On the Relationship Between Well-orderings in Intuitionistic Type Theory and Data Types Inductively Defined

Edward Hermann Haeusler and Oliver Bittel[1]

1. Introduction

When defining a data type, one often uses the well-known equation *datatype* = *objects* + *operations*. However, when the data-type is defined by induction (lists, trees, stacks, etc.) it is usual to construct the *objects* using the operations themselves on some *initial* elements. Martin-Lœf's type theory (ITT) as an intuitionistic approch works by defining types in a way that *objects* definition proceeds simultaneously with the *operations* definition. Martin-Lœf has defined some well-known data-types (list and natural numbers). He also defined the type of well-ordered sets and showed how to derive Natural numbers from this type. Nevertheless, it seemed to him (cf. [Martin-Lœf 79]) that this well-ordered type was not much related to computational aspects, since its main purpose was the definition of transfinite induction, which apparently had nothing to do with computation. In this work we show how to derive some well-known data-types, namely trees, lists and stacks, from the already mentioned well-ordered type.

In section 2 we briefly present the type theory used in this work. In section 3 we define the type of trees. In section 4 we show how to see the list-type defined by Martin-Lœf as an instance of the well-ordered type and point out an interesting feature of ITT when we get stacks as an instance of it too. Finally we conclude this work by asking a question : Is the well-ordered type enough to specify all the inductively defined data-types ?

1 PUC-Rio and GMD-Karlsruhe

2. ITT : an overview

The main idea of ITT is to formalize constructive reasoning. This is achieved by writing a kind of calculus and giving some possible interpretations to it, but not necessarily « semantics ». The part concerned with the calculus is built from some atomic elements called propositions. The other syntatic elements are combinations of propositions built up from operators or contructors. Thus the interpretations are given by interpreting the atomic elements and the several ways of combining them with the use of such constructors. Martin-Lœf calls these interpretations « judgments ». They will be succintly described below.

The basic propositions are « A set » and « $a \in A$ » and their different judgements are shown below :

A set	$A \in A$	obs
A is a set	a is an element of A	$A \neq \emptyset$
A is a proposition	a is a proof of A	A is true
A is an intention	a is a method of realizing A	A is realiz.
A is a problem	a is a method of solving A	A is solvable

The first is the set theoretical judgement, the second is the logical one. The third is due to Heyting [Heyting 31]. The fourth is due to Kolmogorov [Kolmogorov 32]. Briefly, we can say that Kolmogorov's meaning of problem is « something to do » (a task) and its solution is « how to do ». Kolmogorov called programs such solutions.

Following the constructive approach, Martin-Lœf believes that there are no universes a priori, i.e., the universes must be built. This approach had begun before with Brouwer [Brouwer]. So the objects of constructivism are all constantly being constructed, that is, they have a kind of potential existence. Thus, we can understand a set as a kind of procedure to generate its elements. This is a natural way to regard the set-theoretical form of judgement. Another thing that we must ensure is the equality of elements. We must indicate how to construct equal elements. In the logical interpretation this approach means that a proposition is true when we say what we can take into account as being a proof of it, and when two proofs are equal proofs of the same proposition.

In order to implement the notions above, Martin-Lœf uses the concept of canonical element, or canonical proof. So the normal form theorem underlies the concept above. Thus, the formation rules of a given set must not only include the way the canonical elements of the set are generated but also the rules concerning equality among canonical elements. So an element (in general) of a set is a method to yield canonical elements.

All sets (types) of predicative intuitionistic type theory are built from already constructed sets, i.e., from sets which have already their procedures

of generation. These procedures must be well defined rules of formation, i.e., there are some primitive sets. The primitive sets do not need other sets for their construction, which is done using well defined constructors determined by the formation rules. They are the finite sets and the natural numbers set. In order to produce higher types we use the following constructors :

- The Cartesian product (Π),
- The disjoint union or sum of a family set (Σ),
- The disjoint union of two sets (+),
- Propositional equality (I),
- Lists of elements of a set (List),
- Wellorderings (W).

Nevertheless, as Martin-Lœf derives the natural numbers from the well-orderings type (using the finite sets), we can say that the finite sets are indeed the primitive types of ITT. It is easy to see that one can only use these constructors a finite number of times to construct higher types from the primitive types (sets). So we have only a finite type theory.

In order to add transfinite types, Martin-Lœf introduced universes in his theory. The main idea of the universes is to describe the least set closed under certain specified set-forming operations. This is made in this way because intuitionistically we cannot have the set of all sets, since we cannot exihibit, once and for all, all possible set forming operations. But the concept of universes is enough to allow us to construct set of sets. Martin-Lœf gives two ways to the defining universes, a Tarskian and a Russellian. For more details about universes see [Martin Lœf 84].

In the sequel we present briefly the types used in this work giving more emphasis to the type of *Well-orderings*.

A very important concept in Martin-Lœf is that of hypothetical judgement. That is, a judgement based on a priori established judgements. We restrict ourselves to the case of one hypothesis only. The general case is an immediate consequence. A well-known concept in set theory related to hypothetical judgements is that of an indexed families of sets. To which we contruct a family A (i) based on another set, the set of indexes. From a constructive point of view, the building of an indexed family is a process that assumes a previously built set, or better, assumes the existence of its building process. This is undoubtly a hypothetical reasoning. Indeed, all ways in ITT of hypothetical reasoning are

represented by the above mentioned. This is a consequence of the fact that we can interpret the family concept just as a proof that uses hypothesis (this is done in the logical interpretation of ITT). There are some general facts about judgements :

- The instantiation of any indexed family of sets by a set of the index type is a set.
- Equal elements produce equal instantiations

These rules are called substitution rules.

$$\frac{\begin{array}{cc} & [x \in A] \\ & \vdots \\ a \in A & B\,(x)\ Set \end{array}}{B\,(a)\ Set}$$

$$\frac{\begin{array}{cc} & [x \in A] \\ & \vdots \\ a = c \in A & B\,(x)\ Set \end{array}}{B\,(a) = B\,(c)}$$

where B (x) [x $\in$ A] says that B is a family indexed by A. There are more substitutions rules in ITT, but we do not need them in this work. Each type in ITT is specified by giving a set of rules. These rules provide a kind of « meaning as use » semantics for the types. They fall down into four kinds :

1. Formation rules, which tell us the conditions needed to construct a new type. The conditions are in general information saying that such and such constituent object must be of such and such type.
2. Introduction rules, which tell us how the canonical solutions of the introduced type problem look like.
3. Elimination rules, which tell us how to retrieve information about any of the constituent types from the new type information. The information, according to our definition of problem, refers to canonical solutions.
4. Equality rules, which tell us when two canonical solutions of the new type are equal. These rules generally include either implicitly or explicitly the information about when two canonical solutions of each of the constituent types are equal.

Note that this structured way of construction is very close to the Natural Deduction way of modeling mathematical reasoning, which is not always constructive.

The disjoint union $(A + B)$ of two sets A and B can be viewed as a set which has as elements the elements of A and of B. However, together with each element of $A + B$ we have information about to each of A or B this element belongs. Martin-Lœf uses the letters i and j to denote this information. Thus, we have the following rules for the disjoint union.

+ –Formation

$$\frac{A \ Set \quad B \ Set}{A + B \ Set}$$

+ –Introduction

$$\frac{a \in A}{i(a) \in A + B} \qquad \frac{b \in B}{j(b) \in A + B}$$

+ –Elimination

$$\frac{c \in A + B \qquad \begin{array}{c}[x \in A] \\ | \\ d(x) \in C(i(x))\end{array} \qquad \begin{array}{c}[y \in B] \\ | \\ e(y) \in C(j(y))\end{array}}{D(c, (x)\, d(x), (y)\, e(y)) \in C(c)}$$

+ –Equality

$$\frac{a \in A \qquad \begin{array}{c}[x \in A] \\ | \\ d(x) \in C(i(x))\end{array} \qquad \begin{array}{c}[y \in B] \\ | \\ e(y) \in C(j(y))\end{array}}{D(i(a), (x)\, d(x), (y)\, e(y)) = d(a) \in C(c)}$$

$$\frac{b \in A \qquad \begin{array}{c}[x \in A] \\ | \\ d(x) \in C(i(x))\end{array} \qquad \begin{array}{c}[y \in B] \\ | \\ e(y) \in C(j(y))\end{array}}{D(j(b), (x)\, d(x), (y)\, e(y)) = e(b) \in C(c)}$$

It is interesting to note that the equality rules explain the computational meaning of D.

Another type that we use (in this work) is the Cartesian product of a family of sets. In a certain way we can say that this type formalizes the notions of hypothetical judgement and substitution.

Π–Formation

$$\frac{\begin{array}{c}[x \in A]\\ \vdots \\ A\ Set \qquad B\ (x)\ Set\end{array}}{(\Pi\ x \in A)\ B\ (x)\ \ Set}$$

Π–Introduction

$$\frac{\begin{array}{c}[x \in A]\\ \vdots \\ b\ (x) \in (\Pi\ x \in A)\ B\ (x)\end{array}}{\lambda\ x\ b\ (x) \in B\ (x)}$$

Π–Elimination

$$\frac{c \in (\Pi\ x \in A)\ B\ (x)\ \ a \in A}{Ap\ (c, a) \in B\ (a)}$$

Π–Equality

$$\frac{\begin{array}{c}[x \in A]\\ \vdots \\ a \in A \qquad b\ (x) \in (\Pi\ x \in A)\ B\ (x)\end{array}}{Ap\ (\lambda\ xb\ (x), a) = b\ (a) \in B\ (a)}$$

The primitive sets in ITT are just the finite sets, which are given outright; hence their set formation will have no premisses. For each natural number n (in the metalanguage) we have the following rules :

N_n–Formation and N_n–Introduction

$$N_n\ Set \qquad\qquad m_n \in N_n\ (m = 0, \ldots, n\text{-}1)$$

N_n–Elimination

$$\frac{c \in N_n\ \ c_m \in C\ (m_n)\ (m = 0, \ldots, n\text{-}1)}{R_n\ (c, c_0, \ldots, c_{n\text{-}1}) \in C\ (c)}$$

Where the computational meaning of R_n is given by the following R_n–Equality rule.

$$\frac{c_m \in C\ (m_n) \qquad (m = 0, \ldots, n\text{-}1)}{R_n\ (m_n, c_0, \ldots, c_{n\text{-}1}) = c_m \in C\ (m_n)}$$

The set N_0 is the empty set, for it does not have any introduction rule.

2.1 The well-ordered type

We now present the main type in this work, namely the Well-ordering type. This type formalizes two concepts :

- The notion of predecessor (or successor).
- The existence of an initial element for each instance of the type.

A well-known example of a well-ordered constructive set is the set of natural numbers, where *0* is the minimal element and successor is a primitive notion. However, in the case of natural numbers each element has only one predecessor. We know that this is not the case in general. There may be well-ordered sets with elements with infinitely many predecessors. An example is the set of all subsets of the natural numbers set with inclusion as the order relation. Thus, the set N (of the natural numbers) has $N - \{n\}$ as predecessor for each n. Note that there is no problem with the axiom of choice in the example above, since it is proved in ITT.

A well-ordered set may constructively be given by saying how we obtain the predecessors of any element. Thus any element of a well-ordered set should contain all information needed to get each of its predecessors, that is :

- How many predecessors does it have ?
- « What » are they ?

Martin-Lœf structured the necessary information in the following way :

1. The quantity of elements is given, in a rather abstract way, by a set. For example an element that has no predecessors will have the empty set N_0 associated to it.
2. The predecessors are given by a function from the already mentioned set into the well-ordered set.

Since each element of the well-ordered set has a different number of predecessors, the information about the set mentioned in the first item is given by an index. Note that in this way there is no special treatment of the initial elements besides associating to them the index of the empty set. Thus, we need a family of sets to build any well-ordered set. This is the meaning of the rule below.

W-Formation

$$\frac{\begin{array}{cc}[x \in A] & \\ | & \\ A\ Set & B\ (x)\ Set\end{array}}{(W\, x \in A)\, B\ (x)\ Set}$$

Following the remarks above an element of a well-ordered set is of the form $Sup\ (a, f)$ where a is the index of the set that provides the numbers of predecessors and f is the function that generates them. Thus we have the following :

W-Introduction

$$\frac{a \in A \qquad b \in B\ (a) \rightarrow (Wx \in A)\, B\ (x)}{Sup\ (a, b) \in (Wx \in A)\, B\ (x)}$$

The W-Elimination rule represents the structure of proofs by induction on the well-ordered sets.

W-Elimination

$$\frac{\begin{array}{cc}c \in (Wx \in A)\, B\ (x) & [x \in A, y \in B\ (x) \rightarrow (Wx \in A)\, B\ (x), z \in (\Pi v \in B\ (x))\, C\ (Ap\ (y, v))] \\ & | \\ & d\ (x, y, z) \in C\ (Sup\ (x, y))\end{array}}{T\ (c, (x, y, z)\, d\ (x, y, z)) \in C\ (c)}$$

In the rule above, z may be viewed as a proof that the C holds for all predecessors of $Sup\ (x, y)$ and $d\ (x, y, z)$ is the proof that C holds for $Sup\ (x, y)$ under this assumption. Thus, in these conditions $T\ (c, (x, y, z)\ d\ (x, y, z))$ is a proof that C holds for c arbitrary. The way of computing T is given by the W-Equality rule below, that is just the recursion on well-ordered sets.

W-Equality

$$\frac{\begin{array}{cc}a \in A\, b \in B\ (a) \rightarrow (Wx \in A)\, B\ (x) & [x \in A, y \in B\ (x) \rightarrow (Wx \in A)\, B\ (x), z \in (\Pi v \in B\ (x))\, C\ (Ap\ (y, v))] \\ & | \\ & d\ (x, y, z) \in C\ (Sup\ (x, y))\end{array}}{T\ (Sup\ (a, b), (x, y, z)\, d\ (x, y, z)) = d\ (a, b, \lambda v T\ (Ap\ (b, v), (x, y, z)\, d\ (x, y, z))) \in C\ (Sup\ (a, b))}$$

Martin-Lœf derived the natural numbers from the W-type by setting :

1. $A = N_2$

2. $B\ (0_2) = N_0$

3. $B\ (1_2) = N_1$

Thus our relationships are :

- $0 = \text{Sup}\ (0_2, \lambda\ xR_0\ (x))$ that is the initial element.
- $a' = \text{Sup}\ (1_2, \lambda\ xa)$ is the successor of a.

Thus, we can see the set (type) of natural numbers of *ITT* (as presented in [Martin-Lœf 84] as $(Wx \in N_2)\ B\ (x)$.

Another example is the use of the *W*-type to define simultaneous recursion on a pair of natural numbers. To obtain this simultaneous recursion we need to build up a well-ordered set that has (as the initial elements) the pairs (x, y) with either x or y equal to zero. Besides we need to have as only predecessor of (x', y') the pair (x, y). So our set has an infinite number of initial elements and each element has only one predecessor. We call this set $2\ \omega$.

Below, we give the $2\ \omega$–Introduction rules as instances of *W*–Introduction by doing $A = N \times N$, $B\ ((0, x)) = B\ ((y, 0)) = N_0$ and $B\ ((x', y')) = N_1$.

$$\frac{x \in N}{Sup\ ((x, 0), \lambda vR_0\ (v))}$$

$$\frac{y \in N}{Sup\ ((0, y), \lambda vR_0\ (v))}$$

$$\frac{Sup\ ((x, y), f) \in 2\ \omega}{Sup\ ((x', y'), \lambda vSup\ ((x, y), f))}$$

Next we show the $2\ \omega$–Equality rule which will explain the simultaneous resursion mechanism. For the two initial cases, we only have :

$$d \in C\ (sup\ ((0, y), \lambda yR\ (y))) \qquad d \in C\ (sup\ ((x, 0), \lambda yR\ (y)))$$

For the other case, we do the following in order to get the instance of the *W*–Equality :

1. We take the assumption $a \in A$ as $(x', y') \in N \times N$.
2. Instead of the premiss about the predecessor function, we take the element $sup\ ((x, y), f) \in 2\ \omega$.

3. Since $B((x', y'))$ has only one element (1_1), we replace $z \in (\Pi v \in B((x', y')))$ $C(Ap(\lambda y sup((x, y), f), v)$ by $Ap(z, 1_1) \in C(sup((x, y), f))$.

4. Finally we replace $a \in A$ and $b \in B(a) \to 2\ \omega$ by its $2\ \omega$–Introduction conclusion.

Thus :

$$\frac{\begin{array}{c}[(x', y') \in N \times N, Sup((x, y), f) \in 2\ \omega, Ap(z, 1_1) \in C(Sup((x, y), f))] \\ | \\ Sup((a', b'), \lambda y Sup((a, b), g)) \qquad d(x, y, f, Ap(z, 1_1)) \in C(Sup((x', y'), \lambda v Sup((x, y), f)))\end{array}}{T(Sup((a', b'), \lambda v Sup((a, b), f)), (x, y, f, z) d(x, y, f, Ap(z, 1_1))) = d(a', b', g, T(Sup((a, b), g), (x, y, f, z) d(x, y, f, Ap(z, 1_1))))}$$

We note that the rules depend mainly on x, y and z. Thus, we can relate directly $N \times N$ to $2\ \omega$, simplifying the rule above such that :

$$\frac{\begin{array}{c}[(x, y) \in N \times N, z \in C(x, y)] \\ | \\ (a', b') \in N \times N \qquad d(x, y, z) \in C((x', y'))\end{array}}{T((a', b'), (x, y, z)\ d(x, y, z)) = d(a', b', T((a, b), (x, y, z)\ d(x, y, z)))}$$

where we get (x', y') e N x N from (x, y) e N x N.

Thus doing the same for the initial cases and writing all cases in a single rule we have the desired simultaneous recursion rule :

$$\frac{\begin{array}{c}[(x, y) \in N \times N, z \in C(x, y)] \\ | \\ (a, b) \in N \times N\ d_{0y} \in C(0, y)\ d_{x0} \in C(x, 0) \qquad d(x, y, z) \in C(x', y')\end{array}}{R2\ \omega((a, b), d_{0y}, d_{x0}, (x, y, z)\ d(x, y, z)) \in C(a, b)}$$

The equality rule says that :

1. If $a = 0$, then we have $d \in C(0, b)$.

2. If $b = 0$, then we have $d \in C(a, 0)$.

3. Else we have $a = i'$ and $b = j'$ and $d(i', j', R2w(i, j, (x, y, z)\ d(x, y, z))) \in C(a, b)$.

3. The $Tree_n(D)$ type

In this section we define the type of the trees of maximum order equals n ($Tree_n(D)$). To get trees from the *W–type* we need to see them as members

of a well-ordered set (type). However, as trees themselves have a natural ordering, we should specify this in the *W–type*. Using this approach, we see that the initial element for all trees is the empty one. It seems clear that a tree which has only one node is the natural successor of the empty tree, for we can see it as a tree which has this node and all of its n subtrees are empty. Naturally, the successor of any set $(b_1, \ldots, b_n)$ of trees is the tree which has a node a as root and its subtrees are $b_1, \ldots, b_n$.

From the point of view of the *W–type*, the empty tree must be associated to the initial element. We can do this as it was done for the natural numbers in [Martin Lœf 84]. However, here we have to take into account the labels of the nodes that are elements of D. As the empty tree does not have any label, and we must label all the nodes of a nonempty tree, we cannot associate any member of D to the empty set (N_0) in order to describe the initial element. So we add a new element creating a new set for set of indexes. We use the disjoint sum of two sets to obtain the desired set. Besides, as any nonempty tree has at most n predecessors, then the family of indexed sets is as follows for $D + N_1$ as set of indexes :

- $B\,(i\,(d)) = N_n$
- $B\,(j\,(0_1)) = N_0$

Thus, doing :

- $nil\text{-}tree = Sup\,(j\,(0_1), \lambda\, xR_0\,(x))$
- $a = Sup\,(i\,(a), \lambda\, xR_n\,(x, nil\text{-}tree, \ldots, nil\text{-}tree))$
- $b_1 \overset{a}{\wedge} b_n = Sup\,(i\,(a), \lambda\, xR_n\,(x, b_1, \ldots, b_n))$ (with $\Delta \ldots \Delta$ below)

We have the following rules as Tree–Introduction :

- $nil\text{-}tree \in Tree_n\,(D)$
- $$\frac{a \in D \qquad b_1 \in Tree_n\,(D) \ldots b_n \in Tree_n\,(D)}{b_1 \overset{a}{\wedge} b_n \in Tree_n\,(D)}$$ (with $\Delta \ldots \Delta$ below)

The former we can justify as following :

$$\frac{\begin{array}{cc} & [x \in N_0] \\ 0_1 \in N_1 & \vdots \\ \vdots & R_0(x) \in (Wx \in (D + N_1))\ B(x) \\ j(0_1) \in D + N_1 & \lambda x R_0(x) \in N_0 \to (Wx \in (D + N_1)) \end{array}}{Sup(j(0_1), \lambda x R_0(x)) \in (Wx \in (D + N_1))\ B(x)}$$

And we justify the last with the following derivation :

$$\frac{\begin{array}{cc} & [x \in N_n],\ b_i \in (Wx \in (D + N_1))\ B(x) \quad (i = 1, \ldots, n) \\ a \in D & \vdots \\ \vdots & R_n(x, b_1, \ldots, b_n) \in (Wx \in (D + N_1))\ B(x) \\ i(a) \in D + N_1 & \lambda x R_n(x, b_1, \ldots, b_n) \in N_n \to (Wx \in (D + N_1))\ B(x)) \end{array}}{Sup(i(a), \lambda x R_n(x, b_1, \ldots, b_n)) \in (Wx \in (D + N_1))\ B(x)}$$

However, the rules above only show how to construct trees of order n. Nothing was said about how to deal with trees. In a general view we need rules that allow us to prove properties about trees. A computer scientist proves properties of trees by structural induction. Thus, we need induction rules for trees. Here, we will not content ourselves with the presentation of the induction rules as elimination rules. We include the equality rules as being part of the explanation of « what a structural induction on trees is ».

To proof some general property about trees we must prove it for the empty tree (nil-tree). Furthermore we must prove that the property hold for a tree, assuming that it holds for all of its subtrees, indeed only the greatest proper subtrees need to be taken into account. This kind of proof is formalized by the following rule :

Tree–Elimination

$$\frac{\begin{array}{cc} & [x \in D, y_i \in Tree_n(D), z_i \in C(y_i)] \\ & \vdots \\ & e(x, \vec{y}, \vec{z}) \in C(y_1 \overset{x}{\wedge} y_n) \\ t \in Tree_n(D)\ d \in C(nil\text{-}tree) & \Delta \ldots \Delta \end{array}}{Treerec(t, d, (x, \vec{y}, \vec{z})\ e(x, \vec{y}, \vec{z})) \in C(t)}$$

where $\vec{y} = (y_1, \ldots, y_n)$ and $\vec{z} = (z_1, \ldots, z_n)$. Note $d \in C$ (*nil-tree*) should be read as d is a proof that the property C holds for the empty tree. $e(x, \vec{y}, \vec{z})$ should be read as the proof that C holds for a tree with root x and subtrees y_i, from the assumptions that z_i are proofs that C holds for each respective subtree y_i. The last is just our induction hypothesis. Well, the conclusion should be read as *Treerec* $(t, d, (x, \vec{y}, \vec{z})\ e(x, \vec{y}, \vec{z}))$ is a proof that C holds for any tree t. But,

from the intuitionistic point of view a proof is a computable object and we must say how to compute *Treerec*. Intuitively, we know that *Treerec* represents the recursion over trees. But, before we justify the rule above with the *W–introduction* rule.

Since there is no predecessors for the empty tree, the *W–elimination* as well as the *W–equality* for it is only $d \in C\ (nil)$ and :

$$\frac{d \in C\ (nil\text{-}tree)}{T\ (nil\text{-}tree, d) = d \in C\ (nil\text{-}tree)}$$

respectively.

In the general case we have that :

- $$\frac{[x \in N_n] \quad b_i \in (Wx \in (D + N_1))\ B\ (x) \quad (i = 1, \ldots, n)}{\dfrac{R_n\ (x, b_1, \ldots, b_n) \in (Wx \in (D + N_1))\ B\ (x)}{\lambda\ xR_n\ (x, b_1, \ldots, b_n) \in N_n \rightarrow (Wx \in (D + N_1))\ B\ (x))}}$$

  justifies the assumption $y \in N_n \rightarrow (Wx \in (D + N_1))\ B\ (x)$ of the W–Elimination rule by taking y as being $\lambda\ xR_n\ (x, y_1, \ldots, y_n)$ so its discharge can be viewed as a discharge of all $y_i \in Tree_n\ (D)$.

- Using the derivations above, the assumptions of the Tree–Elimination rule, and some rules of Martin-Lœf type theory we can justify the assumption $\lambda\ vR_n\ (v, z_1, \ldots, z_n) \in (\Pi\ v \in N_n)\ C\ (Ap\ (y, v))$ need to use the *W*–Elimination.

$$\dfrac{[v \in N_n] \quad \dfrac{z_i \in C\ (y_i) \quad \dfrac{x \in Tree_n\ (D) \quad C\ (x)}{\dfrac{y_i \in Tree_n\ (D), y \in N_n \rightarrow Tree_n\ (D)}{\Pi}}}{Ap\ (y, m_i) = y_i \in Tree_n\ (D)}}{\dfrac{C\ (Ap\ (y, m_i)) = C\ (y_i)}{\dfrac{z_i \in C\ (Ap\ (y, m_i))}{\dfrac{R_n\ (v, z_1, \ldots, z_n) \in C\ (Ap\ (y, v))}{\lambda\ vR_n\ (v, z_1, \ldots, z_n) \in (\Pi\ v \in N_n)\ C\ (Ap\ (y, v))}}}}$$

where Π is :

$$\dfrac{\dfrac{m_i \in N_n \quad y \in N_n \rightarrow Tree_n\ (D)}{Ap\ (y, m_i) = R_n\ (m_i, y_1, \ldots, y_n) \in Tree_n\ (D)} \quad \dfrac{y_i \in Tree_n\ (D)}{R_n\ (m_i, y_1, \ldots, y_n) = y_i \in Tree_n\ (D)}}{Ap\ (y, m_i) = y_i \in Tree_n\ (D)}$$

It should be noted that in order to apply the R_n we need to make sure the premisses for each $m_i \in \{0, \ldots, n\text{-}1\}$ hold, but for the sake of space we let this implicit rather than explicit.

- In order to see the Tree–Elimination rule as an instance of the *W*–Elimination, we prove the assumption $i\,(x) \in (D + N_1)$ using $x \in D$, and taking :

  – $d\,(i\,(x), \lambda\, xR_n\,(x, y_1, \ldots, y_n), \lambda\, vR_n\,(v, z_1, \ldots, z_n))$ as $e\,(x, \vec{y}, \vec{z})$, and

  – $T\,(t, (x, \vec{y}, \vec{z})\, e\,(x, \vec{y}, \vec{z}))$ as $Treerec\,(t, d, (x, \vec{y}, \vec{z})\, e\,(x, \vec{y}, \vec{z})$.

  The last by putting the two cases of the W–Elimination (instantiated to Tree–Elimination) in the computational meaning of *Treerec*, adding one arity in order to get the basis step d.

In order to obtain the computational meaning of *Treerec*, we should look to the Tree–Equality rules which are instances (as above) of the *W*–Equality rules.

Tree–Equality

$$\frac{d \in C\,(nil\text{-}tree) \qquad \begin{array}{c}[x \in D,\ y_i \in Tree_n\,(D),\ z_i \in C\,(y_i)] \\ | \\ e\,(x, \vec{y}, \vec{z}) \in C\,(y_1 \overset{x}{\frown} y_n) \\ \Delta \ldots \Delta\end{array}}{Treerec\,(nil, d, (x, \vec{y}, \vec{z})\, e\,(x, \vec{y}, \vec{z})) = d \in C\,(nil\text{-}tree)}$$

$$\frac{a \in D \quad b_i \in Tree_n\,(D) \quad d \in C\,(nil\text{-}tree) \qquad \begin{array}{c}[x \in D,\ y_i \in Tree_n\,(D),\ z_i \in C\,(y_i)] \\ | \\ e\,(x, \vec{y}, \vec{z}) \in C\,(y_1 \overset{x}{\frown} y_2) \\ \Delta \ldots \Delta\end{array}}{\begin{array}{c}Treerec\,(b_1 \overset{a}{\frown} b_n, d, (x, \vec{y}, \vec{z})\, e\,(x, \vec{y}, \vec{z})) = \\ \Delta \ldots \Delta\end{array}}$$

$$e(a, b_1, \ldots, b_n, Treerec\,(b_1, d, (x, \vec{y}, \vec{z})\, e(x, \vec{y}, \vec{z}), \ldots, Treerec\,(b_n, d, (x, \vec{y}, \vec{z})\, e(x, \vec{y}, \vec{z}))$$
$$\in C\,(b_1 \overset{a}{\frown} b_n)$$
$$\Delta \ldots \Delta$$

4. The list-type

We get lists from trees by stating :

- $B\,(j\,(0_1)) = N_0$.

- $B\ (i\ (d)) = N_1$.
- List $(D) = (Wx \in (D + N_1))\ B\ (x)$.

This means that a list is, for us, an unary tree. However, we can derive the Martin-Lœf's rules for lists from ours for trees. Thus, for List–introduction, we have :

$$\frac{a \in D \quad b \in List\ (D)}{a.b \in List\ (D)}$$

where *List* (D) is taken as $Tree_1\ (D)$. Thus, from the point of view of the W–type $a.b = Sup\ (i\ (a), \lambda x b)$, reminding $R_1\ (x,\ b) = b$. Besides, we can take *nil-list* as being *nil-tree*, for *nil-tree* $\in Tree_n\ (D)$ for all n.

For the List–Elimination we have :

$$\frac{\begin{array}{cc} & [x \in D, y \in List\ (D), z \in C\ (y)] \\ & \vdots \\ l \in List\ (D)\ d \in C\ (nil) & e\ (x, y, z) \in C\ (x.y) \end{array}}{Listrec\ (l, d, (x, y, z)\ e\ (x, y, z)) \in C\ (l)}$$

where *Listrec* $(l, d, \ldots) = Treerec\ (l, d, \ldots)$. Finally, it is clear that we can see the List–equality as an instantiation of Tree–Equality and hence an instantiation of W–Equality.

$$\frac{\begin{array}{cc} & [x \in D, y \in List\ (D), z \in C\ (y)] \\ & \vdots \\ d \in C\ (nil\text{-}list) & e\ (x, y, z) \in C\ (x.y) \end{array}}{Listrec\ (l, d, (x, y, z)\ e\ (x, y, z)) = d \in C\ (nil\text{-}list)}$$

It is interesting to note that the presence of the induction step in the rule is inherited from the W–equality rule.

$$\frac{\begin{array}{cc} & [x \in D, y \in List\ (D), z \in C\ (y)] \\ & \vdots \\ a \in D \quad b \in List\ (D) \quad d \in C\ (nil\text{-}list) & e\ (x, y, z) \in C\ (x.y) \end{array}}{Listrec\ (a.b, d, (x, y, z)\ e\ (x, y, z)) = e\ (a, b, Listrec\ (b, d, (x, y, z)\ e\ (x, y, z))) \in C\ (a.b)}$$

4.1. Stacks

It is usual to specify data-types algebraically. This kind of specification consists in declaring a set of operations in a rather axiomatic way. Thus, for example we may have the following specification of the data-type *Stack* of any basic type *Btype*. (adapted from [Guttag 77]).

- $\begin{array}{rcl} \textit{Newstack}: & \rightarrow & \textit{Stack} \\ \textit{Push}: \textit{Stack} \times \textit{Btype} & \rightarrow & \textit{Stack} \\ \textit{Top}: \textit{Stack} & \rightarrow & \textit{Btype} \\ \textit{Pop} \; \textit{Stack} & \rightarrow & \textit{Stack} \\ \textit{IsNewStack ?} \; \textit{Stack} & \rightarrow & \textit{Boolean} \end{array}$

- – *IsNewStack* ? (*Newstack*) = *true*
  – *IsNewStack* ? (*Push* (*s*, *e*)) = *false*
  – *Pop* (*Push* (*s*, *e*)) = *s*
  – *Top* (*Push* (*s*, *e*)) = *e*
  – *Pop* (*Newstack*) = *error*
  – *Top* (*Newstack*) = *error*

Note that this kind of specification usually uses a distinguished element (*error*) in order to not be partial. However, this may not be a good idea since we should declare that the result of any sequence of operation will be « error » whenever it is an intermediate result. For example *Pop* (*Pop* (*Newstack*)) = *error*. This is a way to supply continuity to the operations. Nevertheless the introduction of error may be seen as (a little) artificial. Another solution may simply state that *Pop* (*Newstack*) is a kind of miswritten construction, that is the problem s reduced to a syntactic one. As *ITT* can be viewed as having its semantics based on the « meaning as use » approach we will see in the sequel that the last is the most natural way of doing it.

The type *Stack* may be specified in *ITT* as following :

- *Push* (*a*, *b*) = a.*b*

- *Top* (*l*) = *Listrec* (*l*, *nil-list*, (*x*, *y*, *z*) *x*)

- *Pop* (*l*) = *Listrec* (*l*, *nil-list*, (*x*, *y*, *z*) *y*)

- *IsNewStack* ? (*l*) = *Listrec* (*l*, 0_2, (*x*, *y*, *z*) 1_2)

where should be noticed that we related *nil-list* to the empty *stack* which is specified by the operation *Newstack*. However, one can note that the function *Pop* was defined in a way that there is no undefiniteness. The reason is that we can only express the totally undefined function, that is $\lambda \; xR_0 \; (x) \in (\Pi \; x \in N_0) \; B \; (x)$ for any type $B \; (x) \; (x \in N_0)$. However, we cannot execute this function since it is a function with empty range. Thus, all the functions specified by the type theory are total. We should note that the only way to express the undefiniteness is with R_0.

Nevertheless, the solution above cannot be used with regard to *Top*, i.e., *Top* (*l*) = *Listrec* (*l*, *nil-list*, (*x*, *y*, *z*) *x*), for this term cannot be derived from

List–Elimination since *nil-list* is not of the type *Btype*. Besides, an element of *Btype* cannot be used without losing the intended meaning of *Top*. Thus, we may include the *error*. But, as it has already been pointed out, this would be too artificial if we are not treating of exceptions. Note that this could be the case too, as for example in a compiler specification.

We may also assume that exceptions treatment should not be included in a data type specification, i.e., the partiality is in a certain way desired. Thus, we can take into account that *Top* (*Newstack*) only has no meaning. So, we do not need to give a meaning to it (*error*). In this case we will see that *ITT* provides tools enough for this task in a natural way. The below derivation may be of some help.

$$\dfrac{\dfrac{\dfrac{\dfrac{[x \in D] \quad [l \in List\ (D)]}{x.l \in List\ (D)} \qquad [x \in D] \quad [x \in D]}{Listrec\ (x.l, x, (x, y, z)\ x) \in D}}{\lambda\, x Listrec\ (x.l, x, (x, y, z) \in D \to D}}{\lambda\, l\, \lambda\, x Listrec\ (x.l, x, (x, y, z) \in D \to (D \to D)}$$

And we can define $Top\ (a.b) = Ap\ (Ap\ (\lambda\, l\, \lambda\, x Listrec\ (x.l, x, (x, y, z)), b), a)$. But, this is the same that $Top\ (a.b) = b$. Anyway we have defined it in such a way that *Top* (*nil-list*) has no meaning. However, we should note that we implicitly related *List* (*D*) to $D \times List\ (D)$ for all the non-nil elements. Nevertheless this can be justified in the conclusion if we note that $A \otimes B \to C$ may be viewed as $A \to (B \to C)$. We remind the reader that this is just the logical intuitionistic relationship between $A \wedge B \to C$ and $A \to (B \to C)$ provable in *ITT*.

In order to prove that this specification is really a *stack* we should prove the equations of the algebraic specification. $Top\ (Push\ (e, s)) = e \in D$ is one of them (expressed in *ITT*). Nevertheless, because of our definition we do not need any inductive proof. It is enough to note that $Top\ (Push\ (e, s)) = Top\ (e.s) \in D$ and $Top\ (e.s) = e \in D$ under the assumptions $e \in D$ and $s \in List\ (D)$. The other equation is $Pop\ (Push\ (e, s)) = s$. Using the List–Equality rule we can obtain the result desired.

$$\dfrac{e \in D \quad s \in List\ (D) \quad nil\text{-}list \in List\ (D) \quad [y \in List\ (D)]}{Listrec\ (e.s, nil\text{-}list, (x, y, z)\ y) = s \in List\ (D)}$$

Using the definition of *Push* and a substitution rule we get that $Listrec\ (Push\ (e, s), nil\text{-}list, (x, y, z)\ y) = s \in List\ (D)$. Finally, by the definition of *Pop* we reach the conclusion.

5. Conclusion

In this work we have shown how to get some inductive types from the *W*–type. However, based on the stacks example we can note that *ITT* gives

us more than the list type when we derived it from the *W*-type. Since we defined the operations of the *Stack* type using a general recursion principle on lists, we could say that the rules about *Lists* have already power enough to do the specification of the operations refered. In other words, when we define a type in *ITT* we define how the *objects* would be used rather than the type itself. What is diferent in *ITT* is that *objects* definition is done using construction operations (introduction rules) as well as analysis operations (elimination rules) together with reduction relationships (equality rules). Thus, it may be the case that the operations coincide with the data type operations. We may even consider what does this coincidence mean. Anyway we cannot expect that in the case of an algebraic specification this should be the right thing.

Thus, in order not to get into philosophical questions about the meaning in computation we use the concept of *data structures* as being related to the *objects* in a *data type*. Thus we can say that the *W-type* was used to construct the *objects* used in a data type specification. For example, the type *List* may be used to specify *Stacks* or the data type *List* (note the difference) or even other data type that uses objects with the same inductive principle of construction.

We remark that the advantage of using *ITT* to specify types is that we have rules of construction that may be used as verification rules as well. And all the properties about data type are proved from the *objects* construction level. Finally, we would like to conjecture that all inductively defined data structures can be defined using *W*-types as the main type in the specification. This work was enough to raise the question. However, a more general treatment is needed to answer it.

6. Acknowledgements

One of us (E. H. H.) would like to thank GMD, for the nice stay, as well as for the financial support, and particularly Hendrik C. Lock, who was responsable for the invitation. We also thank Valéria Paiva for the useful corrections suggested after the reading of the first draft.

References

[Brouwer] *Cambridge Lectures on Intuitionism.*

[Guttag 77] Guttag, John. "Abstract Data Types and the Development of Data Structures" Communications of ACM, Vol. 20, Number 6, June 1977.

[Heyting 31] Heyting, A. "Die Intuitionische Grundlegung der Mathematik", Erkenntnis, Vol. 2, 1931.

[Kolmogorov 32] Kolmogorov, A.N. "Zur Deutung der Intuitionischen Logik", Mathematische Zeitschrift, Vol. 35, 1932.

[Martin-Lœf 84] Martin-Lœf, Per. *Intuitionistic TypeTheory*, Bibliopolis, Edizioni di Filosofia e Scienza, 1984.

[Martin-Lœf 79] Martin-Lœf, "Constructive Mathematics and Computer Programming", paper read at the 6-th International Congress for Logic, Methodology and Phil. of Science. Hannover, Aug 1979, pp. 11.