenient, because we don't get any recursion for free. In [8] we have made a more thorough comparison between these various notions of data type, also studying the combined *Church-Scott data types*.

Continuation Calculus. In previous work on the *Continuation Calculus* (CC) [7,9], we also use Scott data types, but our approach is slightly different from what we present here: it is based on the idea of taking $\bot$ to be the type of *executable terms*. Then we define

$$\mathsf{N} := \mu Y.\bot \to (Y \to \bot) \to \bot.$$

We view a number as taking two continuations, $c : \bot$ and $s : \mathsf{N} \to \bot$ and returning c – in case the number is 0 – or $s\, n$ – in case the number is $n + 1$. In the case of CC we have simple types, a base type $\bot$ and positive recursive types. As usual from logic we abbreviate $A \to \bot$ to $\neg A$. Furthermore, in the Continuation Calculus we have *iterators*.

To define iterators in CC, we first notice that every data type B in CC has the following shape: $B = B^1 \to \ldots \to B^m \to \bot$ with each B^k of the shape $\ldots \to \bot$. We abbreviate $B^1 \to \ldots \to B^m \to \bot$ to $\overline{B} \to \bot$. Then the iterators for N are:

$$\text{(It-CBN)}\quad \frac{f_1 : B \quad f_2 : B \to B \quad c_1 : B^1 \dots c_m : B^m \quad n : \mathsf{N}}{\mathsf{Itcbn}\, f_1\, f_2\, n\, c_1 \dots c_m : \bot}$$

$$\text{(It-CBV)}\quad \frac{f_1 : \neg\neg B \quad f_2 : B \to \neg\neg B \quad c : \neg B \quad n : \mathsf{N}}{\mathsf{Itcbv}\, f_1\, f_2\, c\, n : \bot}$$

The call-by-name iterator very much looks like the well-known iterator for data types: given $f_1 : B$ and $f_2 : B \to B$, we basically have $\mathsf{Itcbn}\, f_1\, f_2 : \mathsf{N} \to B$. Only now, this function is fully applied to obtain a term of type $\bot$ (because only terms of type $\bot$ can evaluate). If one abstracts over $n : \mathsf{N}, c_1 : B^1, \dots c_m : B^m$ in the term above one gets the η-expansion of $\mathsf{Itcbn}\, f_1\, f_2$, that is $\lambda n : \mathsf{N}.\lambda c_1 : B^1 \dots . \lambda c_m : B^m.\mathsf{Itcbn}\, f_1\, f_2\, n\, c_1 \dots c_m : \mathsf{N} \to B$. However, in CC there is no λ-abstraction.

The call-by-value iterator is basically the iterator in double-negated form: If $f_1 : \neg\neg B$ and $f_2 : B \to \neg\neg B$, then $\mathsf{Itcbv}\, f_1\, f_2 : \neg B \to \neg\mathsf{N}$. An η-expansion of this term is $\lambda c : \neg B.\lambda n : \mathsf{N}.\mathsf{Itcbv}\, f_1\, f_2\, c\, n : \neg B \to \neg\mathsf{N}$.

The reduction rules for the call-by-name and the call-by-value iterators for N are as follows.

$$\begin{array}{lll} \mathsf{Itcbn}\, f_1\, f_2\, n\, \overline{c} & \to_n & n\, (f_1\, \overline{c})\, (\lambda x : \mathsf{N}.f_2\, (\lambda \overline{z} : \overline{B}.\mathsf{Itcbn}\, f_1\, f_2\, x\, \overline{z}) \overline{c}) \\ \mathsf{Itcbv}\, f_1\, f_2\, c\, n & \to_v & n\, (f_1\, c)\, (\lambda x : \mathsf{N}.\mathsf{Itcbv}\, f_1\, f_2\, (\lambda y : B.f_2\, y\, c)\, x) \end{array}$$

The idea is that call-by-value breaks down the input number n completely and stores in the continuation the number of times that f_2 has to be applied (as a series of compositions of f_2). On the contrary, call-by-name computes part of the result until one can do a case analysis on the output (what constructor the outputs begins with) and the passes it on to the appropriate continuation c_1, c_2, ... or c_m.

2. A typed λ-calculus with continuations and call-by-value and call-by-name iterators

We now define our system $\lambda 2\mu\mathrm{It}$.

Definition 2.1. *The* types *of $\lambda 2\mu It$ are*

$$\mathsf{T} := \bot \mid \mathsf{TVar} \mid (\mathsf{T} \to \mathsf{T}) \mid \mu\mathsf{TVar}.\mathsf{T} \mid \forall\mathsf{TVar}.\mathsf{T},$$

where, in $\mu X.\Phi$, we require X to occur positively *(see Definition 2.2) in Φ. The set TVar is a countable collection of type variables; we typically use X, Y, Z for type variables.*

As usual, we leave out the parentheses around functions types, so $A \to B \to C$ always means $A \to (B \to C)$.

Definition 2.2. *Given a type variable X and a type Φ, the notions X* occurs positively in Φ *and X* occurs negatively in Φ *are mutually defined as follows.*

1. *If $X \notin FV(\Phi)$ or $\Phi \equiv X$, then X occurs positively in Φ.*
2. *If X occurs positively in Φ and $X \neq Y$, then X occurs positively in $\mu Y.\Phi$ and $\forall Y.\Phi$.*
3. *If $\Phi \equiv \Phi_1 \to \Phi_2$ and X occurs negatively in Φ_1, and positively in Φ_2, then X occurs positively in Φ.*
4. *If $X \notin FV(\Phi)$, then X occurs negatively in Φ.*
5. *If X occurs negatively in Φ and $X \neq Y$, then X occurs negatively in $\mu Y.\Phi$ and $\forall Y.\Phi$.*
6. *If $\Phi \equiv \Phi_1 \to \Phi_2$ and X occurs negatively in Φ_2, positively in Φ_1, then X occurs negatively in Φ.*

The intention of the recursive type $\mu X.\Phi(X)$ is that it denotes a type A for which $A = \Phi(A)$. There is no rule stating that it is the *least* fixed point, even though in the model we interpret $\mu X.\Phi(X)$ via a least fixed point construction. To give the μ-types their semantics, we introduce type equalities.

Definition 2.3. *We define* equality between types, $A = B$, *as the least equivalence relation that can be derived using the following rules.*

$$(\textit{unfold})\quad \frac{}{\mu X.A = A\,[{}^{\mu X.A}/_{X}]} \qquad (\to\text{-}=)\quad \frac{A_1 = B_1 \qquad A_2 = B_2}{A_1 \to A_2 = B_1 \to B_2}$$

$$(\forall\text{-}=)\quad \frac{A = B}{\forall X.A = \forall X.B} \qquad (\mu\text{-}=)\quad \frac{A = B}{\mu X.A = \mu X.B}$$

We will in particular be concerned with μ-types that represent data types, because we have special terms for those: the iterators. The data types will be printed in **bold**, as we have seen in Example 1.4. The representation of these data type in $\lambda 2\mu$It will be printed in sand serif font. We first give the example of the data types of Example 1.4 in $\lambda 2\mu$It.

Example 2.4. *The well-known data types of booleans, numbers, lists and binary trees are defined as follows.*

$$\begin{array}{lcl} \mathsf{B} & := & \forall X.X \to X \to X \\ \mathsf{N} & := & \mu T.\forall X.X \to (T \to X) \to X \\ \mathsf{L}_A & := & \mu T.\forall X.X \to (A \to T \to X) \to X \\ \mathsf{BTree}_A & := & \mu T.\forall X.(A \to X) \to (T \to T \to X) \to X \end{array}$$

Example 2.4 gives the interpretation of some standard data-types in $\lambda 2\mu$It. We now show how first order data types in general (Definition 1.3) are represented in $\lambda 2\mu$It, and we define the constructors as closed $\lambda 2\mu$It-terms.

Definition 2.5. *Let D be a first order data type D as in Definition 1.3, with n constructors, where, for $1 \le i \le n$,*

$$\mathbf{c}_i : D_1^i \to \ldots \to D_{\mathsf{ar}(i)}^i \to D.$$

We abbreviate $D_1^i \to \ldots \to D_{\mathsf{ar}(i)}^i$ to $\overline{D}^i$, so we have $\mathbf{c}_i : \overline{D}^i \to D$.

We define D as the following type in $\lambda 2\mu$It.

$$\mathsf{D} := \mu T.\forall X.(\overline{D}^1[T/D] \to X) \to \ldots \to (\overline{D}^n[T/D] \to X) \to X.$$

For $i \in [1 \ldots n]$, we define the following constructor-terms

$$\begin{array}{lcl} \mathsf{c}_i^D & : & \overline{D}^i[\mathsf{D}/D] \to \mathsf{D}, \\ \mathsf{c}_i^D & := & \lambda x_1^i \ldots x_{\mathsf{ar}(i)}^i\, c_1 \ldots c_n.c_i\, x_1^i \ldots x_{\mathsf{ar}(i)}^i \end{array}$$

The definition of a data type $\mathsf{D} := \mu T.\forall X.(\overline{D}^1[T/D] \to X) \to \ldots \to (\overline{D}^n[T/D] \to X) \to X$ gives rise to the following type equality that will be used throughout.

$$\mathsf{D} = \forall X.(\overline{D}^1[\mathsf{D}/D] \to X) \to \ldots \to (\overline{D}^n[\mathsf{D}/D] \to X) \to X.$$

Example 2.6. *It can immediately be verified that the data types of Example 1.4 give rise to the types of Example 2.4 in $\lambda 2\mu$It.*

The typed λ calculus $\lambda 2\mu$It is defined by taking as types the ones of Definition 2.1 and as *data types* the ones of Definition 2.5. We also assume the derivation rules for type equations of Definition 2.3.

Definition 2.7. *The* pseudo-terms, Term *of* $\lambda 2\mu It$ *are defined by*

$$\mathsf{Term} ::= \mathsf{Var} \mid (\mathsf{Term}\,\mathsf{Term}) \mid (\lambda \mathsf{Var}.\mathsf{Term}) \mid \mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,d\,\overline{c} \mid \mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,c\,d,$$

where D *and* B *are data types.*

In $\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,d\,\overline{c}$*, we require that the length of* $\overline{f}$ *is the number of constructors of* D *and the length of* $\overline{c}$ *is the number of constructors of* B*; In* $\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,c\,d$*, we require that the length of* $\overline{f}$ *is the number of constructors of* D*.*

The terms do not contain type information: we write $\lambda x.M$ and not $\lambda x : A.M$, so we are in an *à la Curry* setting and not *à la Church*. Nevertheless, we will sometimes write a type with the variable in the λ-abstraction to increase understanding, as it makes it easier to read off the type from a term.

A *context* is a finite sequence of variable declarations to types, $x_1 : A_1, \ldots, x_n : A_n$. We write Γ for a context. The derivable judgments of $\lambda 2\mu$It are of the shape $\Gamma \vdash M : A$, where A is a type and M a pseudo-term.

Definition 2.8. *The derivation rules for judgments of* $\lambda 2\mu It$ *are as follows.*

$$(var)\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad (app)\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

$$(abs)\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad (=\text{-}conv)\ \frac{\Gamma \vdash x : A}{\Gamma \vdash x : B}\ \textit{if } A = B$$

$$(type\text{-}app)\ \frac{\Gamma \vdash M : \forall X.A}{\Gamma \vdash M : A[B/X]} \qquad (type\text{-}abs)\ \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall X.A}\ \textit{if } X \notin FV(\Gamma)$$

In defining the derivation rules for Itcbn *and* Itcbv *we omit '*$\Gamma \vdash$*' in the judgments. Let* D *and* B *be data-types as in Definition 2.5, so we have*

$$\begin{aligned} \mathsf{D} &= \forall X.(\overline{D}^1[\mathsf{D}/D] \to X) \to \ldots \to (\overline{D}^n[\mathsf{D}/D] \to X) \to X, \\ \mathsf{B} &= \forall X.(\overline{B}^1[\mathsf{B}/B] \to X) \to \ldots \to (\overline{B}^m[\mathsf{B}/B] \to X) \to X. \end{aligned}$$

We have the following rules. Let A *be an arbitrary type. We denote by* $\neg_A X$ *the type* $X \to A$*.*

The rule (It-CBN) is:

$$\frac{f_1 : \overline{D}^1[\mathsf{B}/D] \to \mathsf{B}\ \ldots\ f_n : \overline{D}^n[\mathsf{B}/D] \to \mathsf{B}\quad c_1 : \overline{B}^1[\mathsf{B}/B] \to A\ \ldots\ c_m : \overline{B}^m[\mathsf{B}/B] \to A\quad d : \mathsf{D}}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,f_1 \ldots f_n\,d\,c_1 \ldots c_m : A}$$

The rule (It-CBV) is

$$\frac{f_1 : \overline{D}^1[\mathsf{B}/D] \to \neg_A\neg_A\mathsf{B}\ \ldots\ f_n : \overline{D}^n[\mathsf{B}/D] \to \neg_A\neg_A\mathsf{B}\quad c : \neg_A\mathsf{B}\quad d : \mathsf{D}}{\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,f_1 \ldots f_n\,c\,d : A}$$

The idea of the rule (It-CBN) is:

$$\frac{f_1 : \overline{D}^1[\mathsf{B}/D] \to \mathsf{B}\ \ldots\ f_n : \overline{D}^n[\mathsf{B}/D] \to \mathsf{B}}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,f_1 \ldots f_n : \mathsf{D} \to \mathsf{B}}$$

which is indeed what we get if we use a type variable X for A and we abstract over $c_1, \ldots, c_m$ and X and d.

The idea of the rule (It-CBV) is

$$\frac{f_1 : \overline{D}^1[\mathsf{B}/D] \to \neg_A\neg_A\mathsf{B}\ \ldots\ f_n : \overline{D}^n[\mathsf{B}/D] \to \neg_A\neg_A\mathsf{B}}{\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,f_1 \ldots f_n : \forall X.\neg_X\mathsf{B} \to \neg_X\mathsf{D}}$$

which is indeed what we get if we use a type variable X for A and we abstract over c, d and X.

The reason for writing the rules (It-CBN) and (It-CBV) in the form of Definition 2.8 is that we want to have these terms in "fully applied" form, because it is easier to deal with in the analysis of the computations.

We now define the reduction rule for $\lambda 2\mu$It, which is the union of the β-rule and the rules for call-by-value and call-by-name.

Definition 2.9. *We recall the equation*

$$\mathsf{D} = \forall X.(\overline{D}^1[\mathsf{D}/D] \to X) \to \ldots \to (\overline{D}^n[\mathsf{D}/D] \to X) \to X.$$

(1) The rule for call-by-name-iteration, $\to_n$, *is*

$$\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, d\, c_1 \ldots c_m \quad \to_n \quad d\, t^1 \ldots t^n$$

where, abbreviating $f_1 \ldots f_n$ *to* $\overline{f}$ *and* $c_1 \ldots c_m$ *to* $\overline{c}$,

$$\begin{array}{rcl} t^i & := & \lambda x_1 \ldots x_{\mathsf{ar}(i)}.f_i\, \mathbf{x}_1 \ldots \mathbf{x}_{\mathsf{ar}(i)}\, \overline{c} \\ \textit{with } \mathbf{x}_j & = & x_j \textit{ if } D^i_j \neq D \\ \mathbf{x}_j & = & \lambda z_1 \ldots z_m.\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f} x_j\, z_1 \ldots z_m \textit{ if } D^i_j = D. \end{array}$$

(2) The rule for call-by-value-iteration, $\to_v$, *is*

$$\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, c\, d \quad \to_v \quad d\, s^1_1 \ldots s^n_1$$

where, abbreviating $f_1 \ldots f_n$ *to* $\overline{f}$, *and* $r_1 \ldots r_{\mathsf{ar}(i)}$ *to* $\overline{r}$,

$$\begin{array}{rcl} s^i_j & := & \lambda x_j \ldots x_{\mathsf{ar}(i)}, r_1 \ldots r_{j-1}.s^i_{j+1}\, x_{j+1} \ldots x_{\mathsf{ar}(i)}\, r_1 \ldots r_{j-1}\, x_j \textit{ if } D^i_j \neq D \\ s^i_j & := & \lambda x_j \ldots x_{\mathsf{ar}(i)}, r_1 \ldots r_{j-1}.\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, (s^i_{j+1}\, x_{j+1} \ldots x_{\mathsf{ar}(i)}\, r_1 \ldots r_{j-1})\, x_j \textit{ if } D^i_j = D, \\ s^i_{\mathsf{ar}(i)+1} & := & \lambda \overline{r}.f_i\, \overline{r}\, c \end{array}$$

The call-by-value reduction rule has to take care that, in case data-type D has a constructor with more than one recursive sub-term, *all recursive sub-terms* will be evaluated. To understand better the terms s^i_j in the $\to_v$ rule we have the following lemma.

Lemma 2.10. *In the definition of* $\to_v$, *for* $1 \leq j \leq \mathsf{ar}(i)$,

$$s^i_{j+1} : D^i_{j+1} \to \ldots \to D^i_{\mathsf{ar}(i)} \to D^i_1[\mathsf{B}/D] \to \ldots \to D^i_j[\mathsf{B}/D] \to A.$$

Proof. For $j = \mathsf{ar}(i)$, the result is immediate. For other j, the result follows from the result for $j+1$, making a case distinction between $D^i_j[\mathsf{B}/D] = D^i_j$ or $D^i_j[\mathsf{B}/D] = \mathsf{B}$. □

Definition 2.11. *The reduction rule of* $\lambda 2\mu$*It,* $\to_{\beta vn}$, *is the union of the usual* β*-rule:*

$$(\lambda x.M)P \to_\beta M[P/x],$$

and the rules for call-by-name-iteration, $\to_n$, *and* call-by-value-iteration, $\to_v$ *(see Definition 2.9). These are closed under the term constructors, so we have*

$$\frac{M \to_\beta N}{M \to_{\beta vn} N} \quad \frac{M \to_v N}{M \to_{\beta vn} N} \quad \frac{M \to_n N}{M \to_{\beta vn} N} \quad \frac{M \to_{\beta vn} N}{M\,P \to_{\beta vn} N\,P} \quad \frac{M \to_{\beta vn} N}{P\,M \to_{\beta vn} P\,N} \quad \frac{M \to_{\beta vn} N}{\lambda x.M \to_{\beta vn} \lambda x.N}$$

$$\frac{M \to_{\beta vn} N}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, M\, \overline{c} \to_{\beta vn} \mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, N\, \overline{c}} \quad \frac{\overline{f} \to_{\beta vn} \overline{f'}}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, M\, \overline{c} \to_{\beta vn} \mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f'}\, M\, \overline{c}} \quad \frac{\overline{c} \to_{\beta vn} \overline{c'}}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, M\, \overline{c} \to_{\beta vn} \mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, M\, \overline{c'}}$$

and similar rules for $\mathsf{Itcbv}^{D}_{B}\, \overline{f}\, c\, d$.

As usual, we define the transitive reflexive closure *of* $\to_{\beta vn}$ *as* $\twoheadrightarrow_{\beta vn}$ *and the* transitive symmetric closure *of* $\twoheadrightarrow_{\beta vn}$ *as* $=_{\beta vn}$.

Lemma 2.12. *The reduction relation* $\longrightarrow\!\!\!\!\!\rightarrow_{\beta vn}$ *is* Church-Rosser *on* Term*: If* $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} N_1$ *and* $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} N_2$*, then* $N_1 \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} Q$ *and* $N_2 \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} Q$ *for some* Q*.*

As a consequence, we have confluence*: if* $M =_{\beta vn} N$*, then* $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} Q$ *and* $N \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} Q$ *for some* Q*.*

Proof. The well-know proof method of Takahashi [23] can be applied: define the notion of *development* in the standard way: $M \Rightarrow_{\beta vn} N$ capturing the idea of contracting zero or more redexes in M in parallel to obtain N. One can prove that $M \rightarrow_{\beta vn} N$ implies $M \Rightarrow_{\beta vn} N$ and $M \Rightarrow_{\beta vn} N$ implies $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} N$. Then define M^* as the complete development of M and prove (1) $M \Rightarrow_{\beta vn} M^*$ and (2) if $M \Rightarrow_{\beta vn} N$, then $N \Rightarrow_{\beta vn} M^*$. From that, Church-Rosser follows.

The confluence property follows in the standard way from Church-Rosser. □

The constructors of data types can now all be defined. We easily check that the constructors as defined in Definition 2.5 are all well-typed. As an example, we give the definitions of the constructors for the data types of Example 2.4.

Example 2.13. *(1) Remember that*

$$\mathsf{N} = \forall X.X \to (\mathsf{N} \to X) \to X.$$

The constructors for N *are as follows. (We write types with the* λ*-abstractions for easier type checking):*

$$\begin{array}{rcl} \mathsf{zero} & := & \lambda z : X.\lambda s : \mathsf{N} \to X.z : \mathsf{N} \\ \mathsf{suc} & := & \lambda n : \mathsf{N}.\lambda z : X.\lambda s : \mathsf{N} \to X.s\, n : \mathsf{N} \to \mathsf{N} \end{array}$$

(2) Remember that

$$\mathsf{L}_A = \forall X.X \to (A \to \mathsf{L}_A \to X) \to X.$$

The constructors for L_A *are as follows.*

$$\begin{array}{rcl} \mathsf{nil} & := & \lambda n : X.\lambda c : A \to \mathsf{L}_A \to X.n \\ \mathsf{cons} & := & \lambda a : A.\lambda l : \mathsf{L}_A.\lambda n : X.\lambda c : A \to \mathsf{L}_A \to X.c\, a\, l \end{array}$$

(3) Remember that

$$\mathsf{BTree}_A = \forall X.(A \to X) \to (\mathsf{BTree}_A \to \mathsf{BTree}_A \to X) \to X$$

The constructors for BTree_A *are as follows.*

$$\begin{array}{rcl} \mathsf{leaf} & := & \lambda a : A.\lambda l : A \to X.\lambda j : \mathsf{BTree}_A \to \mathsf{BTree}_A \to X.l\, a \\ \mathsf{join} & := & \lambda t_1, t_2 : \mathsf{BTree}_A.\lambda l : A \to X.\lambda j : \mathsf{BTree}_A \to \mathsf{BTree}_A \to X.j\, t_1\, t_2 \end{array}$$

Lemma 2.14 (Subject Reduction)**.** *The reduction rules preserve types: if* $\Gamma \vdash M : A$ *and* $M \rightarrow_{\beta vn} N$*, then* $\Gamma \vdash N : A$

Proof. The proof is by induction on the structure of M. The only interesting cases are when M is itself a redex. For the β-case, the result follows from a Substitution Lemma: If $\Gamma_1, x : A, \Gamma_2 \vdash P : B$ and $\Gamma_1 \vdash R : A$, then $\Gamma_1, \Gamma_2 \vdash P[R/x] : B$. This is proved by induction on the derivation of $\Gamma_1, x : A, \Gamma_2 \vdash P : B$. For the call-by-name case, it is a straightforward check that $d\, t^1 \dots t^n$ has the same type as $\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, d\, \overline{c}$. For the call-by-value case, we use Lemma 2.10 to check that $d\, s^1_1 \dots s^n_1$ has the same type as $\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, \overline{f}\, c\, d$. □

The system $\lambda 2\mu\mathrm{It}$ is intended as a computational model for functional programming. The evaluation relation $\rightarrow^{wh}$ is *weak head reduction*, which means head reduction that does not reduce under a λ. The reduction relation $\rightarrow_{\beta vn}$ can reduce everywhere inside a term, but the relation $\rightarrow^{wh}$ evaluates a term in a completely deterministic way. Weak head reduction is usually seen as a call-by-name evaluation strategy, but in our case it can mimic both call-by-name and call-by-value

evaluation, depending on the definition one chooses for the function: using call-by-name iteration or call-by-value iteration.

Definition 2.15. *We define the evaluation relation* $\to^{wh}$ *by*

$$\frac{}{(\lambda x.M)P \to^{wh} M[P/x]} \qquad \frac{M \to^{wh} N}{M\,P \to^{wh} N\,P}$$

$$\frac{}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, d\, c_1 \ldots c_m \to^{wh} d\, t^1 \ldots t^n} \qquad \frac{}{\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, c\, d \to^{wh} d\, s^1_1 \ldots s^n_1}$$

where the right hand sides in the rules for Itcbn *and* Itcbv *are as in Definition 2.9.*

3. Examples

3.1. **Natural numbers.** We consider the data-type N and we give two definitions of the addition function, one call-by-value and one call-by-name. The constructors are

$$\begin{array}{rcl} \mathsf{zero} & := & \lambda z.\lambda s.z \\ \mathsf{suc} & := & \lambda n.\lambda z.\lambda s.s\,n \end{array}$$

Definition 3.1. *For first order data types we define the* interpretation *of a data element d as a λ-term in the standard way, and use notation $\langle d\rangle$ for these terms. For natural numbers, this amounts to*

$$\begin{array}{rcl} \langle 0\rangle & := & \mathsf{zero} \\ \langle n+1\rangle & := & \mathsf{suc}\,\langle n\rangle \end{array}$$

Note that the representation of data is not in weak-head normal form. For example, $\langle 1\rangle = (\lambda n\, z\, s.s\, n)\mathsf{zero}$, which evaluates to $\lambda z\, s.s\,\mathsf{zero}$. Our original representation of natural numbers as Scott numerals, in Definition 1.1, represents a number n as $\underline{n}$. It is easy to verify that, for $n \in \mathbf{N}$, $\underline{n}$ is in weak head normal form and $\langle n\rangle \longrightarrow\!\!\!\!\!\rightarrow^{wh} \underline{n}$.

Let B be some data type and let A be a type. The call-by-value iteration scheme for N is

$$\text{(It-CBV)}\quad \frac{f_1 : \neg_A\neg_A B \quad f_2 : B \to \neg_A\neg_A B \quad c : \neg_A B \quad n : \mathsf{N}}{\mathsf{Itcbv}\, f_1\, f_2\, c\, n : A}$$

With reduction rule

$$\mathsf{Itcbv}\, f_1\, f_2\, c\, n \quad \to_v \quad n\,(f_1\, c)\,(\lambda x : \mathsf{N}.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : \mathsf{B}.f_2\, y\, c)\, x)$$

We define addition in call-by-value style (taking B to be N and A a type variable X that we abstract over) as follows

$$\begin{array}{rcl} \mathsf{AddCBV} & : & \forall X.\mathsf{N} \to \mathsf{N} \to \neg_X\neg_X\mathsf{N} \\ \mathsf{AddCBV} & := & \lambda x\, y : \mathsf{N}.\lambda c : \neg_X\mathsf{N}.\mathsf{Itcbv}\, \hat{y}\, \widehat{\mathsf{suc}}\, c\, x \end{array}$$

with $\hat{y} := \lambda c.c\, y$, $\widehat{\mathsf{suc}} := \lambda n\, c.c(\mathsf{suc}\, n)$. Then we have the nice property:

$$\mathsf{AddCBV}\,\langle n\rangle\,\langle m\rangle\, c \longrightarrow\!\!\!\!\!\rightarrow^{wh} c\,\langle n+m\rangle.$$

As a matter of fact, $\mathsf{AddCBV}\,\langle n\rangle\, y\, c$ evaluates $\langle n\rangle$ completely (in call-by-value style) and returns $c\,(\mathsf{suc}^n\, y)$.

For call-by-name, we let B be a data type with

$$\mathsf{B} = \forall X.(\overline{B}^1[\mathsf{B}/B] \to X) \to \ldots \to (\overline{B}^m[\mathsf{B}/B] \to X) \to X.$$

We abbreviate the expressions $\overline{B}^i[\mathsf{B}/B]$ further by writing $\Phi^i(\mathsf{B})$ for it. So we have

$$\mathsf{B} = \forall X.(\Phi^1(\mathsf{B}) \to X) \to \ldots \to (\Phi^m(\mathsf{B}) \to X) \to X.$$

The call-by-name iterator for N is (for A an arbitrary type):

$$\text{(It-CBN)}\quad \frac{f_1 : \mathsf{B} \quad f_2 : \mathsf{B} \to \mathsf{B} \quad c_1 : \Phi^1(\mathsf{B}) \to A \ldots c_m : \Phi^m(\mathsf{B}) \to A \quad n : \mathsf{N}}{\mathsf{Itcbn}\, f_1\, f_2\, n\, c_1 \ldots c_m : A}$$

With reduction rule:

$$\mathsf{Itcbn}\, f_1\, f_2\, n\, \overline{c} \quad \to_n \quad n\, (f_1\, \overline{c})\, (\lambda x : \mathsf{N}. f_2\, (\lambda \overline{z}. \mathsf{Itcbn}\, f_1\, f_2\, x\, \overline{z}) \overline{c})$$

We now define addition in call-by-name style (taking B to be N) as follows.

$$\begin{array}{lcl} \mathsf{AddCBN} & : & \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ \mathsf{AddCBN} & := & \lambda x\, y : \mathsf{N}. \lambda c_1 : X. \lambda c_2 : \mathsf{N} \to X. \mathsf{Itcbn}\, y\, \mathsf{suc}\, x\, c_1\, c_2 \end{array}$$

Then

$$\begin{array}{rcl} \mathsf{AddCBN}\, \mathsf{zero}\, y\, c_1\, c_2 & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & y\, c_1\, c_2 \\ \mathsf{AddCBN}\, \langle n+1 \rangle\, y\, c_1\, c_2 & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{Itcbn}\, y\, \mathsf{suc}\, \langle n+1 \rangle\, c_1\, c_2 \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{suc}\, (\lambda z_1 z_2. \mathsf{Itcbn}\, y\, \mathsf{suc}\, \langle n \rangle\, z_1\, z_2)\, c_1\, c_2 \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & c_2\, (\lambda z_1 z_2. \mathsf{Itcbn}\, y\, (\mathsf{suc}\, \langle n \rangle)\, z_1\, z_2) \\ & =_{\beta vn} & c_2\, (\mathsf{AddCBN}\, \langle n \rangle\, y) \end{array}$$

If c_2 is a variable, weak head reduction stops here. We note that $\mathsf{AddCBN}\, t\, q$ computes the first 'pattern' of the output and passes it on to the appropriate continuation. We can prove by induction on n that

$$\mathsf{AddCBN}\, \langle n \rangle\, y =_{\beta\eta vn} \mathsf{suc}^n(y),$$

so $\mathsf{AddCBN}\, \langle n \rangle \langle m \rangle$ and $\langle n + m \rangle$ are observationally equal.

Using addition, we can define multiplication in the usual way. We show the details in call-by-value style. We use $\mathsf{B} := \mathsf{N}$ in the call-by-value scheme and given a type X and $y : \mathsf{N}$ we define

$$\begin{array}{lcl} f_1 & := & \widehat{\mathsf{zero}} : \neg_X \neg_X \mathsf{N} \\ f_2 & := & \mathsf{AddCBV}\, y : \mathsf{N} \to \neg_X \neg_X \mathsf{N} \end{array}$$

Then we define

$$\mathsf{MultCBV} := \lambda x\, y : \mathsf{N}. \lambda c : \neg_X \mathsf{N}. \mathsf{Itcbv}\, \widehat{\mathsf{zero}}\, (\mathsf{AddCBV}\, y)\, c\, x : \mathsf{N} \to \mathsf{N} \to \neg_X \neg_X \mathsf{N}.$$

One now verifies that $\mathsf{MultCBV}\, \langle n \rangle\, \langle m \rangle\, c \longrightarrow\!\!\!\!\!\rightarrow^{wh} c\, \langle n * m \rangle$.

3.2. **Storage operators.** Storage operators have been introduced by Krivine [12] and have been studied extensively by Krivine and Nour (e.g. see [14, 18]. They were invented to mimic call-by-value using call-by-name evaluation. Let us first cast the concept in the setting of $\lambda 2\mu$It. The idea of a storage operator, Stor, for the natural numbers N is the following: for every $n \in \mathbf{N}$ there is a term t_n such that, given $f : \mathsf{N} \to A$ and $M : \mathsf{N}$ with $M =_{\beta vn} \langle n \rangle$,

$$\mathsf{Stor}\, M\, f \longrightarrow\!\!\!\!\!\rightarrow^{wh} f\, t_n.$$

This means that first M is reduced to some kind of 'standard form' t_n, that is independent of f, and then this term is fed to f. So, in case we want to compute $f\, M$ call-by-value, we compute $\mathsf{Stor}\, M\, f$, because it will first compute M and then feed this into f.

The simplest form of a storage operator is where t_n is just $\langle n \rangle$. In the work of Krivine and Nour, this is not always the case, so therefore the refinement in the notion as given above. In $\lambda 2\mu$It we can define storage operators for all data types, showing that we can really do call-by-value computation in the system. Just like in [13], storage operators are of a double-negated type.

Definition 3.2. *A* storage operator *for data type* D *is a term* $\mathsf{Stor} : \mathsf{D} \to \forall X. \neg_X \neg_X X \mathsf{D}$ *satisfying, for all* M *with* $M =_{\beta vn} \langle d \rangle$,

$$\mathsf{Stor}\, M\, f \longrightarrow\!\!\!\!\!\rightarrow^{wh} f\, \langle d \rangle.$$

As the property also holds for a variable f and the evaluation $\to^{wh}$ is deterministic, a storage operator must *first* compute M to a normal form and then pass it to f (for any f).

We now define a storage operator for N, Stor_N, and its inverse $\mathsf{Unstor}_\mathsf{N}$.

Definition 3.3. *Define the two terms* $\mathsf{Stor}_\mathsf{N} : \mathsf{N} \to \forall X.\neg_X\neg_X X\mathsf{N}$ *and* $\mathsf{Unstor}_\mathsf{N} : (\forall X.\neg_X\neg_X X\mathsf{N}) \to \mathsf{N}$ *as follows.*

$$\begin{array}{rcl} \mathsf{Stor}_\mathsf{N} & := & \lambda n\lambda f.\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,(\lambda m\, b.b(\mathsf{suc}\, m))\, f\, n \\ \mathsf{Unstor}_\mathsf{N} & := & \lambda f.\lambda z\, s.f\,(\lambda n.n\, z\, s) \end{array}$$

with $\widehat{\mathsf{zero}} := \lambda c.c\,\mathsf{zero}$.

To prove that Stor_N is a storage operator and that $\mathsf{Unstor}_\mathsf{N}$ is its inverse, we first prove two auxiliary lemmas. We also need another definition first.

Lemma 3.4. *(1) For all terms M, if $M =_{\beta vn} \langle 0\rangle$, then for all P, Q, $M\,P\,Q \longrightarrow\!\!\!\!\!\rightarrow^{wh} P$.*

(2) For all $n \in \mathbf{N}$ and terms M, if $M =_{\beta vn} \langle n+1\rangle$, then for all P, Q, $M\,P\,Q \longrightarrow\!\!\!\!\!\rightarrow^{wh} Q\,R$ for some $R =_{\beta vn} \langle n\rangle$.

Proof. (1) If $M =_{\beta vn} \langle 0\rangle$, then $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} \langle 0\rangle$ due to confluence (Lemma 2.12). So $M \longrightarrow\!\!\!\!\!\rightarrow^{wh} \lambda z.M'$ with $M' \longrightarrow\!\!\!\!\!\rightarrow^{wh} \lambda s.M''$ with $M'' \longrightarrow\!\!\!\!\!\rightarrow^{wh} z$. So for all P, Q, $M\,P\,Q \longrightarrow\!\!\!\!\!\rightarrow^{wh} P$.

(2) If $M =_{\beta vn} \langle n+1\rangle$, then $M \longrightarrow\!\!\!\!\!\rightarrow_{\beta vn} \lambda z\, s.s\,\langle n\rangle$ due to confluence (Lemma 2.12). So $M \longrightarrow\!\!\!\!\!\rightarrow^{wh} \lambda z.M'$ with $M' \longrightarrow\!\!\!\!\!\rightarrow^{wh} \lambda s.M''$ with $M'' \longrightarrow\!\!\!\!\!\rightarrow^{wh} s\,M'''$ with $M''' =_{\beta vn} \langle n\rangle$. So for all P, Q, $M\,P\,Q \longrightarrow\!\!\!\!\!\rightarrow^{wh} Q\,R$ for some $R =_{\beta vn} \langle n\rangle$.

□

To understand the evaluation behavior of $\mathsf{Stor}_\mathsf{N}\,M\,f$, we now study more closely, for arbitrary f and M, the term $\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,(\lambda m\, b.b(\mathsf{suc}\, m))\, f\, M$.

Definition 3.5. *We abbreviate $f_2 := \lambda m\, b.b(\mathsf{suc}\, m)$, so $f_2 : \mathsf{N} \to \neg_X\neg_X\mathsf{N}$, and we define, for $f : \neg_X\mathsf{N}$ and $p \in \mathbf{N}$, the term $F^p(f)$ as follows.*

$$\begin{array}{rcl} F^0(f) & := & f \\ F^{p+1}(f) & := & \lambda y.f_2\, y\,(F^p(f)). \end{array}$$

Then $F^p(f) : \neg_X\mathsf{N}$ for all $p \in \mathbf{N}$.

Note that $F^p(f)$ is a term for every p and f, so it should not be read as an iterated application of F to f. In some sense, it is an iterated application of f_2 to f. To emphasize that $F^p(f)$ is a term, we will often write it in brackets: $(F^p(f))$ when it occurs as a sub-term.

Lemma 3.6. *For f_2 as in Definition 3.5 and arbitrary $g : \neg_X\mathsf{N}$,*

(1) For all $p, n \in \mathbf{N}$, $F^p(g)\,\langle n\rangle \longrightarrow\!\!\!\!\!\rightarrow^{wh} g\,\langle p+n\rangle$.

(2) If $M =_{\beta vn} \langle n+1\rangle$, then

$$\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\, f_2\,(F^p(g))\, M \longrightarrow\!\!\!\!\!\rightarrow^{wh} \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\, f_2\,(F^{p+1}(g))\, M' \text{ for some } M' =_{\beta vn} \langle n\rangle.$$

If $M = \langle n+1\rangle$, then even $M' = \langle n\rangle$.

Proof. (1) We prove it by induction on p:

- $F^0(g)\,\langle n\rangle = g\,\langle n\rangle$.
- $F^{p+1}(g)\,\langle n\rangle = (\lambda y.f_2\, y\,(F^p(g)))\,\langle n\rangle \longrightarrow\!\!\!\!\!\rightarrow^{wh} f_2\,\langle n\rangle\,(F^p(g)) \longrightarrow\!\!\!\!\!\rightarrow^{wh} F^p(g)\,(\mathsf{suc}\langle n\rangle) \longrightarrow\!\!\!\!\!\rightarrow^{wh}$ (by IH) $g\,\langle p+1+n\rangle$.

(2) Assume $M =_{\beta vn} \langle n+1 \rangle$ and use Lemma 3.4.

$$\begin{array}{lll} \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^p(g))\,M & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & M\,(\widehat{\mathsf{zero}}\,(F^p(g)))(\lambda x.\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(\lambda y.f_2\,y\,(F^p(g)))\,x) \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & (\lambda x.\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^{p+1}(g))\,x)\,M' \qquad (\text{for some } M' =_{\beta vn} \langle n \rangle) \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^{p+1}(g))\,M' \qquad (\text{for some } M' =_{\beta vn} \langle n \rangle) \end{array}$$

We immediately check that, in case $M = \langle n+1 \rangle$, then $M' = \langle n \rangle$.

□

Lemma 3.7. *The term* $\mathsf{Stor_N}$ *is a storage operator for* N *and* $\mathsf{Unstor_N}(\mathsf{Stor_N}\,\langle n \rangle) =_{\beta vn} \langle n \rangle$ *for all* $n \in \mathbf{N}$.

Proof. We need to prove that for any f and $M =_{\beta vn} \langle n \rangle$, $\mathsf{Stor_N}\,M\,f \longrightarrow\!\!\!\!\!\rightarrow^{wh} f\langle n \rangle$. Let f_2 as in Definition 3.5. First note that

$$\mathsf{Stor_N}\,M\,f \longrightarrow\!\!\!\!\!\rightarrow^{wh} \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^0(f))\,M$$

To prove that $\mathsf{Stor_N}$ is a storage operator, we prove by induction on $n \in \mathbf{N}$ that if $M =_{\beta vn} \langle n \rangle$, then for all $p \in \mathbf{N}$:

$$\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^p(f))\,M \longrightarrow\!\!\!\!\!\rightarrow^{wh} f\,\langle n+p \rangle.$$

- Suppose $M =_{\beta vn} \mathsf{zero}$. Then $\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^p(f))\,M \longrightarrow\!\!\!\!\!\rightarrow^{wh} \widehat{\mathsf{zero}}\,(F^p(f)) \longrightarrow\!\!\!\!\!\rightarrow^{wh} F^p(f)\,\mathsf{zero} \longrightarrow\!\!\!\!\!\rightarrow^{wh} f\,\langle p \rangle$ (by Lemma 3.6).
- Suppose $M =_{\beta vn} \langle n+1 \rangle$. Then, by Lemma 3.6,

$$\begin{array}{lll} \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^p(f))\,M & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^{p+1}(f))\,M' \text{ (for some } M' =_{\beta vn} \langle n \rangle) \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & f\,\langle n+1+p \rangle \text{ (by IH).} \end{array}$$

To prove that $\mathsf{Unstor_N}(\mathsf{Stor_N}\,\langle n \rangle) =_{\beta vn} \langle n \rangle$ for all $n \in \mathbf{N}$, we show that for variables z, s,

$$\begin{array}{rcl} \mathsf{Unstor_N}(\mathsf{Stor_N}\,\langle 0 \rangle)\,z\,s & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & z \\ \mathsf{Unstor_N}(\mathsf{Stor_N}\,\langle n+1 \rangle)\,z\,s & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & s\,\langle n \rangle \end{array}$$

First note that

$$\mathsf{Unstor_N}(\mathsf{Stor_N}\,M\,z\,s \longrightarrow\!\!\!\!\!\rightarrow^{wh} \mathsf{Stor_N}\,M\,(\lambda n.n\,z\,s) \longrightarrow\!\!\!\!\!\rightarrow^{wh} \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^0(\lambda n.n\,z\,s))\,M$$

So the first follows from $\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^0(\lambda n.n\,z\,s))\,\mathsf{zero} \longrightarrow\!\!\!\!\!\rightarrow^{wh} z$. For the second, we prove by induction on n that for all p

$$\mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^p(\lambda n.n\,z\,s))\,\langle n+1 \rangle \rightarrow^{wh} s\,\langle n+p \rangle$$

We use Lemma 3.6:

$$\begin{array}{lll} \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^p(\lambda n.n\,z\,s))\,\langle n+1 \rangle & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{Itcbv}\,\widehat{\mathsf{zero}}\,f_2\,(F^{p+1}(\lambda n.n\,z\,s))\,\langle n \rangle \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & s\,\langle n+p+1 \rangle \text{ (by IH)} \end{array}$$

□

3.3. **Lists over natural numbers.** We now consider lists of natural numbers. We are especially interested in using the continuations to define a function that multiplies a list of natural numbers in a special way: as soon as we encounter a 0, we should 'jump out of the recursion' and just return 0. So we consider the following data-type of lists over N.

$$\mathsf{L_N} := \mu T.\forall X.X \to (\mathsf{N} \to T \to X) \to X$$

The constructors are

$$\begin{array}{rcl} \mathsf{nil} & := & \lambda n.\lambda c.n \\ \mathsf{cons} & := & \lambda p.\lambda l.\lambda n.\lambda c.c\,p\,l \end{array}$$

For defining a function on $\mathsf{L_N}$, we can use the appropriate iteration schemes, which we give explicitly. Let A be a type and let B be a data type with

$$\mathsf{B} = \forall X.(\Phi^1(\mathsf{B}) \to X) \to \ldots \to (\Phi^m(\mathsf{B}) \to X) \to X.$$

$$\text{(It-CBN)}\quad \frac{f_1 : \mathsf{B} \quad f_2 : \mathsf{N} \to \mathsf{B} \to \mathsf{B} \quad c_1 : \Phi^1(\mathsf{B}) \to A \ldots c_m : \Phi^m(\mathsf{B}) \to A \quad l : \mathsf{L_N}}{\mathsf{Itcbn}\, f_1\, f_2\, l\, c_1 \ldots c_m : A}$$

$$\text{(It-CBV)}\quad \frac{f_1 : \neg_A\neg_A\mathsf{B} \quad f_2 : \mathsf{N} \to \mathsf{B} \to \neg\neg\mathsf{B} \quad c : \neg\mathsf{B} \quad l : \mathsf{L_N}}{\mathsf{Itcbv}\, f_1\, f_2\, c\, l : \bot}$$

With reduction rules

$$\begin{array}{lll} \mathsf{Itcbn}\, f_1\, f_2\, l\, \overline{c} & \to_n & l\,(f_1\,\overline{c})\,(\lambda x : \mathsf{N}.\lambda k : \mathsf{L_N}.f_2\,x\,(\lambda \overline{z}.\mathsf{Itcbn}\, f_1\, f_2\, k\, \overline{z})\overline{c}) \\ \mathsf{Itcbv}\, f_1\, f_2\, c\, l & \to_v & l\,(f_1\,c)\,(\lambda x : \mathsf{N}.\lambda k : \mathsf{B}.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : B.f_2\,x\,y\,c)\,k) \end{array}$$

In the previous Section, we have seen the definition of call-by-value multiplication, satisfying

$$\mathsf{MultCBV}\langle n\rangle\langle m\rangle r \longrightarrow\!\!\!\!\!\rightarrow^{wh} r\langle n * m\rangle \text{ (for } n, m \in \mathbf{N} \text{ and any term } r).$$

We now define $\mathsf{ListMult} : \mathsf{L_N} \to \neg\neg\mathsf{N}$ that multiplies all elements in a list l by iterating over l, and stopping immediately with output 0 when a 0 is encountered.

Example 3.8. *The function* $\mathsf{ListMult} : \mathsf{L_N} \to \neg\neg\mathsf{N}$ *is defined by, given* $l : \mathsf{L_N}$ *and* $r : \neg\mathsf{N}$*, defining*

$$\mathsf{ListMult}\, l\, r := \mathsf{Itcbn}\, f_1\, f_2\, l\,(r\,\mathsf{zero})\, r,$$

where

$$\begin{array}{lcl} f_1 & := & \lambda c_1\, c_2.c_2\,\mathsf{one} : \mathsf{N} \\ f_2 & := & \lambda x : \mathsf{N}.\lambda b : \mathsf{N}.\lambda c_1\, c_2.x\,(r\,\mathsf{zero})(\lambda z.b\, c_1\,(\lambda y : \mathsf{N}.\mathsf{MultCBV}\, x\, y\, c_2)) : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \end{array}$$

Then

$$\begin{array}{rcl} \mathsf{ListMult}\,\mathsf{nil} & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & f_1\,(r\,\mathsf{zero})\, r, \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & r\,\mathsf{one} \\ \mathsf{ListMult}\,(\mathsf{cons}\, a\, l') & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & f_2\, a\,(\lambda z_1\, z_2.\mathsf{Itcbn}\, f_1\, f_2\, l'\, z_1\, z_2)(r\,\mathsf{zero})\, r \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & a\,(r\,\mathsf{zero})(\lambda z.(\lambda z_1\, z_2.\mathsf{Itcbn}\, f_1\, f_2\, l'\, z_1\, z_2)(r\,\mathsf{zero})(\lambda y.\mathsf{MultCBV}\, a\, y\, r)) \\ \textit{if } a = \mathsf{zero} : & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & r\,\mathsf{zero} \\ \textit{if } a = \mathsf{suc}\, b : & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & (\lambda z.(\lambda z_1\, z_2.\mathsf{Itcbn}\, f_1\, f_2\, l'\, z_1\, z_2)(r\,\mathsf{zero})(\lambda y.\mathsf{MultCBV}\, a\, y\, r))b \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{Itcbn}\, f_1\, f_2\, l'\,(r\,\mathsf{zero})(\lambda y.\mathsf{MultCBV}\, a\, y\, r) \end{array}$$

If $l' = \mathsf{cons}\,\mathsf{zero}\, l''$*, we see that*

$$\begin{array}{rcl} \mathsf{ListMult}\,(\mathsf{cons}\, a\, l') & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{Itcbn}\, f_1\, f_2\,(\mathsf{cons}\,\mathsf{zero}\, l'')\,(r\,\mathsf{zero})(\lambda y.\mathsf{MultCBV}\, a\, y\, r) \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & \mathsf{zero}\,(r\,\mathsf{zero})(\lambda z.(\lambda z_1\, z_2.\mathsf{Itcbn}\, f_1\, f_2\, l'\, z_1\, z_2)(r\,\mathsf{zero})(\lambda y.\mathsf{MultCBV}\, a\, y\, r)) \\ & \longrightarrow\!\!\!\!\!\rightarrow^{wh} & r\,\mathsf{zero} \end{array}$$

So, when encountering a zero*, the recursion terminates immediately and the function returns* $r\,\mathsf{zero}$*, also escaping from the context of previously stacked elements of the list. These elements are stored in the continuations* $(\lambda y.\mathsf{MultCBV}\, a\, y\, r$ *in the example above), which is thrown away.*

4. Models

We now describe models of $\lambda 2\mu$It. In our models, types are interpreted as sets of untyped terms, Term, which is an extension of the untyped λ-calculus with constants for the iterators: $\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,d\,\overline{c}$ and $\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,c\,d$. So we do not add Itcbn and Itcbv as constants to Λ, but we add 'fully applied' terms $\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,d\,\overline{c}$ and $\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,\overline{f}\,c\,d$ with the proviso that the arity constraints are respected.

Definition 4.1. *A* model *for $\lambda 2\mu$It consists of a* collection of saturated sets $\mathrm{SAT} \subseteq \mathcal{P}(\mathsf{Term})$.

We say that $\mathrm{SAT} \subseteq \mathcal{P}(\mathsf{Term})$ *is a collection of saturated sets if*

(1) SAT *is closed under intersection:*

$$\text{if } S_i \in \mathrm{SAT} \text{ for all } i \in I, \text{ then } \bigcap_{i\in I} S_i \in SAT.$$

(2) SAT *is closed under function space: if $S, T \in \mathrm{SAT}$, then $S \to T \in \mathrm{SAT}$, where*

$$S \to T := \{M \in \mathsf{Term} \mid \forall N \in S(M\,N \in T)\}.$$

(3) Each $S \in \mathrm{SAT}$ is non-empty,

(4) Each $S \in \mathrm{SAT}$ is closed under evaluation $\to^{wh}$: if $M \in S$ and $M \to^{wh} N$, then $N \in S$,

(5) Each $S \in \mathrm{SAT}$ is closed under key-redex expansion*, or inverse evaluation: if $M \in S$, $N \in \mathsf{Term}$ and $N \to^{wh} M$, then $N \in S$,*

The inverse evaluation in the last item is also called *key-redex expansion*, because $N \to^{wh} M$ in case $N \to_{\beta vn} M$ with a reduction-step 'at the top'. This can be made more precise by observing that a $\to^{wh}$-step is of either one of the following shapes. (Where $\overline{Q}$ is an arbitrary, possibly empty, sequence of terms.)

(1) $(\lambda x : A.M)\,N\,\overline{Q} \to^{wh} M[N/x]\,\overline{Q}$

(2) $\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\,f_1 \ldots f_n\,d\,c_1 \ldots c_m\,\overline{Q} \to^{wh} d\,t^1 \ldots t^n\,\overline{Q}$, with $t^1 \ldots t^n$ as in Definition 2.9.

(3) $\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\,f_1 \ldots f_n\,c\,d\,\overline{Q} \to^{wh} d\,s^1_1 \ldots s^n_1\,\overline{Q}$ with $s^1_1 \ldots s^n_1$ as in Definition 2.9.

The following are immediate.

Lemma 4.2. *Let* SAT *be a collection of saturated sets and $S, T \in \mathrm{SAT}$. If $\forall N \in S(M[N/x] \in T)$, then $\lambda x.M \in S \to T$.*

Proof. Let $N \in S$ and $M[N/x] \in T$. Then $(\lambda x.M)N \to^{wh} M[N/x]$, so $(\lambda x.M)N \in T$. So, $\forall N \in S((\lambda x.M)N \in T)$. So $\lambda x.M \in S \to T$. □

Lemma 4.3. *Let* SAT *be a collection of saturated sets. With the ordering relation $\subseteq$,* SAT *is a complete lattice with a bottom element $\bot := \cap_{S\in\mathrm{SAT}} S$.*

Proof. That SAT is a complete lattice follows immediately from Definition 4.1. The equation for $\bot$ is also trivial, because we can take $n = 0$ and we see that $\bot$ occurs under the $\cap$ on the right hand side. □

As a corollary, by the Knaster-Tarski fixed-point theorem, we know that every monotone function $F : \mathrm{SAT} \to \mathrm{SAT}$ has a least fixed point, which can be defined as the intersection of all *prefixed-points* of F:

$$\mathrm{lfp}(F) = \bigcap \{S \in \mathrm{SAT} \mid F(S) \subseteq S\}.$$

We now define the interpretation of types as saturated sets: for A a type, we define $[\![A]\!]_\xi$, where $\xi : \mathsf{TVar} \to \mathrm{SAT}$ is a valuation of type variables.

Definition 4.4. *Given a valuation of type variables $\xi : \mathsf{TVar} \to \mathrm{SAT}$, we define the interpretation of types, $[\![A]\!]_\xi \in \mathrm{SAT}$ as follows.*

$$\begin{array}{rcl} [\![X]\!]_\xi & := & \xi(X) \\ [\![A \to B]\!]_\xi & := & [\![A]\!]_\xi \to [\![B]\!]_\xi \\ [\![\forall X.A]\!]_\xi & := & \cap_{S \in \mathrm{SAT}} [\![A]\!]_{\xi[X \mapsto S]} \end{array}$$

This is only part (a) of the definition of $[\![-]\!]$: we haven't given a definition for μ-types yet. Given the fixed-point theorem, the straightforward choice would be $[\![\mu X.A]\!]_\xi := \mathrm{lfp}(S \mapsto [\![A]\!]_{\xi[X \mapsto S]})$, which is well-defined, because X occurs positively in A (and so $S \mapsto [\![A]\!]_{\xi[X \mapsto S]}$ is monotone, thus the lfp exists). This interpretation works well for μ-types that are not a data type. However, for data types we also need some closure properties for the iterators Itcbn and Itcbv, so this definition is not sufficient.

First, we only give the interpretation for N, and then we proceed to the general case for data types. We allow ourselves some further notational conveniences, writing $\neg_X S$ also for the saturated set $S \to X$ (and also $\neg_X \neg_X S$ for $(S \to X) \to X$ etc.)

We recall the data-type N. Its constructors are

$$\begin{array}{rcl} \mathsf{zero} & := & \lambda z.\lambda s.z \\ \mathsf{suc} & := & \lambda n.\lambda z.\lambda s.s\, n \end{array}$$

Let B be a data type, which we assume to satisfy the following defining equation.

$$\mathsf{B} = \forall X.(\Phi^1(\mathsf{B}) \to X) \to \ldots \to (\Phi^m(\mathsf{B}) \to X) \to X.$$

The iterators for N are (for A an arbitrary type):

$$\text{(It-CBV)} \quad \frac{f_1 : \neg_A \neg_A \mathsf{B} \quad f_2 : \mathsf{B} \to \neg_A \neg_A \mathsf{B} \quad c : \neg_A \mathsf{B} \quad n : \mathsf{N}}{\mathsf{Itcbv}\, f_1\, f_2\, c\, n : A}$$

$$\text{(It-CBN)} \quad \frac{f_1 : \mathsf{B} \quad f_2 : \mathsf{B} \to \mathsf{B} \quad c_1 : \Phi^1(\mathsf{B}) \to A \ldots c_m : \Phi^m(\mathsf{B}) \to A \quad n : \mathsf{N}}{\mathsf{Itcbn}\, f_1\, f_2\, n\, c_1 \ldots c_m : A}$$

With reduction rules

$$\begin{array}{rcl} \mathsf{Itcbn}\, f_1\, f_2\, n\, \overline{c} & \to_n & n\,(f_1\,\overline{c})\,(\lambda x : \mathsf{N}.f_2\,(\lambda \overline{z}.\mathsf{Itcbn}\, f_1\, f_2\, x\, \overline{z})\overline{c}) \\ \mathsf{Itcbv}\, f_1\, f_2\, c\, n & \to_v & n\,(f_1\, c)\,(\lambda x : \mathsf{N}.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : B.f_2\, y\, c)\, x) \end{array}$$

Definition 4.5. *We define* $\mathrm{Nat} := \mathrm{lfp}(\mathcal{N})$*, the least fixed point of* $\mathcal{N}$*, where* $\mathcal{N} : \mathrm{SAT} \to \mathrm{SAT}$ *is the function defined by, for* $S \in \mathrm{SAT}$*:*

$$\mathcal{N}(S) := \{M \in \mathsf{Term} \mid M \in \cap_{T \in \mathrm{SAT}} T \to (S \to T) \to T \wedge M \text{ satisfies (I) and (II)}\}.$$

Here, the properties (I) and (II) are defined by

(I) $\forall B, X \in \mathrm{SAT}\ \forall f_1 \in \neg_X \neg_X B\ \forall f_2 \in B \to \neg_X \neg_X B\ \forall c \in \neg_X B, \mathsf{Itcbv}\, f_1\, f_2\, c\, M \in X$

(II) $\forall B, X \in \mathrm{SAT}\ \forall F_1 \ldots F_m \in \mathrm{SAT} \to \mathrm{SAT},$ *monotone*,
$\forall f_1 \in B\ \forall f_2 \in B \to B\ \forall c_1 \in F_1(B) \to X, \ldots, c_m \in F_m(B) \to X, \mathsf{Itcbn}\, f_1\, f_2\, M\, \overline{c} \in X$

Lemma 4.6. *$\mathcal{N}$ is monotone and, for $S \in \mathrm{SAT}$, $\mathcal{N}(S) \in \mathrm{SAT}$ as well.*

Proof. The monotonicity follows from the fact that S occurs positively in the expression $\cap_{T \in \mathrm{SAT}} T \to (S \to T) \to T$. That $\mathcal{N}(S) \in \mathrm{SAT}$ is verified by checking that (1) $\mathcal{N}(S)$ is closed under $\to_{\beta v n}$ and that (2) $\mathcal{N}(S)$ is closed under key-redex expansion.

(1) We have to show that, if $M \in \mathcal{N}(S)$ and $M \to^{wh} N$, then $N \in \cap_{T\in \mathrm{SAT}} T \to (S \to T) \to T$ and N satisfies (I) and (II). First of all, $\cap_{T\in \mathrm{SAT}} T \to (S \to T) \to T$ is a saturated set, so if M is in it and $M \to^{wh} N$, then N is in it as well.

Now we show that if $M \in \mathcal{N}(S)$ and $M \to^{wh} N$, then N satisfies (I). (The proof that N satisfies (II) is similar.) Let $B, X f_1, f_2, c$ be as in (I). Then $\mathsf{Itcbv}\, f_1\, f_2\, c\, M \in X$, so $M\,(f_1\, c)\,(\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : B.f_2\, y\, c)\, x) \in X$, by closure under $\to^{wh}$, so
$N\,(f_1\, c)\,(\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : B.f_2\, y\, c)\, x) \in X$, by closure under $\to^{wh}$, so $\mathsf{Itcbv}\, f_1\, f_2\, c\, N \in X$ by closure under key-redex expansion. We conclude that N satisfies (I).

(2) Suppose $M \in \mathcal{N}(S)$ and $N \to^{wh} M$. We have to show that $N \in \cap_{T\in \mathrm{SAT}} T \to (S \to T) \to T$ and N satisfies (I) and (II). First, $\cap_{T\in \mathrm{SAT}} T \to (S \to T) \to T$ is a saturated set, so N is in it, because M is in it and $M \to^{wh} N$.

Now we show that N satisfies (I). (The proof that N satisfies (II) is similar.) Let $B, X f_1, f_2, c$ be as in (I). Then $\mathsf{Itcbv}\, f_1\, f_2\, c\, M \in X$, so
$M\,(f_1\, c)\,(\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : B.f_2\, y\, c)\, x) \in X$, by closure under $\to^{wh}$,
so $N\,(f_1\, c)\,(\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y : B.f_2\, y\, c)\, x) \in X$, by closure under key-redex expansion, so $\mathsf{Itcbv}\, f_1\, f_2\, c\, N \in X$ by closure under key-redex expansion. We conclude that N satisfies (I).

□

Lemma 4.7. Nat *is a fixed point of* $S \mapsto \cap_{T\in \mathrm{SAT}} T \to (S \to T) \to T$*, that is*

$$\mathrm{Nat} = \cap_{T\in \mathrm{SAT}} T \to (\mathrm{Nat} \to T) \to T.$$

Proof. We have $\mathrm{Nat} \subseteq \cap_{T\in \mathrm{SAT}} T \to (\mathrm{Nat} \to T) \to T$ by definition of Nat.

For the reverse inclusion, suppose $M \in \cap_{T\in \mathrm{SAT}} T \to (\mathrm{Nat} \to T) \to T$. We claim that M satisfies (I) and (II). Then we are done, because then $M \in \mathrm{Nat}$. We only prove the first part of our claim: that M satisfies (I). (The proof that M satisfies (II) is similar.)

Let $B, X \in \mathrm{SAT}$, $f_1 \in \neg_X \neg_X B$, $f_2 \in B \to \neg_X \neg_X B$, $c \in \neg_X B$. Then

$$\mathsf{Itcbv}\, f_1\, f_2\, c\, M \to_v M(f_1\, c)\,(\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y.f_2\, y\, c)\, x).$$

We have $f_1\, c \in X$ and $\lambda y.f_2\, y\, c \in B \to X$ (using Lemma 4.2).

Given $t \in \mathrm{Nat}$, we know that t satisfies (I), so $\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y.f_2\, y\, c)\, t \in X$, so (using Lemma 4.2) $\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y.f_2\, y\, c)\, x \in \mathrm{Nat} \to X$. So,

$$M(f_1\, c)\,(\lambda x.\mathsf{Itcbv}\, f_1\, f_2\,(\lambda y.f_2\, y\, c)\, x) \in X$$

and by key-redex expansion also

$$\mathsf{Itcbv}\, f_1\, f_2\, c\, M \in X.$$

So M satisfies (I). □

Definition 4.8 (Case for N of Definition 4.4.)**.** *We define* $[\![\mathsf{N}]\!] := \mathrm{Nat}$.

For the general case, we assume D to be a data type in $\lambda 2\mu\mathrm{It}$ with n constructors, as in Definition 2.5. We have that D satisfies a defining equation as follows (for Ψ^j positive type schemes).

$$\mathsf{D} = \forall X.(\Psi^1(\mathsf{D}) \to X) \to \ldots \to (\Psi^n(\mathsf{D}) \to X) \to X.$$

Let B be another data type in $\lambda 2\mu\mathrm{It}$, which we assume to satisfy the following defining equation.

$$\mathsf{B} = \forall X.(\Phi^1(\mathsf{B}) \to X) \to \ldots \to (\Phi^m(\mathsf{B}) \to X) \to X.$$

The rule (It-CBV) is

$$\frac{f_1 : \Psi^1(\mathsf{B}) \to \neg_A \neg_A \mathsf{B} \;\ldots\; f_n : \Psi^n(\mathsf{B}) \to \neg_A \neg_A \mathsf{B} \quad c : \neg_A \mathsf{B} \quad d : \mathsf{D}}{\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, c\, d : A}$$

The rule (It-CBN) is:

$$\frac{f_1 : \Psi^1(\mathsf{B}) \to \mathsf{B} \ \ldots \ f_n : \Psi^n(\mathsf{B}) \to \mathsf{B} \quad c_1 : \Phi^1(\mathsf{B}) \to A \ \ldots \ c_m : \Phi^m(\mathsf{B}) \to A \quad d : \mathsf{D}}{\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, d\, c_1 \ldots c_m : A}$$

With reduction rules

$$\begin{aligned} \mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, d\, c_1 \ldots c_m &\to_n d\, t^1 \ldots t^n \\ \mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \ldots f_n\, c\, d &\to_v d\, s^1_1 \ldots s^n_1 \end{aligned}$$

as in Definition 2.9.

As we don't allow mutual inductive types, we may assume that in the definition of D, we only use types that have been previously defined. So, we may assume that the type scheme Ψ^j already has an interpretation $[\![\Psi^j]\!]$. This interpretation is a function $[\![\Psi^j]\!] : \mathrm{SAT} \to \mathrm{SAT}$, because Ψ^j is positive.

Definition 4.9. *We define* $\mathbf{D} := \mathrm{lfp}(\mathcal{D})$*, the least fixed point of* $\mathcal{D}$*, where* $\mathcal{D} : \mathrm{SAT} \to \mathrm{SAT}$ *is the function defined by, for* $S \in \mathrm{SAT}$*:*

$$\begin{aligned} \mathcal{D}(S) := \{M \in \mathsf{Term} \quad | \quad & M \in \cap_{T \in \mathrm{SAT}}([\![\Psi^1]\!](S) \to T) \to \ldots \to ([\![\Psi^n]\!](S) \to T) \to T \\ \wedge \quad & M \text{ satisfies (I) and (II)}\}. \end{aligned}$$

Here, the properties (I) and (II) are defined by

(I) $\forall B, X \in \mathrm{SAT}\ \forall f_1 \in [\![\Psi^1]\!](B) \to \neg_X \neg_X B, \ldots, f_n \in [\![\Psi^n]\!](B) \to \neg_X \neg_X B\ \forall c \in \neg_X B$
$\mathsf{Itcbv}\, \overline{f}\, c\, M \in X$

(II) $\forall B, X \in \mathrm{SAT}\ \forall F_1 \ldots F_m \in \mathrm{SAT} \to \mathrm{SAT},$ *monotone,*
$\forall f_1 \in [\![\Psi^1]\!](B) \to B, \ldots, f_n \in [\![\Psi^n]\!](B) \to B\ \forall c_1 \in F^1(B) \to A, \ldots, c_m \in F^m(B) \to A,$
$\mathsf{Itcbn}\, \overline{f}\, M\, \overline{c} \in X$

Lemma 4.10. *$\mathcal{D}$ is monotone and, for $S \in \mathrm{SAT}$, $\mathcal{D}(S) \in \mathrm{SAT}$ as well.*

Proof. The proof is very much the same as for $\mathcal{N}$, see Lemma 4.6. The monotonicity follows from the fact that S occurs positively in the expression

$$\cap_{T \in \mathrm{SAT}}([\![\Psi^1]\!](S) \to T) \to \ldots \to ([\![\Psi^n]\!](S) \to T) \to T.$$

That $\mathcal{D}(S) \in \mathrm{SAT}$ is verified by checking that (1) $\mathcal{D}(S)$ is closed under $\to_{\beta vn}$ and that (2) $\mathcal{D}(S)$ is closed under key-redex expansion. The verification of these two facts is the same as in the proof of Lemma 4.6. □

Lemma 4.11. *the set* $\mathbf{D}$ *is a fixed point of* $S \mapsto \cap_{T \in \mathrm{SAT}}([\![\Psi^1]\!](S) \to T) \to \ldots \to ([\![\Psi^n]\!](S) \to T) \to T$*, that is*

$$\mathbf{D} = \bigcap_{T \in \mathrm{SAT}} ([\![\Psi^1]\!](\mathbf{D}) \to T) \to \ldots \to ([\![\Psi^n]\!](\mathbf{D}) \to T) \to T.$$

Proof. The proof is very much like the proof for Nat in Lemma 4.7. We have
$\mathbf{D} \subseteq \cap_{T \in \mathrm{SAT}}([\![\Psi^1]\!](\mathbf{D}) \to T) \to \ldots \to ([\![\Psi^n]\!](\mathbf{D}) \to T) \to T$ by definition of $\mathbf{D}$. For the reverse inclusion, suppose $M \in \cap_{T \in \mathrm{SAT}}([\![\Psi^1]\!](\mathbf{D}) \to T) \to \ldots \to ([\![\Psi^n]\!](\mathbf{D}) \to T) \to T$. Then M satisfies (I) and (II), so we are done, because now $M \in \mathbf{D}$. The proofs are very similar to the ones of Lemma 4.7. □

Definition 4.12 (Final case of Definition 4.4.)**.** *We define* $[\![\mathsf{D}]\!] := \mathbf{D}$.

We now formally state and prove the soundness of our model.

Proposition 4.13. *Let a valuation* $\xi : \mathsf{TVar} \to \mathrm{SAT}$ *be given. If* $\Gamma \vdash M : B$*, where* $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ *and* $P_1 \in [\![A_1]\!]_\xi, \ldots, P_n \in [\![A_n]\!]_\xi$*, then*

$$M[P_1/x_1, \ldots, P_n/x_n] \in [\![B]\!]_\xi.$$

Proof. By induction on the derivation of $\Gamma \vdash M : B$. □

4.1. **Strong Normalization.** We now prove strong normalization of $\lambda 2\mu$It using a saturated sets / candidat de réducibilité argument, as originally developed by Girard and Tait. As a matter of fact, we have just defined a general model construction of which the strong normalization proof is almost an instance.

We define SN $\subset$ Term as the set of strongly normalizing pseudo-terms. Now we don't consider saturated sets as subsets of Term but as subsets of SN. Then we consider a very specific collection of saturated sets, called Sat.

Definition 4.14. *Change the Definition 4.1 by considering only the set* SN $\subset$ Term *and by changing the β-case of* closure under key redex expansion *to:*

$$\textit{If } M[N/x]\,\overline{Q} \in S \textit{ and } N \in \mathrm{SN}, \textit{ then } (\lambda x : A.M)\,N\,\overline{Q} \in S.$$

The refinement of the β-case is necessary, because, if $x \notin \mathrm{FV}(M)$, $(\lambda x : A.M)\,N\,\overline{Q}$ need not be SN, when $M[N/x]\,\overline{Q}$ is SN.

We now define Sat as a specific collection of saturated sets, in order to prove SN. (This is the usual construction.)

Definition 4.15. *The collection* Sat $\subset \mathcal{P}(\mathrm{SN})$ *is defined by* $S \in \mathrm{Sat}$ *if*

(1) for x a variable and $\overline{P}$ a sequence of terms from SN, $x\overline{P} \in S$,

(2) S is closed under weak-head reduction: if $M \in S$ and $M \to^{wh} N$, then $N \in S$,

(3) S is closed under key-redex expansion*: if $M \in S$ and $N \to_{\mathrm{key}} M$, then $N \in S$,*

where $N \to_{\mathrm{key}} M$ is defined by

(1) If $N \in \mathrm{SN}$, then $(\lambda x : A.M)\,N\,\overline{P} \to_{\mathrm{key}} M[N/x]\,\overline{P}$

(2) $\mathsf{Itcbn}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \dots f_n\, d\, c_1 \dots c_m \to_{\mathrm{key}} d\, t^1 \dots t^n$, with $t^1 \dots t^n$ as in Definition 2.9.

(3) $\mathsf{Itcbv}^{\mathsf{D}}_{\mathsf{B}}\, f_1 \dots f_n\, c\, d \to_{\mathrm{key}} d\, s^1_1 \dots s^n_1$ with $s^1_1 \dots s^n_1$ as in Definition 2.9.

Lemma 4.16. *The collection* Sat *is indeed a* collection of saturated sets *in the sense of Definition 4.14.*

Proof. All $S \in \mathrm{Sat}$ are non-empty and satisfy the required closure properties. We still have to show that Sat is closed under $\cap$ and $\to$. Closure under $\cap$ is immediate. For closure under $\to$:

Suppose $S, T \in \mathrm{Sat}$ and consider $S \to T := \{M \mid \forall N \in S(M\,N \in T)\}$. We verify the properties for Sat.

(1) for x a variable and $\overline{P}$ a sequence of terms from SN, $x\overline{P} \in S \to T$, because for all $N \in S$, we have $x\overline{P}N \in T$.

(2) If $M \in S \to T$ and $M \to^{wh} M'$, then for all $N \in S$, we have $M\,N \to^{wh} M'\,N$ and $M\,N \in T$, so $M'\,N \in T$. Therefore $M' \in S \to T$.

(3) If $M \in S \to T$ and $M' \to_{\mathrm{key}} M$, then for all $N \in S$, we have $M'\,N \to_{\mathrm{key}} M\,N$ and $M\,N \in T$, so $M'\,N \in T$. Therefore $M' \in S \to T$.

□

The collection of saturated sets Sat is the key to the proof of strong normalization. All definitions for general models apply immediately, including Proposition 4.13. Strong normalization is just a corollary.

Theorem 4.17. *All typable terms of $\lambda 2\mu$It are strongly normalizing.*

Proof. Say the term M is typable in the context $x_1 : A_1, \ldots, x_n : A_n$ with type B. We apply Proposition 4.13 with Sat as the collection of saturated sets. Especially, SN $\in$ Sat and for every $S \in$ Sat we have $S \subset$ SN. We choose a valuation $\xi : \mathsf{TVar} \to$ Sat and elements of each $[\![A_i]\!]_\xi$ to substitute for x_i in M as follows.

$$\begin{aligned} \xi(X) &:= \text{SN for each } X, \\ x_i &:= x_i. \end{aligned}$$

Note that the latter is allowed, because $x_i \in S$ for each $S \in SAT$. We conclude that $M \in [\![B]\!]_\xi$, so $M \in$ SN. □

As a corollary of Theorem 4.17, we obtain that all well-typed programs in continuation calculus (CC, see [9]) are normalizing. One can define a reduction preserving mapping from well-typed CC-terms to well-typed terms of $\lambda 2\mu$It. The constructors and iterators are translated in the straightforward way to constructors and iterators in $\lambda 2\mu$It. Non-circular programs in CC are of the shape $n.x_1.x_2.\ldots.x_p \longrightarrow u$, with n not occurring in u. Then we can translate n to $\lambda x_1\, x_2 \ldots x_p.u$. This gives a reduction preserving mapping, so we have that all well-typed programs of CC are strongly normalizing.

5. Conclusion and Future work

We have shown how to use Scott data types for functional programming by defining a polymorphic λ-calculus with recursive types in which one can represent data types in the Scott data types style. We have shown that the system is strongly normalizing. The recursive types are really 'fixed point' types': we only require that we have a solution to a type equation $A = \Phi(A)$ where A occurs positively in $\Phi(A)$. This solution we denote by $\mu X.\Phi(X)$, but it need not be smallest (nor largest). Scott data types are not very expressive, as they don't have any recursion 'built in' (as opposed to the Church data types). But this is also an advantage, because we can now add two separate iteration schemes: one for call-by-name and one for call-by-value. We have shown how these are used to define various functions in call-by-value or call-by-name style, where the evaluation relation is always weak head reduction. We have also shown how one can switch between call-by-name and call-by-value by doing a kind of CPS translation on the terms and a double negation translation on the types. This shows that a storage operator can be defined.

As future work, we may want to extend this to higher order data types. Also, it would be interesting to give a representation of Felleisen's $\mathcal{C}$ ([5]) in $\lambda 2\mu$It, comparable to Krivine's representation of $\mathcal{C}$ for Church data types in [13].

Also, in [13], storage operators are completely characterized by their type, also for so called "classical integers'. It is an interesting question if we can also capture storage operators by a type in $\lambda 2\mu$It. Krivine uses the system AF2, which is a λ-calculus with dependent types (second order predicate logic) where the proofs are untyped λ-terms. An interesting question is if we can understand better the terms and types of $\lambda 2\mu$It by developing an AF2 like system for Scott data types.

References

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

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

[3] 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.

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

[5] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[6] Peng Fu and Aaron Stump. Self types for dependently typed lambda encodings. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2014.
[7] Bram Geron and Herman Geuvers. Continuation calculus. In Ugo de'Liguoro and Alexis Saurin, editors, *Proceedings First Workshop on Control Operators and their Semantics, COS 2013, Eindhoven, The Netherlands, June 24-25, 2013.*, volume 127 of *EPTCS*, pages 66–85, 2013.
[8] H. Geuvers. The Church-Scott representation of inductive and coinductive data, 2014. Submitted.
[9] Herman Geuvers, Wouter Geraedts, Bram Geron, and Judith van Stegeren. A type system for continuation calculus. In Paulo Oliva, editor, *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014, Vienna, Austria, July 13, 2014.*, volume 164 of *EPTCS*, pages 1–17, 2014.
[10] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.
[11] Jan Martin Jansen. Programming in the λ-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.
[12] Jean-Louis Krivine. Opérateurs de mise en mémoire et traduction de Gödel. *Archive for Mathematical Logic*, 30(4):241–267, 1990.
[13] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. Pure Appl. Logic*, 68(1):53–78, 1994.
[14] Jean-Louis Krivine. A general storage theorem for integers in call-by-name lambda-calculus. *Theor. Comput. Sci.*, 129(1):79–94, 1994.
[15] Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda calculus. *Logical Methods in Computer Science*, 8(3), 2012.
[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] Karim Nour. A general type for storage operators. *Math. Log. Q.*, 41:505–514, 1995.
[19] Michel Parigot. Recursive programming with proofs. *Theor. Comput. Sci.*, 94(2):335–336, 1992.
[20] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
[21] 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 [4]).
[22] Aaron Stump. Directly reflective meta-programming. *Higher-Order and Symbolic Computation*, 22(2):115–144, 2009.
[23] Masako Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127, 1995.
[24] Ch. Wadsworth. Some unusual λ-calculus numeral systems. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

Radboud University Nijmegen and Eindhoven University of Technology, The Netherlands

E-mail address: herman@cs.ru.nl