

An induction principle for nested datatypes in intensional type theory

RALPH MATTHES*

I.R.I.T., C.N.R.S. & Université Paul Sabatier
118 route de Narbonne, F-31062 Toulouse Cedex 9, France

Abstract

Nested datatypes are families of datatypes that are indexed over all types such that the constructors may relate different family members (unlike the homogeneous lists). Moreover, the argument types of the constructors refer to indices given by expressions where the family name may occur. Especially in this case of true nesting, termination of functions that traverse these data structures is far from being obvious.

A joint article with A. Abel and T. Uustalu (TCS 333(1-2), pp. 3-66, 2005) proposes iteration schemes that guarantee termination not by structural requirements but just by polymorphic typing. They are generic in the sense that no specific syntactic form of the underlying datatype “functor” is required. However, there was no induction principle for the verification of the programs thus obtained although they are well-known in the usual model of initial algebras on endofunctor categories.

The new contribution is a representation of nested datatypes in intensional type theory (more specifically, in the Calculus of Inductive Constructions) that is still generic and covers true nesting, guarantees termination of all expressible programs and has an induction principle that allows to prove functoriality of monotonicity witnesses (maps for nested datatypes) and naturality properties of iteratively defined polymorphic functions.

1 Introduction

The algebra of programming (Bird & de Moor, 1997) shows the benefits of programming recursive functions in a structured fashion, in particular with iterators: there are equational laws that allow a calculational way of verification. Also for nested datatypes (Bird & Meertens, 1998), already intuitively introduced in the abstract, laws have been important from the beginning.

In previous work, the author concentrated on polymorphic lambda-calculi with nested datatypes that guarantee termination of all functions that follow the proposed iteration schemes. See, in particular, the comprehensive article with Abel and Uustalu (Abel *et al.*, 2005). Laws were not given, but the schemes were more general than previous work (Bird & Paterson, 1999a; Hinze, 2000; Martin *et al.*, 2004) in that they impose minimal conditions on the datatype “functor” F of rank 2 (F is a function that takes type transformations to type transformations) whose least fixed point μF is still a type transformation and

* with financial support by the European Union FP6-2002-IST-C Coordination Action 510996 “Types for Proofs and Programs”

not just a type. There is no need to require continuity properties or that F belongs to some given set of higher-order functors that is generated from some closure properties.¹ Since the ambient calculus is not a category, and the “functors” are no functors since no functoriality laws are required, the laws of program transformation and verification were not considered. The present article proposes a combination of two worlds: the world of terminating programs known from type theory and the categorical laws used in advanced functional programming.

The advantages of Mendler’s style (Mendler, 1987) are once more demonstrated:² the approach is very flexible since no syntactic criterion on the form of recursive calls is applied for termination checking. It is *type-based termination*: the types of the recursive calls ensure that there is no infinite reduction sequence starting with a well-typed term. For a discussion and the example of the *map* function for lists, see Section 3.1 of our previously cited article (Abel *et al.*, 2005). However, there has not been any contribution on the verification of programs in Mendler’s style for nested datatypes. For truly nested datatypes (for a definition, see page 8), this is even more important since there, termination is very unintuitive. On the other hand, plain heterogeneous families can be used well in the conventional style of iteration that directly follows the concept of initial algebras. This conventional style is available in the Calculus of Inductive Constructions (Coquand & Paulin, 1990) on which the theorem proving environment Coq is based, but also in other systems.

We want to do verification in the same system in which we write our programs, and we want a termination guarantee. Moreover, we insist on decidable type-checking. Thus, as our ambient calculus, we have chosen the Calculus of Inductive Constructions (CIC), in the current form in which it is implemented in the Coq proof assistant (Coq Development Team, 2006). We will only need concepts and features of Coq that are explained in the Coq textbook (Bertot & Castéran, 2004).

It will turn out that, after having introduced non-canonical elements into Mendler’s style – following Uustalu & Vene (2002) – the CIC supports reasoning on inductive *types* in Mendler’s style very well. A naive lifting of this approach to nested datatypes can also be expressed in the CIC. Unfortunately, this does not give enough reasoning power since one programs polymorphic functions on nested datatypes for which naturality laws are needed if more serious verification is aimed at. In order to enrich Mendler’s style beyond the plain lifting to families of datatypes, one has to leave the realm of inductive families towards simultaneous inductive-recursive definitions as proposed by Dybjer since 1991, available in final journal version (Dybjer, 2000), while Dybjer and Setzer found a finite axiomatization (Dybjer & Setzer, 2003). The single inductive-recursive definition we will use will not directly be an instance of these proposals due to two reasons:

- We make use of impredicativity in our system (not in the formulation, but its justification requires impredicativity), and those systems are predicative.

¹ This is not to say that all of the operational behaviours of the cited proposals were covered, see section 9 of our previous work (Abel *et al.*, 2005) that explains in which way *efolds* (Martin *et al.*, 2004) are indeed reconstructed and how *gfolds* (Bird & Paterson, 1999a) resist this effort.

² For inductive *types*, i. e., not inductive families, this is developed with many examples in Uustalu’s PhD thesis (1998).

- The map function $map_{\mu F}$ for the inductive family μF that we define simultaneously with the inductive generation of μF involves the inductive family not only in the source type constructor but also in the target type constructor, and that is excluded right from the outset in those systems.

The first problem is overcome by Capretta’s unpublished note (Capretta, 2004) that aims at a justification of simultaneous inductive-recursive definitions in impredicative CIC. The second problem, however, is not dealt with in that note. The idea for our construction is nevertheless taken from Capretta, but the induction principle for the inductive family is genuinely new work. It profits from the fact that our map function $map_{\mu F}$ will not be recursive at all.³ It is defined by case analysis on the inductive constructor – this definition principle is called *inversion* in theorem provers like Coq. The system is not just “simultaneous induction-inversion” since iteration in Mendler’s style also has to be justified simultaneously with the inductive generation process.

The next section introduces the important concepts for this article and discusses how Mendler’s style for nested datatypes can also be used in CIC. The problems will be shown and partial solutions sketched. Section 3 contains the precise description of the extension of the CIC we propose under the name *LNMI* for “logic of natural Mendler-style iteration”. It will be proven in *LNMI* that the iterator only produces natural transformations—under well-motivated assumptions, and that the computation rule for the Mendler-style iterator uniquely determines that iterator (again under reasonable assumptions). An essential ingredient of this system is a generalized datatype constructor *In* that can produce non-canonical elements of the nested datatype μF . In Section 4, we look back at the canonical elements with which we started in Section 2 and see that *LNMI* is well-behaved for those canonical elements. Section 5 proves that *LNMI* can be *defined* within CIC with impredicative *Set* plus proof-irrelevance. Section 6 gives a further illustration of the richness of the allowed nested datatypes with a study of the evaluation of explicit flattenings as an example of true nesting. Some conclusions are drawn and further work indicated. Coq vernacular files for the results are provided on the author’s web page (Matthes, 2008).

This article is based on a workshop contribution (Matthes, 2006). The most important conceptual change is the clarification of the role of impredicativity: While the earlier article just assumed *Set* to be impredicative, the specification of *LNMI* is now done with predicative *Set* (the default type-theoretic system of Coq since version 8.0 is the “pCIC”, namely the Predicative Calculus of (Co)Inductive Constructions⁴), and only the justification is done impredicatively. This also required a modularization of the Coq scripts. Moreover, closure properties of the datatype “functors” F and case studies with the representation of explicit flattening in untyped lambda-calculus and operations on “bushy” lists have been added.

³ For *canonical* elements of μF , the expected recursive equation does hold, see Section 4, but the general definition involves no recursive calls.

⁴ The impredicative CIC proves the negation of the law of excluded middle in *Set*, following an idea by Hurkens, see the FAQ on the Coq home page. It would certainly be an interesting question whether *LNMI* alone is incompatible with the law of excluded middle in *Set*.

2 Towards the System

In this article, the only nested datatypes we study are fixed points of endofunctions on type transformations. More precisely, this will mean the following. Let κ_0 stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. In pCIC (the predicative calculus of inductive constructions, as defined in the Coq manual), this will be the sort *Set*. Hence, $\kappa_0 := \text{Set}$. Then, let κ_1 be the kind of type transformations, hence $\kappa_1 := \kappa_0 \rightarrow \kappa_0$. A typical example would be *List* of kind κ_1 , where *List**A* is the type of finite lists with elements from type *A*. Finally, the endofunctions on type transformations shall be the type constructors of kind $\kappa_2 := \kappa_1 \rightarrow \kappa_1$. A prominent example is self-composition $\lambda X^{\kappa_1} \lambda A^{\kappa_0} . X(XA)$.

In this section, we fix a type constructor *F* of kind κ_2 . It need not be closed and might even just be a variable.⁵ We are interested in its least fixed point μF of kind κ_1 . This type transformation μF is to be seen as the inductive family $(\mu F A)_{A:\text{Set}}$ where the index runs through all types in *Set*.

Our running example will be that of “bushes” (Bird & Meertens, 1998). Define

$$\text{Bush}F := \lambda X^{\kappa_1} \lambda A^{\kappa_0} . 1 + A \times X(XA)$$

with one-element type 1 (the only element is denoted by *tt*), product \times with pairing notation (\cdot, \cdot) and disjoint sum $+$ with injections *inl* and *inr*. Its least fixed point $\mu \text{Bush}F$ shall be denoted *Bush* (its existence will be discussed below), and being fixed point of *Bush**F* can intuitively be expressed by the equation

$$\text{Bush}A = 1 + A \times \text{Bush}(\text{Bush}A) .$$

Compare this with the equation for *List*:

$$\text{List}A = 1 + A \times \text{List}A .$$

A list of *A*’s either corresponds to the element of 1, called empty list \square or an element *a* of *A*, followed by a list ℓ of *A*’s (denoted by $a :: \ell$). Likewise, a bush of *A*’s is either the empty bush or an element of *A*, followed by a bush of bushes of *A*. This is list-like, with the difference that the *i*th element is of type $\text{Bush}^i A$, $i \geq 0$. As the inventors of *Bush* write, “at each step down the list, entries are ‘bushed’” (Bird & Meertens, 1998). For a given fixed type *A*, one can fully understand the inductive definition of *List**A*. The family member *Bush**A* of the inductive family *Bush* cannot be understood in isolation since the recursion refers to all the types $\text{Bush}^i A$. This is the feature that constitutes a nested datatype and not just a parameterized inductive datatype: inhabitants of $\mu F A$ are constructed from inhabitants of types that involve $\mu F A'$ for a type $A' \neq A$. If *A* is instantiated with a type variable, it also becomes clear that A' cannot be a smaller type, in whatever sense of the word. Hence, this is in no way a construction by recursion on the family index.

⁵ Later on, certain categorically motivated properties and some extensionality will be required.

2.1 Mendler's Style

There are different possibilities to specify μF . We follow the Mendler-style formulation for higher kinds that has been embodied in system $MI\tau^\omega$ (Abel *et al.*, 2005).⁶ First, we need an abbreviation for polymorphic function space: For $X, Y : \kappa_1$, define the type

$$X \subseteq Y := \forall A : Set. XA \rightarrow YA .$$

The expression $X \subseteq Y$ is of kind *Type* (which is also the kind of *Set* and does itself not allow impredicative constructions) since we want to work in *predicative* CIC, i. e., with predicative *Set*. In the sequel, we will also write types as superscripts to variables instead of after the colon, as in $\forall A^{Set}. XA \rightarrow YA$, if this does not lead to multiple superscripts.

μF is specified by three ingredients: an introduction rule for constructing elements of $\mu F A$, an elimination rule for using elements of $\mu F A$ in a disciplined fashion—in our case, this is plain iteration—and a reduction rule for computing the iteration. Introduction and elimination are provided by two constants:

$$\begin{aligned} in & : F(\mu F) \subseteq \mu F , \\ MI\tau & : \forall G^{\kappa_1}. (\forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G) \rightarrow \mu F \subseteq G . \end{aligned}$$

The reduction rule is

$$MI\tau G s A (in A t) \longrightarrow s(\mu F) (MI\tau G s) A t .$$

Here, $t : F(\mu F)A$ and $s : \forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G$. The latter is called the *step term* of the iteration since it provides the inductive step that extends the function from the type transformation X that is to be viewed as approximation to μF , to a function from FX to G . Here, function means an inhabitant of the universally quantified implication, hence a polymorphic function.

In the example of bushes, *in* has type $\forall A^{Set}. 1 + A \times Bush(Bush A) \rightarrow Bush A$, which allows to define

$$\begin{aligned} bnil & : \forall A^{Set}. Bush A , \\ bcons & : \forall A^{Set}. A \rightarrow Bush(Bush A) \rightarrow Bush A \end{aligned}$$

by $bnil := \lambda A^{Set}. in A (inl tt)$ and $bcons := \lambda A^{Set} \lambda a^A \lambda b^{Bush(Bush A)}. in A (inr(a, b))$.

We define a function $BtL : Bush \subseteq List$ (BtL is a shorthand for $BushToList$) that gives the list of all elements in the bush:

$$\begin{aligned} BtL & := MI\tau List (\lambda X^{\kappa_1} \lambda it^{X \subseteq List} \lambda A \lambda t^{Bush F X A}. \text{match } t \text{ with } inl _ \mapsto [] \\ & \quad | inr(a^A, b^{X(XA)}) \mapsto a :: flat_map (XA) A (it A) (it (XA) b)) . \end{aligned}$$

Here, we used the operation $flat_map : \forall A^{Set} \forall B^{Set}. (A \rightarrow List B) \rightarrow List A \rightarrow List B$, where $flat_map f \ell$ concatenates all the lists $f a$ for the elements a of ℓ . Moreover, pattern matching is used intuitively. Note that when the term t of type $Bush F X A$ is matched with $inr(a, b)$, the variable b is of type $X(XA)$. This is the essence of Mendler's style: the recursive calls come in the form of uses of *it* that does not have type $Bush \subseteq List$ but just $X \subseteq List$, and the

⁶ That system contains nested datatypes of arbitrary ranks, here we concentrate on the case $\kappa = \kappa_1$ and therefore omit the superscripts altogether.

type arguments of the datatype constructors are replaced by variants that only mention X instead of $Bush$. So, the definitions have to be uniform in that type transformation variable X , but this is already the only requirement to ensure termination, see below. Writing \longrightarrow^+ for the transitive closure of all the reduction rules, one easily verifies

$$\begin{aligned} BtLA(bnilA) &\longrightarrow^+ \square , \\ BtLA(bconsAab) &\longrightarrow^+ a :: flat_map(BtLA)(BtL(BushA)b) , \end{aligned}$$

where we have omitted the type arguments to $flat_map$. The recursive call $BtL(BushA)b$ is already with a different type parameter (this is called polymorphic recursion), but the mapping goes beyond usual intuitions of recursive calls: it is the function $BtLA$ that is mapped over the result of the other recursive call.

In what follows, type and constructor arguments will be omitted if they may be reconstructed mechanically. In Coq, this will be possible by the mechanism of implicit arguments that is available since version 8.0. The reduction rule is thus written as

$$MIts(int) \longrightarrow s(MIts)t .$$

However, it should be kept in mind that the formal parameter X in the type of s is instantiated with μF . In the previously cited article (Abel *et al.*, 2005), a direct definition within F^ω (Girard, 1972) of a slight reformulation, using a function symbol of arity 1 instead of the constant MIt , is shown. That translation also simulates the reduction rule, in the sense that a reduction step with our new rule is transformed into at least one rewrite step of F^ω . Thus, since this mentioned transformation behaves well with respect to both type and term substitution, strong normalization follows from that of F^ω . Recall that no requirement at all is imposed on $F : \kappa_2$ for this result, which is still obtained by a Church encoding, i. e., a generalization of the construction of polymorphic Church numerals that works uniformly in F .

2.2 Mendler's Style with Non-canonical Elements

The aim of this work is to provide a dependently-typed analogue of the elimination rule: a rule in the format of an induction principle that, given some predicate $P : \forall A^{Set}. \mu FA \rightarrow Prop$ with $Prop$ the sort of propositions in pCIC, allows to conclude that P holds universally, i. e., proves the proposition $\forall A^{Set} \forall r^{\mu FA}. P_A r$ from a suitable inductive step (the argument A is written as an index to P for enhanced clarity; for this purpose, indexing will often be done in the sequel). Strictly speaking, the author does not have a proposal for such an induction principle that would be justifiable in CIC or a consistent extension thereof: the least solution μF that is generated just from $in : F(\mu F) \subseteq \mu F$ cannot yet be treated by the author.

The way out will be a more liberal datatype constructor than in . The straightforward generalization of the rule μI by Uustalu & Vene (1997) – later used as the introduction rule of system $UVIT$ in the author's thesis (Matthes, 1998) and called $mapwrap$ in the journal version (Uustalu & Vene, 2002) – to nested datatypes would have a datatype constructor in' of type

$$\forall X^{\kappa_1}. X \subseteq \mu F \rightarrow FX \subseteq \mu F .$$

An induction principle for nested datatypes in intensional type theory 7

If in' is instantiated with $X := \mu F$ and given the polymorphic identity on μF as an argument, the result type is $F(\mu F) \subseteq \mu F$, which previously was the type of in . By further instantiation to a type A and application to a term of type $F(\mu F)A$, we get elements of μFA that will be called *canonical*.

In the example of bushes, one would define the generalized datatype constructors as follows:

$$\begin{aligned} \mathit{bnil}' & : \quad \forall X^{\kappa_1} \forall A^{\mathit{Set}}. X \subseteq \mathit{Bush} \rightarrow \mathit{Bush} A \ , \\ \mathit{bnil}' & := \quad \lambda X^{\kappa_1} \lambda A^{\mathit{Set}} \lambda j^{X \subseteq \mathit{Bush}}. \mathit{in}' X j A (\mathit{inl} tt) \ , \\ \mathit{bcons}' & : \quad \forall X^{\kappa_1} \forall A^{\mathit{Set}}. X \subseteq \mathit{Bush} \rightarrow A \rightarrow X(XA) \rightarrow \mathit{Bush} A \ , \\ \mathit{bcons}' & := \quad \lambda X^{\kappa_1} \lambda A^{\mathit{Set}} \lambda j^{X \subseteq \mathit{Bush}} \lambda a^A \lambda b^{X(XA)}. \mathit{in}' X j A (\mathit{inr}(a, b)) \ , \end{aligned}$$

and one would obtain from them bnil and bcons with the previously shown typing by

$$\begin{aligned} \mathit{bnil} & := \quad \lambda A^{\mathit{Set}}. \mathit{bnil}' \mathit{Bush} A (\lambda A^{\mathit{Set}} \lambda x^{\mathit{Bush} A}. x) \ , \\ \mathit{bcons} & := \quad \lambda A^{\mathit{Set}} \lambda a^A \lambda b^{\mathit{Bush}(\mathit{Bush} A)}. \mathit{bcons}' \mathit{Bush} A (\lambda A^{\mathit{Set}} \lambda x^{\mathit{Bush} A}. x) a b \ . \end{aligned}$$

Terms of the form $\mathit{bnil} A$ or $\mathit{bcons} A a b$ would denote “canonical bushes”.

The single datatype constructor in' specifies the inductive family μF in impredicative CIC since the reference to μF in the antecedent is strictly positive. Impredicativity of Set is needed here in order to ensure that μFA is a type in Set and not only in Type . Then, CIC will have canonical elimination rules associated with μF , and Coq will generate them. The minimality scheme for sort Set (as it is called in Coq) will be typed by

$$\forall G^{\kappa_1}. (\forall X^{\kappa_1}. X \subseteq \mu F \rightarrow X \subseteq G \rightarrow FX \subseteq G) \rightarrow \mu F \subseteq G \ .$$

This is the lifting of the type of Mendler’s *recursor* (Mendler, 1987) to nested datatypes. And the generated induction principle has a type that supports the following reasoning: given a predicate P as above, i. e., $P : \forall A^{\mathit{Set}}. \mu FA \rightarrow \mathit{Prop}$, we may deduce P holds universally, i. e., $\forall A^{\mathit{Set}} \forall r^{\mu FA}. P_A r$, if, for every $X : \kappa_1$ and every $j : X \subseteq \mu F$, from the *inductive hypothesis*

$$\forall A^{\mathit{Set}} \forall x^{XA}. P_A(j_A x)$$

we can infer (this is called the *inductive step*)

$$\forall A^{\mathit{Set}} \forall t^{FXA}. P_A(\mathit{in}' j t) \ .$$

In other words, the principle is as follows:

$$\begin{aligned} \forall P : \forall A^{\mathit{Set}}. \mu FA \rightarrow \mathit{Prop}. \left(\forall X^{\kappa_1} \forall j^{X \subseteq \mu F}. (\forall A^{\mathit{Set}} \forall x^{XA}. P_A(j_A x)) \right. \\ \left. \rightarrow \forall A^{\mathit{Set}} \forall t^{FXA}. P_A(\mathit{in}' j t) \right) \rightarrow \forall A^{\mathit{Set}} \forall r^{\mu FA}. P_A r \ . \end{aligned}$$

Note that the link in the inductive step comes from j and not from the argument t . What does the inductive step require in the case of canonical elements, i. e., for $X := \mu F$ and j the polymorphic identity on μF ? The inductive hypothesis in this case is—after normalizing away the β -redex that is implicitly done in the CIC—the proposition $\forall A^{\mathit{Set}} \forall r^{\mu FA}. P_A r$, which amounts to the conclusion of the whole induction principle. The induction step in this case is thus a triviality. Therefore, the case that is dedicated to deal with the canonical elements of the family μF does in no way contribute to the induction step. Hence, the conclusion of the induction principle can only be justified from the induction step in the

cases that produce non-canonical elements. We conclude that our reasoning is entirely based on non-canonical elements.⁷

By ignoring the additional hypothesis $X \subseteq \mu F$ in the step term of the above-mentioned minimality scheme, we can get back *MIt* with the original type, and the following equation holds even with respect to convertibility:⁸

$$MIts(in' jt) = s(\lambda A. (MIts)_A \circ j_A)t ,$$

where $s : \forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G$, $j : X \subseteq \mu F$, $t : FXA$, and $g \circ f$ denotes function composition $\lambda x. g(fx)$ (for types of f and g that fit together). So, here X is instantiated with X itself and the shown first argument to s gets type $X \subseteq G$, hence both sides of the equation get type GA . Note also that we no longer indicate the type *Set* of bound variables named A , B or C .

With these iteration and induction principles, one may try to program and verify functions on nested datatypes. This will be especially interesting in the case of *truly nested datatypes* since they are not directly supported by the CIC. Here, truly nested datatype shall mean: The inductive family has at least one datatype constructor for which one of the argument types has a nested call to the family name, i. e., the family name appears somewhere inside the type argument of the family name occurrence in the argument type of that datatype constructor. Nested datatypes that are not truly nested are called linear nested datatypes by Bird & Paterson (1999a). *Bush* is a truly nested datatype: The second term argument to *bcons* has type *Bush*(*Bush* A). Here *Bush* occurs with argument *Bush* A that makes a reference to *Bush*. Another canonical example is a higher-order representation of de Bruijn terms with an explicit notion of flattening for which elimination of flattening can be programmed (Abel *et al.*, 2005), see Section 6. Another extension of de Bruijn terms that yields a truly nested datatype *TermE* is described by Bird & Paterson (1999b).

2.3 Enriching Mendler's Style with Laws

The running reference of the present work (Abel *et al.*, 2005) shows some programs on nested datatypes (including on truly nested datatypes) but does not at all aim at verifying them. It turns out, however, that the induction principle of the system just described would be too weak for that purpose. Here is an intuitive argument why this is so. One of the first programs for a nested datatype is a map function that applies some function $f : A \rightarrow B$ to all elements of type A contained in the data structure of type $\mu F A$:

$$map_{\mu F} : \forall A \forall B. (A \rightarrow B) \rightarrow \mu F A \rightarrow \mu F B ,$$

just as the usual function

$$map : \forall A \forall B. (A \rightarrow B) \rightarrow List A \rightarrow List B$$

for lists. It is well-known that *map* satisfies the functor laws of category theory. We would like to establish the functor laws for this generic $map_{\mu F}$ as well, namely that $map_{\mu F}$

⁷ This is unfortunate since it does not support intuitive reasoning. But, fortunately, proof assistants like Coq allow to ensure sound proofs also in those situations.

⁸ This rule also appeared in Peter Aczel's presentation at TYPES 2003.

An induction principle for nested datatypes in intensional type theory 9

behaves as the morphism part of a functor whose object part is just μF . With our induction principle, this will not be possible since we do not know anything about the parameter X itself.

The first main idea here is to require that X is accompanied by some map term $m : mon X$ where (*mon* stands for “monotonicity”)

$$mon X := \forall A \forall B. (A \rightarrow B) \rightarrow XA \rightarrow XB ,$$

(*mon X* has type *Type*) and require that m is functorial: it satisfies

$$\begin{aligned} fct_1 m &:= \forall A \forall x^{XA}. mAA (\lambda y. y) x = x , \\ fct_2 m &:= \forall A, B, C \forall f^{A \rightarrow B} \forall g^{B \rightarrow C} \forall x^{XA}. mAC (g \circ f) x = mBC g (mAB f x) , \end{aligned}$$

that are called the first and the second functor law in the sequel. The equality sign $=$ means propositional equality which is implemented by the inductively defined Leibniz equality in the CIC. It is the basis of the rewriting mechanism of Coq that goes beyond definitional equality of the CIC. Definitional equality comes in the form of the fixed built-in and automatically animated convertibility relation \simeq which is implemented as one fixed strongly normalizing and confluent rewrite system.⁹ Universally quantified equations such as $fct_1 m$ and $fct_2 m$ have the impredicative kind *Prop* of computationally irrelevant types. We may place such equations as further premisses into our datatype declarations. The functor laws will not be sufficient, though.

Rewriting in intensional theories such as the CIC with Leibniz equality cannot take place under binders such as λ -abstraction, unlike convertibility that can be applied to arbitrary subterms: in CIC, we do not have extensionality for function spaces with respect to this propositional equality. The addition of extensionality as an axiom without extra rules for definitional equality would destroy canonicity in the sense that not every closed term of the type of natural numbers would be definitionally equal to a numeral (a term of the form $S^i 0$ with S the successor function). There are deep studies (Hofmann, 1995; Altenkirch, 1999; Oury, 2005) on a reconciliation of intensional type theory with extensionality for function spaces. However, the present article will stick with the CIC.

Since the additional functoriality conditions are just equations and therefore do not affect convertibility, truly nested datatypes with their tendency to favour programming with functional arguments (Abel *et al.*, 2005) call for a special attention to means to ensure that rewriting can nevertheless take place. The second main idea here is that most of the functionals that occur during programming only depend on the extension of their function arguments. This is typically so for map terms, hence we define

$$ext m := \forall A \forall B \forall f, g : A \rightarrow B. (\forall a^A. fa = ga) \rightarrow \forall r^{XA}. mAB fr = mAB gr .$$

With these definitions, we might now *define* (strictly speaking, only with impredicative *Set*) the type transformation μF by

$$in'' : \forall X^{K_1} \forall m^{mon X}. ext m \rightarrow fct_1 m \rightarrow fct_2 m \rightarrow X \subseteq \mu F \rightarrow FX \subseteq \mu F .$$

⁹ The CIC is an intensional type theory in the sense that there is no rule that infers \simeq from $=$. This would be the equality reflection rule of extensional type theory.

This time, it is much harder to obtain canonical elements: when instantiating X to μF , then we first need to find a map term $map_{\mu F} : mon(\mu F)$ and then show extensionality and functoriality for $map_{\mu F}$, before the polymorphic identity on μF can be given as a further argument to in'' .

In order to avoid overly long formulae in the sequel, the map term m and the three proofs e, f_1, f_2 of $ext\ m, fct_1\ m$ and $fct_2\ m$ are organized as a dependently typed record $\mathcal{E}X$ (expressing that X is an extensional functor) where the type of the fields e, f_1, f_2 depends on the field m . Given a record ef , i. e., an element of type $\mathcal{E}X$, Coq's notation for its field m is mef , and likewise for the other fields. We adopt this notation instead of the more common $ef.m$.

Canonical elements can now be obtained from the following preservation property for F : there has to be a term ($Fp^{\mathcal{E}}$ stands for “ F preserves extensional functors”)

$$Fp^{\mathcal{E}} : \forall X^{\kappa_1}. \mathcal{E}X \rightarrow \mathcal{E}(FX).$$

Note that it would be too demanding to require that monotone X 's were transported to monotone FX 's and that extensionality and the functoriality properties were each preserved separately. In particular, advanced examples require extensionality in order to establish functoriality, see the remark in the proof of Lemma 1 on page 13. That lemma also provides a closed such term $Fp^{\mathcal{E}}$ in the case of $F := BushF$.

With $Fp^{\mathcal{E}}$ and the Mendler recursor (technically speaking, the minimality scheme for sort Set generated from in'' by Coq), one can define $map_{\mu F}$, and one does not even need the possibility to make recursive calls in the definition (but the argument of type $X \subseteq \mu F$ of the step term is essential). Then, induction simultaneously establishes the three properties for $map_{\mu F}$, always using the preservation property of F embodied in $Fp^{\mathcal{E}}$. Notice that this just requires preservation of properties. Functoriality of F is not expressed at all.

However, there is a problem with this approach. Recall the definition of $BtL : Bush \subseteq List$ on page 5 that makes still perfectly sense in the system we are now considering. The type transformation $List$ is called the *target constructor* of the Mendler iteration (in general, it is the instance for the variable G in that scheme). In our case, $G := List$ is monotone in the sense that there exists the operation map for lists. The term $map_{\mu BushF}$ we did not describe in detail shall be denoted by $bush$. A natural question is whether BtL behaves as a natural transformation from $(Bush, bush)$ to $(List, map)$. In general, for $X, Y : \kappa_1$, $mX : monX$, $mY : monY$ and $j : X \subseteq Y$, we define the proposition

$$j \in \mathcal{N}(mX, mY) := \forall A \forall B \forall f^{A \rightarrow B} \forall t^{XA}. j_B(mX A B f t) = mY A B f(j_A t) ,$$

which just says that j is a natural transformation from (X, mX) to (Y, mY) . Note that functoriality of either side is not required. For BtL , this would mean the following property:

$$\forall A \forall B \forall f^{A \rightarrow B} \forall t^{BushA}. BtL(bush f t) = map f (BtL t) .$$

However, a proof does not seem possible, and this is not due to the specific situation of BtL . More generally, one would want to prove that, for a monotone target constructor G —with map term $mG : mon G$ —and a step term $s : \forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G$ with “reasonable” properties (see below), $MIt\ s : \mu F \subseteq G$ behaves like a natural transformation from $(\mu F, map_{\mu F})$ to (G, mG) .

The “reasonable” property above would be

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall it^{X \subseteq G}. it \in \mathcal{N}(mef, mG) \rightarrow sit \in \mathcal{N}(m(Fp^{\mathcal{E}}ef), mG) .$$

Any inductive proof of $MIt s \in \mathcal{N}(map_{\mu F}, mG)$ will break down since there is no information available about the argument of type $X \subseteq \mu F$ of in'' . Let us call this argument j . Then, we would need $j \in \mathcal{N}(m, map_{\mu F})$ in order to complete that proof attempt. Hence, instead of in'' , we want a datatype constructor In of type

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F}. j \in \mathcal{N}(mef, map_{\mu F}) \rightarrow FX \subseteq \mu F .$$

This constructor declaration cannot define μF since $map_{\mu F} : mon(\mu F)$ refers to the μF defined through in'' . So, this $map_{\mu F}$ has to be defined anew, by recursion on the fixed point μF about to be defined by In . The situation is thus: the inductive family μF has to be given simultaneously with the recursive function $map_{\mu F}$ whose type is isomorphic with $\mu F \subseteq G$, where

$$G := \lambda A \forall B. (A \rightarrow B) \rightarrow \mu F B .$$

The type transformation G is a syntactic form of the right Kan extension of μF along the identity and has been used by the author to define map functions for nested datatypes since (Matthes, 2001). So, we may say that μF is the source type constructor of $map_{\mu F}$, and that the recursion is over μF . Unfortunately, the target type constructor G involves μF again, which excludes this situation from being covered by previous formulations of simultaneous induction-recursion, see the introduction. Nevertheless, it is a simultaneous inductive-recursive definition in a broad sense, and Capretta’s idea (Capretta, 2004) for its justification remains applicable, as will be seen in Section 5.

3 The System

We will call *LNMI* (“logic for natural Mendler-style iteration of rank 2”)¹⁰ the extension of pCIC by the following ingredients and later prove that they can already be defined in CIC with impredicative *Set* plus proof-irrelevance (in sort *Prop*, every proposition has at most one proof with respect to $=$).

3.1 Logic for Natural Mendler-style Iteration of Rank 2

Assume $F : \kappa_2$ and $Fp^{\mathcal{E}} : \forall X^{\kappa_1}. \mathcal{E}X \rightarrow \mathcal{E}(FX)$, possibly in some typing context that will become the typing context of all the constants to be introduced.¹¹ F and $Fp^{\mathcal{E}}$ will be parameters of the extension of pCIC by μF , $map_{\mu F}$, In , MIt and $\mu FInd$ that are specified as shown in Figure 1. The nested datatype μF has kind κ_1 , the map function $map_{\mu F}$ has type $mon(\mu F)$, the datatype constructor In is of the type already shown before, as well as the iterator MIt . The equational rules for $map_{\mu F}$ and MIt hold even definitionally, that is with respect to the convertibility relation we denote by \simeq .

¹⁰ For rank 1, naturality does not make any sense because, for inductive *types*, only a non-polymorphic function from a monotype μF to a monotype B is defined.

¹¹ All the examples so far have been carried out in the empty typing context.

Parameters:	
F	$: \kappa_2$
$Fp^{\mathcal{E}}$	$: \forall X^{\kappa_1}. \mathcal{E}X \rightarrow \mathcal{E}(FX)$
Constants:	
μF	$: \kappa_1$
$map_{\mu F}$	$: mon(\mu F)$
In	$: \forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F}. j \in \mathcal{N}(mef, map_{\mu F}) \rightarrow FX \subseteq \mu F$
MIt	$: \forall G^{\kappa_1}. (\forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G) \rightarrow \mu F \subseteq G$
$\mu FInd$	$: \forall P : \forall A. \mu FA \rightarrow Prop. \left(\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(mef, map_{\mu F})}. \right.$ $\left. (\forall A \forall x^{XA}. P_A(j_A x)) \rightarrow \forall A \forall t^{FXA}. P_A(In ef j nt) \right)$ $\rightarrow \forall A \forall r^{\mu FA}. P_A r$
Rules:	
$map_{\mu F} f (In ef j nt)$	$\simeq In ef j n (m(Fp^{\mathcal{E}} ef) ft)$
$MIt s (In ef j nt)$	$\simeq s(\lambda A. (MIt s)_A \circ j_A) t$
$\lambda A \lambda x^{\mu FA}. (MIt s)_A x$	$\simeq MIt s$

Fig. 1. Specification of $LNMI$.

We may say that the induction principle $\mu FInd$ is just the obvious adaptation of the principle we had for the system based on in' , given the two new arguments $ef : \mathcal{E}X$ and n of type $j \in \mathcal{N}(mef, map_{\mu F})$ of the datatype constructor In . Despite this simplicity, it is problematic due to the occurrence of $map_{\mu F}$ in the type of n , see the discussion in the previous section. The definitional rule for $map_{\mu F}$ may seem curious since $map_{\mu F}$ does not appear on the right-hand side. For canonical elements, as discussed in Section 4, the behaviour will nevertheless be the ordinary recursive one. Since definitional equality is contained in propositional equality, the rule for $map_{\mu F}$ immediately implies

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(mef, map_{\mu F})}. (In ef j n)^{FX \subseteq \mu F} \in \mathcal{N}(m(Fp^{\mathcal{E}} ef), map_{\mu F}) .$$

MIt 's definitional behaviour even ignores the arguments ef and n . And the last rule (with the implicit proviso that x does not occur free in s) is just there for technical reasons, to become clear in the proof of Theorem 3.

The fact that $LNMI$ has the first two definitional equations and not just propositional ones brings the termination guarantee in Section 5, where $LNMI$ is defined within the extension of CIC by only one propositional axiom. The convertibility relation of the CIC is decidable through an implementation as a strongly normalizing and confluent term rewrite system. Although we cannot say that the left-hand sides of the two equations will be rewritten to the corresponding right-hand sides, we know by confluence that both sides will be normalized to the same term. Evidently, the left-hand sides are not normal (for the translation into CIC), so that the calculated normal form will not contain instances of the left-hand sides, hence the calculated normal form is also normal with respect to the extension of the rewrite system of the CIC by the two rewrite rules

$$\begin{aligned} map_{\mu F} f (In ef j nt) &\longrightarrow In ef j n (m(Fp^{\mathcal{E}} ef) ft) , \\ MIt s (In ef j nt) &\longrightarrow s(\lambda A. (MIt s)_A \circ j_A) t . \end{aligned}$$

The stronger result that this extended rewrite system is *strongly* normalizing would require an extension of the proof of strong normalization of the CIC, but we content ourselves with having normal forms.

The datatype functors $F : \kappa_2$ that are covered by *LNMI* include all the *nested hofunctors* (Martin *et al.*, 2004). This is to say that for every nested hofunctor F , there exists the associated polymorphic operation $Fp^{\mathcal{E}}$, i. e., a closed term of type

$$p^{\mathcal{E}}F := \forall X^{\kappa_1}. \mathcal{E}X \rightarrow \mathcal{E}(FX) .$$

This will be made explicit in the following. For $op \in \{\times, +\}$ and $X, Y : \kappa_1$, define $X op Y := \lambda A. XA op YA : \kappa_1$. For $X, Y : \kappa_1$, define $X \circ Y := \lambda A. X(YA) : \kappa_1$.

Lemma 1 (Closure properties of \mathcal{E})

There are closed terms of the following types:

- $\mathcal{E}(\lambda A.A)$
- $\forall C. \mathcal{E}(\lambda A.C)$
- $\mathcal{E}(\lambda A. optionA)$ where *optionA* is the type that has exactly one more element than A
- $\forall X^{\kappa_1} \forall Y^{\kappa_1}. \mathcal{E}X \rightarrow \mathcal{E}Y \rightarrow \mathcal{E}(X op Y)$ for $op \in \{\times, +, \circ\}$

Proof

Fairly elementary reasoning. For the last item with $op = \circ$, note that we need extensionality of the map term for Y in order to prove the functoriality laws. \square

Lemma 2 (Closure properties of $p^{\mathcal{E}}$)

There are closed terms of the following types:

- $p^{\mathcal{E}}(\lambda X.X)$
- $p^{\mathcal{E}}(\lambda X \lambda A.XA)$ —extensionally, the same as the line before
- $\forall X^{\kappa_1}. \mathcal{E}X \rightarrow p^{\mathcal{E}}(\lambda Y.X)$
- $\forall F^{\kappa_2} \forall G^{\kappa_2}. p^{\mathcal{E}}F \rightarrow p^{\mathcal{E}}G \rightarrow p^{\mathcal{E}}(\lambda X.(FX) op(GX))$ for $op \in \{\times, +, \circ\}$

Proof

A simple consequence of the previous lemma. \square

The second lemma may be seen as an inductive definition of nested hofunctors, with its third clause not being confined just to X 's with $\mathcal{E}X$ that stem from the first lemma. But for our examples, those will suffice. For an illustration $p^{\mathcal{E}} BushF$ is obtained as follows: We have $\mathcal{E}(\lambda A.1)$, hence $p^{\mathcal{E}}(\lambda X \lambda A.1)$. We have $\mathcal{E}(\lambda A.A)$, hence $p^{\mathcal{E}}(\lambda X \lambda A.A)$. We have $p^{\mathcal{E}}(\lambda X.X)$ and $p^{\mathcal{E}}(\lambda X \lambda A.XA)$, hence $p^{\mathcal{E}}(\lambda X \lambda A.X(XA))$. Therefore, $p^{\mathcal{E}}(\lambda X \lambda A.A \times X(XA))$ and finally $p^{\mathcal{E}} BushF$. Let us call the obtained term $BushFp^{\mathcal{E}}$.

Unfortunately, the system *LNMI* in its present form cannot deal with datatype functors F that have embedded function spaces. Although they are not excluded by any syntactic restriction, one will not be able to construct the associated $Fp^{\mathcal{E}}$ since, in order to establish extensionality, one would have to prove Leibniz equality of functions that call functions f, g that are only extensionally equal.

3.2 Naturality and Uniqueness of $MIt s$

In $LNMI t$, we can now prove the following theorem that could not be proven in the systems of Section 2:

Theorem 1 (Naturality of $MIt s$)

Assume $G : \kappa_1$, $mG : mon G$, $s : \forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G$ and that the following holds:

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall it^{X \subseteq G}. it \in \mathcal{N}(mef, mG) \rightarrow sit \in \mathcal{N}(m(Fp\mathcal{E}ef), mG) .$$

Then $MIt s \in \mathcal{N}(map_{\mu F}, mG)$, hence $MIt s$ is a natural transformation for the respective map terms.

Proof

This is done by induction with

$$P := \lambda A \lambda r^{\mu F A \forall B \forall f^{A \rightarrow B}}. MIt s(map_{\mu F} f r) = mG f(MIt s r) .$$

Assume X, ef, j, n as prescribed. The induction hypothesis is

$$\forall A \forall X^{XA} \forall B \forall f^{A \rightarrow B}. MIt s(map_{\mu F} f(j_A x)) = mG f(MIt s(j_A x)) .$$

Further assume A, t, B, f . It remains to show

$$MIt s(map_{\mu F} f(In ef j nt)) = mG f(MIt s(In ef j nt)) .$$

Abbreviate $it := \lambda A. (MIt s)_A \circ j_A$. By the equational rules for $map_{\mu F}$ and MIt , the previous equation is equivalent to

$$sit(m(Fp\mathcal{E}ef)ft) = mG f(sit) .$$

We want to apply the assumption of the theorem. It suffices to show $it \in \mathcal{N}(mef, mG)$.

Assume A, B, f, t . Show $it_B(mef ft) = mG f(it_A t)$. Its left-hand side is equivalent to

$$MIt s(j_B(mef ft)) = MIt s(map_{\mu F} f(j_A t)) ,$$

where we used the assumption n of type $j \in \mathcal{N}(mef, map_{\mu F})$ for the last step. Now, the induction hypothesis is applicable. \square

By the help of this theorem, one can easily show that the function BtL of Section 2¹² is natural, i. e., setting again $bush := map_{\mu Bush F}$, one can prove

$$\forall A \forall B \forall f^{A \rightarrow B} \forall t^{Bush A}. BtL(bush ft) = map f(BtL) .$$

The proof in the Coq script (Matthes, 2008) only needs in addition a naturality property of $flat_map$.

With the *length* function for lists, we get a function that calculates the size of bushes (indirectly):

$$size_i := \lambda A \lambda t^{Bush A}. length(BtL t) .$$

Thanks to the above naturality of BtL and because map does not change the list length, we have as immediate consequence that $bush$ does not change the size of bushes:

$$\forall A \forall B \forall f^{A \rightarrow B} \forall t^{Bush A}. size_i(bush ft) = size_i t .$$

¹² Notice that no change is needed for the definition of BtL in Section 2 in order to fit into $LNMI t$.

A more direct definition of the size of bushes is possible, although not just by one direct use of *MIt*. As is usual with nested datatypes, a more general polymorphic function has to be found and then instantiated. Define the monotone type transformation (it is non-strictly positive and constitutes the *continuation monad*)

$$G := \lambda A. (A \rightarrow \text{nat}) \rightarrow \text{nat}$$

and $Btv : \text{Bush} \subseteq G$ (the shorthand stands for “Bush to value”) so that $BtvAt^{BushA} f^{A \rightarrow nat}$ gives the “value” of t , obtained as the sum of the values $f a$ for all the elements a of type A that are contained in t :

$$Btv := MIt G (\lambda X^{\kappa_1} \lambda it^{X \subseteq G} \lambda A \lambda t^{BushFXA}. \text{match } t \text{ with } \text{inl } _ \mapsto \lambda f^{A \rightarrow nat}. 0 \\ | \text{inr}(a^A, b^{X(XA)}) \mapsto \lambda f^{A \rightarrow nat}. f a + it (XA) b (\lambda x^{XA}. it A x f)) .$$

We can now define the more direct size function $size_d : \forall A. \text{Bush} A \rightarrow \text{nat}$ by

$$size_d := \lambda A \lambda t^{BushA}. BtvAt (\lambda a^A. 1) .$$

Note that the naturality statement for Btv according to the theorem would express equality of elements of type GB , which are functionals. The lack of extensionality for functions will not allow us to prove such equations, and the theorem is in fact not applicable because naturality of sit cannot be established, again because that would require equality of functionals in GB . This unfortunately shows a limit of the formulation of the theorem that the author was not yet able to overcome in a generic fashion. However, by a direct use of the induction principle $\mu FInd$, we can establish a “pointwise” version of naturality¹³ for Btv :

$$\forall A \forall B \forall f^{A \rightarrow B} \forall t^{BushA} \forall g^{B \rightarrow nat}. Btv(bush f t) g = Btv t (g \circ f) .$$

For this to work and also for the following conclusion, we first have to establish extensionality of Btv in its function parameter, i. e.,

$$\forall A \forall t^{BushA} \forall f^{A \rightarrow nat} \forall g^{A \rightarrow nat}. (\forall a^A. f a = g a) \rightarrow Btv t f = Btv t g ,$$

which can be proven directly by the induction principle $\mu FInd$. From all this, we immediately get that also $size_d$ is not changed by $bush f$:

$$\forall A \forall B \forall f^{A \rightarrow B} \forall t^{BushA}. size_d(bush f t) = size_d t .$$

It is also possible to prove that $size_i$ and $size_d$ yield the same values for all arguments, see the details in the Coq scripts (Matthes, 2008).

Coming back to the general theory, we show that under reasonable assumptions, *MIt* is uniquely characterized by the equation above:

Theorem 2 (Uniqueness of MIt s)

Assume $G : \kappa_1, s : \forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G$ and $h : \mu F \subseteq G$ (the candidate for being *MIt* s). Assume further the following extensionality property of s (s only depends on the extension of its function argument):

$$\forall X^{\kappa_1} \forall f, g : X \subseteq G. (\forall A \forall x^{XA}. f x = g x) \rightarrow \forall A \forall y^{FXA}. s f y = s g y .$$

¹³ This is with respect to the map term for G , see the Coq proofs for the details.

16

R. Matthes

Assume finally that h satisfies the equation for $MIts$:

$$\forall X \kappa_1 \forall ef^{\mathcal{E}X} \forall j^X \subseteq \mu F \forall n^{j \in \mathcal{N}^{(mef, map_{\mu F})}} \forall A \forall t^{FXA}. h_A(Inef jnt) = s(\lambda A. h_A \circ j_A)t .$$

Then, $\forall A \forall r^{\mu FA}. h_A r = MIts r$.

Proof

Induction is used with the evident $P := \lambda A \lambda r^{\mu FA}. h_A r = MIts r$. Then assume the appropriate X, ef, j, n . The inductive hypothesis is $\forall A \forall x^{XA}. h_A(j_A x) = MIts(j_A x)$. Assume further A, t and show

$$h_A(Inef jnt) = MIts(Inef jnt).$$

Applying the hypothesis on h and the computation rule for $MIts$ yields the following equivalent equation:

$$s(\lambda A. h_A \circ j_A)t = s(\lambda A. (MIts)_A \circ j_A)t .$$

The extensionality assumption on s finishes the proof if we can show

$$\forall A \forall x^{XA}. (h_A \circ j_A)x = ((MIts)_A \circ j_A)x ,$$

but this is the induction hypothesis. \square

Note that the analogue of this uniqueness theorem would have been available also in the theory in Section 2 that did not integrate naturality into the approximations to μF and for which Theorem 1 seemed out of reach. So, it appears to be extremely unlikely that Theorem 1 could be proven from the uniqueness theorem in the system $LNMI$ without the induction principle $\mu FInd$.

A natural example where the uniqueness theorem is useful will be given near the end of the next section (idempotency of Btc on page 19).

4 Back to Canonical Elements

In the last section, we did not make any use of extensionality and the functor laws for extensional functors: only the m components of the extensional functors ef have been used.¹⁴ Now, the other components come into play since they allow to prove extensionality and the functor laws for $map_{\mu F}$ (which were the reason why these properties were introduced for the type transformation variable X in the preliminary system in Section 2), and this in turn only allows in our present setting to define the canonical elements of the nested datatype μF .

4.1 Behaviour on Canonical Elements

Theorem 3 (Canonical Elements in LNMI)

There are terms $ef_{\mu F} : \mathcal{E}\mu F$ and $InCan : F(\mu F) \subseteq \mu F$ (the *canonical* datatype constructor that constructs canonical elements) such that the following convertibilities hold:

¹⁴ Without them, the notion of naturality would not even make sense.

$$\begin{array}{l}
mef_{\mu F} \simeq map_{\mu F} \\
map_{\mu F} f (InCant) \simeq InCan(m(Fp\mathcal{E} ef_{\mu F}) ft) \\
MIts(InCant) \simeq s(MIts)t
\end{array}$$

Thus, for canonical elements, we get back the ordinary behaviour.

Proof

We want to take μF as its own approximation, with $map_{\mu F}$ as the map term. Therefore, we need to establish ext , fct_1 and fct_2 for $map_{\mu F}$. Then, trivially, the polymorphic identity on μF serves as argument j to In , and (lambda-abstracted) reflexivity of equality yields the corresponding proof of naturality. Then, the claimed equations follow from those of $LNMI$, in particular the last equation of $LNMI$'s definition allows to remove the composition of $MIts$ with the identity.

Let us establish extensionality; the functoriality properties are proved analogously. The statement $ext map_{\mu F}$ is logically equivalent with universal validity of the predicate

$$P := \lambda A \lambda r^{\mu F A} \forall B \forall f, g : A \rightarrow B. (\forall a^A. fa = ga) \rightarrow map_{\mu F} f r = map_{\mu F} g r .$$

This is proven by inversion on μF , i. e., by using $\mu FInd$ without the induction hypothesis. Then, it remains to show $\forall A \forall t^{FXA}. P_A(In ef jnt)$ in the usual context with X, ef, j, n . So, assume A, t, B, f, g with $\forall a^A. fa = ga$. Show

$$map_{\mu F} f (In ef jnt) = map_{\mu F} g (In ef jnt) .$$

By convertibility, this amounts to

$$In ef jn(m(Fp\mathcal{E} ef) ft) = In ef jn(m(Fp\mathcal{E} ef) gt) ,$$

which follows from $e(Fp\mathcal{E} ef) : ext(m(Fp\mathcal{E} ef))$. \square

As an instance of our earlier discussion in Section 3 (and presupposing Theorem 4 in the next section), we can now say that the CIC produces normal forms that are also normal with respect to the extension of the CIC's rewrite system by the two rules

$$\begin{array}{l}
map_{\mu F} f (InCant) \longrightarrow InCan(m(Fp\mathcal{E} ef_{\mu F}) ft) , \\
MIts(InCant) \longrightarrow s(MIts)t .
\end{array}$$

It should be noted that the term $m(Fp\mathcal{E} ef_{\mu F})$ in the behaviour of $map_{\mu F}$ can usually be simplified to $Fpmon map_{\mu F}$, namely when there is a map-transforming function $Fpmon$ of type $\forall X^{\kappa_1}. mon X \rightarrow mon(F X)$ such that for all X and $ef : \mathcal{E} X$, $m(Fp\mathcal{E} ef) \simeq Fpmon(m ef)$, i. e., when the map term for $F X$ does not depend on the properties of the map term for X . This is usually the case and leads to the standard behaviour of $map_{\mu F}$ that is given in functional programming languages that do not guarantee termination, unlike the present approach: finally, $map_{\mu F}$ has become recursive, to be seen from

$$map_{\mu F} f (InCant) \simeq InCan(Fpmon map_{\mu F} f t) .$$

The move from non-recursive to recursive comes from the inclusion of $map_{\mu F}$ into the definition of $InCan$!

In the example of bushes, observe that for all $X : \kappa_1$ and $ef : \mathcal{E} X$, we have

$$m(\text{BushFp}\mathcal{E} ef) \simeq \lambda A \lambda B \lambda f^{A \rightarrow B} \lambda x^{\text{BushFXA}}. \text{match } x \text{ with} \\ \text{inly } \mapsto \text{inly} \quad | \quad \text{inry } \mapsto \text{inr}(\text{let } (x_1, x_2) := \text{yin}(fx_1, \text{mef}(\text{mef } f)x_2)) ,$$

from which one can read off a term $Fpmon$ for $F := \text{BushF}$ since the right-hand side only depends on the m -component of ef . (Recall that $\text{BushFp}\mathcal{E}$ has been implicitly defined on page 13.) Just as in Section 2 from datatype constructor in , we may define bnil and bcons from InCan :

$$\begin{aligned} \text{bnil} &:= \lambda A. \text{InCanA}(\text{inl } tt) , \\ \text{bcons} &:= \lambda A \lambda a^A \lambda b^{\text{Bush(BushA)}}. \text{InCanA}(\text{inr}(a, b)) . \end{aligned}$$

This yields the following behaviour of bush (recall that bush stands for $\text{map}_{\mu \text{BushF}}$):

$$\begin{aligned} \text{bush } f^{A \rightarrow B}(\text{bnil } A) &\simeq \text{bnil } B , \\ \text{bush } f^{A \rightarrow B}(\text{bcons } ab) &\simeq \text{bcons}(f a)(\text{bush}(\text{bush } f) b) . \end{aligned}$$

The behaviour of BtL in Section 2 is as described on page 6, but with \simeq in place of \longrightarrow^+ .

We compare with the earlier work (Abel *et al.*, 2005) that proposed systems within the framework of F^ω and hence had no internal means of describing an induction principle. As mentioned in Section 2, the above rule for Mendler iteration could be simulated within F^ω , so we even know that the left-hand side rewrites to the right-hand side in that encoding. Indirectly, also $\text{map}_{\mu F}$ has been implemented through MIt , and in order to justify the rule

$$\text{map}_{\mu F} f(\text{InCan } t) \longrightarrow \text{InCan}(\text{Fpmon } \text{map}_{\mu F} f t) ,$$

we had to insist on the following more general type of Fpmon :

$$\forall X^{\kappa_1} \forall Y^{\kappa_1}. (\forall A, B. (A \rightarrow B) \rightarrow XA \rightarrow YB) \rightarrow \forall A, B. (A \rightarrow B) \rightarrow FXA \rightarrow FYB .$$

By instantiating X and Y both with X and quantification over X , we arrive exactly at the type we gave above for Fpmon in LNMI , thus a less general type. Although in all practical examples, the terms Fpmon also have the more general type, the simplification of the typing requirement obtained in the present article lines up better with programming examples in the literature, e. g. Bird & Paterson (1999b).

4.2 Canonization for the Example of Bushes

This last part of section 4 is a study of how to transform arbitrary bushes into canonical ones. Recall from Section 2 that canonical bushes are those that are denoted by a term of the form $\text{bnil } A$ or $\text{bcons } A ab$.¹⁵ Such a transformation could be defined on the generic level of an arbitrary type constructor $F : \kappa_2$, but there are not yet the necessary generic lemmas for its analysis.

We start with the observation that BushF is monotone in a sense that should be called relativized basic monotonicity of rank 2: there is a closed term

$$\text{BushFmon2br} : \forall X^{\kappa_1} \forall Y^{\kappa_1}. \text{mon } Y \rightarrow X \subseteq Y \rightarrow \text{BushFX} \subseteq \text{BushFY} , \text{ namely}$$

¹⁵ The system in Section 2 has a different definition of bnil and bcons . We now mean the definitions within LNMI in this section.

$$\lambda X^{\kappa_1} \lambda Y^{\kappa_1} \lambda m^{\text{mon}Y} \lambda f^{X \subseteq Y} \lambda A \lambda x^{\text{Bush}FXA}. \text{match } x \text{ with} \\ \text{inl } u \mapsto \text{inl } u \quad | \quad \text{inr } (a, b) \mapsto \text{inr } (a, m(XA)(YA)(fA)(f(XA)b)) .$$

If we had not used the assumed map term m in the pattern-matching construct, $\text{Bush}F$ would have been monotone in the sense of basic monotonicity, studied in detail in Abel *et al.* (2005), where it is proven that self-composition $\lambda X^{\kappa_1} \lambda A^{\kappa_0}. X(XA)$ is not monotone in that sense. In previous work (Matthes, 2001), the author required the present relativized basic monotonicity in order to express an iteration rule for nested datatypes that follows the categorical picture of initial algebras and not Mendler’s style. But there were two more requirements: The existence of a function $Fpmon$, discussed previously in this section, and a dual to relativized basic monotonicity, where $\text{mon}X$ is assumed instead of $\text{mon}Y$. The latter requirement was made in order to be able to give a definition of $\text{map}_{\mu F}$ through the iterator (using syntactic Kan extensions, as mentioned on page 11), but this is not needed in the present article, since $\text{map}_{\mu F}$ is a basic constituent of $LNMI$. Anyway, all three requirements are fulfilled for a very large class of datatype functors $F : \kappa_2$ (Matthes, 2001).

The fact that $\text{Bush}F\text{mon}2br$ is not restricted to monotone *first* arguments allows to define the function $Btc : \text{Bush} \subseteq \text{Bush}$ that “canonizes” bushes (Btc is a shorthand for “bush to canonical bush”) as follows:

$$Btc := MI \text{t Bush } (\lambda X^{\kappa_1} \lambda it^{X \subseteq \text{Bush}} \lambda A \lambda t^{\text{Bush}FXA}. \text{InCan}(\text{Bush}F\text{mon}2br \text{ bush } it)) .$$

Directly from the definition of $\text{Bush}F\text{mon}2br$ and the rule for MI , we get

$$BtcA(\text{Inef } jn(\text{inl } tt)) \simeq \text{bnil } A , \\ BtcA(\text{Inef } jn(\text{inr } (a, b))) \simeq \text{bcons } A a b' ,$$

for some b' that we do not need to know here. Hence, we may say that Btc only yields canonical bushes.

Does Btc provide a “canonization”? The minimum requirement seems to be that canonical elements are left unchanged. For $\text{bnil } A$, this is true, but Theorem 3 yields the other equation

$$Btc(\text{bcons } A a b) \simeq \text{bcons } A a (\text{bush}(BtcA)(Btc(\text{Bush}A)b)) .$$

Since we only have iteration available in $LNMI$, the function acts recursively on the argument b , and we cannot program functions that do not *touch* the recursive arguments.

We will first directly show that Btc is idempotent and then introduce “hereditarily canonical” bushes. Finally, it is shown that Btc yields always those bushes and that it does not change them, in the sense that the result is propositionally equal to the argument. Evidently, from these two properties, idempotency of Btc follows once more.

As a preparation for the idempotency proof, we need naturality of Btc , i. e., $Btc \in \mathcal{N}(\text{bush}, \text{bush})$. This is an easy application of Theorem 1, where the second functor law and extensionality of bush are needed.

Idempotency means: $\forall A \forall t^{\text{Bush}A}. Btc(Btc t) = Btc t$. This is of the form of the conclusion of Theorem 2, with the composition of Btc with itself as the candidate function h . The extensionality assumption of that theorem is covered by extensionality of bush , and the correct recursive behaviour is guaranteed by naturality of Btc and the second functor law for bush .

From *BtL*, we immediately get a notion of elements of bushes: For $a : A$ and $t : \text{Bush}A$, define $a \in t$ by “ a is an element of the list *BtLt*”, where elementhood in lists is a simple recursive definition on lists. By using that elements of *flat_map fl* are elements of the lists fa for a an element of l , one can establish the following two closure rules:

- $\forall A \forall \alpha^A \forall b^{\text{Bush}(\text{Bush}A)}. a \in b \text{cons } a \ b$,
- $\forall A \forall \alpha^A \forall b^{\text{Bush}(\text{Bush}A)} \forall e^A \forall t^{\text{Bush}A}. e \in t \rightarrow t \in b \rightarrow e \in b \text{cons } a \ b$.

Thanks to \in , we can define the notion $\text{can}_H : \forall A. \text{Bush}A \rightarrow \text{Prop}$ of *hereditarily canonical bushes* inductively by the following two clauses:

- $\forall A. \text{can}_H(\text{bnil}A)$,
- $\forall A \forall \alpha^A \forall b^{\text{Bush}(\text{Bush}A)}. (\forall t^{\text{Bush}A}. t \in b \rightarrow \text{can}_H t) \rightarrow \text{can}_H b \rightarrow \text{can}_H(b \text{cons } a \ b)$.

This definition is strictly positive and, formally, infinitely branching. However, there are always only finitely many t that satisfy $t \in b$. Notice that nothing is required for the term a in $b \text{cons } a \ b$. Notice also that this is a simultaneous definition of $\text{can}_H A : \text{Bush}A \rightarrow \text{Prop}$ for all A , where $\text{can}_H b$ is in fact $\text{can}_H(\text{Bush}A) b$ and $\text{can}_H(b \text{cons } a \ b)$ is $\text{can}_H A (b \text{cons } a \ b)$.

A refinement of extensionality for *bush* can be given for hereditarily canonical bushes: We have

$$\forall A \forall B \forall f, g^{A \rightarrow B} \forall t^{\text{Bush}A}. \text{can}_H t \rightarrow (\forall \alpha^A. a \in t \rightarrow f a = g a) \rightarrow \text{bush } f t = \text{bush } g t \text{ .}$$

The proof is by induction on the inductive definition of can_H and uses the two closure rules of \in .

From this refined extensionality and the first functor law¹⁶ for *bush*, we can prove—again by induction on can_H —the invariance of hereditarily canonical bushes under *Btc*:

$$\forall A \forall t^{\text{Bush}A}. \text{can}_H t \rightarrow \text{Btc } t = t \text{ .}$$

Finally, we want to show that *Btc* always produces hereditarily canonical bushes:

$$\forall A \forall t^{\text{Bush}A}. \text{can}_H(\text{Btc } t) \text{ .}$$

As an auxiliary statement, we need that every element of $\text{bush } f t$ is equal to fa for some $a \in t$. It is derived from the corresponding property of *map*, using naturality of *BtL*. The last but one step is that *bush f* preserves the property of being hereditarily canonical:

$$\forall A \forall B \forall f^{A \rightarrow B} \forall t^{\text{Bush}A}. \text{can}_H t \rightarrow \text{can}_H(\text{bush } f t) \text{ .}$$

It is proven by induction on can_H , using the previous auxiliary statement.

The desired $\text{can}_H(\text{Btc } t)$ now comes from induction on bushes, that is, by a direct application of $\mu F \text{Ind}$, where the last two statements are used in the case for *bcons*.

All of this is just an illustration by way of the example of the truly nested datatype of bushes. It would certainly be pleasing not to be obliged to distinguish between all bushes, the canonical bushes and the hereditarily canonical bushes, but an appropriate terminating type-based recursion scheme together with a justified induction principle has not yet been conceived.

¹⁶ This is the only direct use of the first functor law for $\text{map}_{\mu F}$ in this article.

5 Justification

Theorem 4 (Main Theorem)

The system *LNMI*t can be defined within the CIC with impredicative *Set*, extended by the principle of proof irrelevance, i. e., by $\forall P : Prop \forall p_1, p_2 : P. p_1 = p_2$.

The proof will occupy the whole section. Capretta’s idea (Capretta, 2004) is to first introduce something bigger than the desired μF , i. e., a type transformation $\mu^+ F$ such that, later, there is a function of type $\mu F \subseteq \mu^+ F$. In fact, μF will be defined as the restriction of $\mu^+ F$ by some predicate, and the mentioned function will just be the first projection out of that strong sum type. While $\mu^+ F$ will not be a “real” recursive type—there is no recursive call to $\mu^+ F$, hence it is just a record—the predicate is defined inductively with induction hypotheses that are in no way a priori smaller than the conclusion. Abbreviate

$$MI\text{Pretype } S := \forall G^{K_1}. (\forall X^{K_1}. X \subseteq G \rightarrow FX \subseteq G) \rightarrow S \subseteq G .$$

The inductive family $\mu^+ F$ is defined by the datatype constructor

$$\begin{aligned} In^+ : \forall G^{K_1} \forall ef^{\mathcal{E}G} \forall G' : K_1 \forall m' : mon\ G' \\ \forall it : MI\text{Pretype } G' \forall j^{G \subseteq G'} . j \in \mathcal{N}(mef, m') \rightarrow FG \subseteq \mu^+ F . \end{aligned}$$

Certainly, the idea is that G' should be μF , m' should be $map_{\mu F}$ and it should be *MI*t. Unfortunately, the method requires that the iteration principle has to be encoded into the construction from the very beginning onwards. This treatment of simultaneous inductive-recursive definitions is closed in the sense that it does not allow any other functions that are defined by recursion on the family afterwards.

By impredicativity of *Set* that we require for the whole construction, the type $\mu^+ FA$ belongs to *Set*, hence $\mu^+ F : K_1$. The minimality scheme for sort *Set* generated from In^+ by Coq is just case analysis on this record-like $\mu^+ F$. With its help, we can immediately define $map_{\mu^+ F} : mon(\mu^+ F)$ with

$$map_{\mu^+ F} f (In^+ ef\ m'\ it\ j\ nt) \simeq In^+ ef\ m'\ it\ j\ n (m(Fp\mathcal{E}\ ef)\ ft) .$$

Similarly, one defines $MI\text{t}^+ : MI\text{Pretype}(\mu^+ F)$ such that

$$MI\text{t}^+ s (In^+ ef\ m'\ it\ j\ nt) \simeq s(\lambda A. (it\ s)_A \circ j_A) t .$$

Obviously, this has nothing to do with iteration since there is no recursive call whatsoever.

With $map_{\mu^+ F}$ and $MI\text{t}^+$ in place, we can now define what is a “good” element of $\mu^+ F$. Following the ideas by Capretta (2004), this is done by way of an inductive predicate $chk_{\mu^+ F} : \forall A. \mu^+ FA \rightarrow Prop$ for which there is a single inductive clause *Inchk* of type

$$\begin{aligned} \forall G^{K_1} \forall ef^{\mathcal{E}G} \forall j^{G \subseteq \mu^+ F} \forall n^{j \in \mathcal{N}(mef, map_{\mu^+ F})}. (\forall A \forall t^{GA}. chk_{\mu^+ F}(j_A t)) \rightarrow \\ \forall A \forall t^{FGA}. chk_{\mu^+ F} \left(In^+ ef\ map_{\mu^+ F}\ MI\text{t}^+ (\lambda A \lambda t : GA. j_A t) nt \right) . \end{aligned}$$

Let us first remark that the η -expansion $\lambda A \lambda t : GA. j_A t$ of j is needed for subtle technical reasons. Except from that, the parameters of In^+ are instantiated as $G' := \mu^+ F$, $m' := map_{\mu^+ F}$ and $it := MI\text{t}^+$.

This is a strictly-positive inductive definition, hence available in the CIC, and Coq generates an induction principle as follows. Given a predicate $P : \forall A. \mu^+ FA \rightarrow Prop$, P holds

“universally”, which means here that $\forall A \forall r^{\mu^+FA}. chk_{\mu^+F} r \rightarrow P_A r$ holds (so, universality is relativized to the good elements), if the following induction step is provided:

$$\forall G^{\kappa_1} \forall ef^{\mathcal{E}G} \forall j^{G \subseteq \mu^+F} \forall n^{j \in \mathcal{N}(mef, map_{\mu^+F})}. (\forall A \forall t^{GA}. chk_{\mu^+F}(j_A t)) \rightarrow (\forall A \forall t^{GA}. P_A(j_A t)) \rightarrow \forall A \forall t^{FGA}. P_A(In^+ ef map_{\mu^+F} MIt^+(\lambda A \lambda t : GA. j_A t) nt) .$$

The premise $\forall A \forall t^{GA}. chk_{\mu^+F}(j_A t)$ yields the inversion principle for chk_{μ^+F} (i. e., that the “ingredients” of a good element are good), the premise $\forall A \forall t^{GA}. P_A(j_A t)$ is the induction hypothesis.

Slightly sloppily, we can say that an element of μ^+FA is good if it is of the form $In^+ \dots$ where the argument m' is replaced by map_{μ^+F} , it is replaced by MIt^+ and all the j -images are already good. The construction of the nested datatype itself is finished by

$$\mu FA := \{r : \mu^+FA \mid chk_{\mu^+F} r\} .$$

This notation stands for the inductively defined *sig* of Coq which is a strong sum in the sense that the first projection yields the element r and the second projection the proof that $chk_{\mu^+F} r$. Since μ^+FA belongs to *Set*, this is also true of μFA , hence $\mu F : \kappa_1$.

The map function $map_{\mu F}$ for μF can now be defined as follows: Assume $A, r : \mu FA, B$ and $f : A \rightarrow B$. We have to define $map_{\mu F} f r$ of type μFB . An r consists of a term $r' : \mu^+FA$ and a proof $p : chk_{\mu^+F} r'$. The first component of our result will be $map_{\mu^+F} f r'$, the second component has to be a proof that $chk_{\mu^+F}(map_{\mu^+F} f r')$. Now we do inversion on p , i. e., induction on chk_{μ^+F} where the induction hypothesis will not be used in the induction step. This is immediate with the computation rule for map_{μ^+F} and the introduction rule for chk_{μ^+F} , invoking the other hypothesis $\forall A \forall t^{GA}. chk_{\mu^+F}(j_A t)$ that yields the inversion principle.

In order to define In of the required type, assume $X : \kappa_1, ef : \mathcal{E}X, j : X \subseteq \mu F, n : j \in \mathcal{N}(mef, map_{\mu F}), A$ and $t : FXA$. We have to define $In ef j n t : \mu FA$. Its first component of type μ^+FA is given by $In^+ ef map_{\mu^+F} MIt^+ j' n' t$ with $j' : X \subseteq \mu^+F$ defined by typewise composing the first projection out of μF with j , and n' its canonical naturality proof that depends on n and the fact that the first projection of $map_{\mu F} f r$ is defined to be map_{μ^+F} , applied to f and the first projection of r . The second component, i. e., the proof part, again follows directly from the introduction rule for chk_{μ^+F} since, by the very definition of μF , we have $\forall A \forall t^{XA}. chk_{\mu^+F}(j'_A t)$.¹⁷ Even with respect to convertibility, this construction fulfills the required equation for $map_{\mu F}$.

The definition for MIt is easier than for $map_{\mu F}$. Just define, given the step term $s : \forall X^{\kappa_1}. X \subseteq G \rightarrow FX \subseteq G$ and $r : \mu FA$, the term $MIt s r : GA$ as MIt^+ , applied to s and the first projection of r . The desired equality for MIt holds even as convertibility since composition is associative also in this sense. (The η -rule for MIt in the specification of $LNMI$ is trivially fulfilled by defining MIt as a lambda-abstraction.)

For the induction principle $\mu FInd$, we currently need proof-irrelevance, for two purposes:

¹⁷ The forced η -expansions can just be achieved by converting the goal to the expanded form.

- A simple consequence is the principle of irrelevance of the proof in elements of type $\mu F A$: If the first projections of r_1 and r_2 of type $\mu F A$ are equal, then $r_1 = r_2$ (hence, the first projection is injective). This could possibly be remedied by changes to CIC that affect the convertibility relation for strong sums (Werner, 2006).
- We need that any two proofs of naturality for the same parameters are equal. The author does not see yet how this could be reduced to the specific instance of proof-irrelevance where only all proofs of the same *equation* are identified (up to propositional equality). The problem here is that naturality is a universally quantified equation, and equational reasoning usually does not reach under binders in intensional type theory, as discussed in Section 2.

Proof-irrelevance is only about propositional equality of proofs of propositions and so does not degenerate the computational world (that is based on definitional equality) inside sort *Set*.

The following proof has no counterpart in Capretta's work. It seems that it profits from our very special situation, while Capretta intended to give a general method for simultaneous inductive-recursive definitions.

In order to prove $\mu F Ind$, assume the predicate P , the inductive step s of type

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^X \subseteq \mu F \forall n^{j \in \mathcal{N}(mef, map_{\mu F})}. (\forall A \forall x^{XA}. P_A(j_A x)) \rightarrow \forall A \forall t^{FXA}. P_A(In\ ef\ j\ n\ t)$$

and $A : Set$, $r : \mu F A$. We have to show $P_A r$. The term r decomposes into a term $r' : \mu^+ F A$ and a proof $p : chk_{\mu^+ F} r'$. We do induction on p . We write *cons* for the opposite operation of this decomposition, hence

$$cons : \forall A \forall r' : \mu^+ F A. chk_{\mu^+ F} r' \rightarrow \mu F A .$$

Thus, we want to show $P_A (cons\ r'\ p)$ by induction on p . As such, this is not covered by the given induction principle for $chk_{\mu^+ F}$. But there is also a more dependent version that can be generated by Coq ("induction scheme for sort *Prop*" that also takes into account the proofs of $chk_{\mu^+ F} r'$): Given a predicate

$$P' : \forall A \forall r' : \mu^+ F A. chk_{\mu^+ F} r' \rightarrow Prop,$$

it holds universally in the usual sense, i. e., $\forall A \forall r' : \mu^+ F A \forall p : chk_{\mu^+ F} r'. P'_A r' p$, if

$$\begin{aligned} \forall G^{\kappa_1} \forall ef^{\mathcal{E}G} \forall j^G \subseteq \mu^+ F \forall n^{j \in \mathcal{N}(mef, map_{\mu^+ F})} \forall k : (\forall A \forall t^{GA}. chk_{\mu^+ F} (j_A t)). \\ (\forall A \forall t^{GA}. P'_A(j_A t)(k_A t)) \rightarrow \forall A \forall t^{FGA}. \\ P'_A(In^+ ef\ map_{\mu^+ F} MI^+ (\lambda A \lambda t^{GA}. j_A t) nt) (Inchk\ ef\ j\ n\ k\ t) . \end{aligned}$$

For our proof, take $P'_A r' p := P_A (cons\ r'\ p)$. Assume G, ef, j, n, k according to this induction principle. Define $j' := \lambda A \lambda t^{GA}. cons(j_A t)(k_A t)$ of type $G \subseteq \mu F$. Under the assumptions $H : \forall A \forall t^{GA}. P_A(j'_A t)$, $A : Set$ and $t : FGA$, we have to prove

$$P_A(cons(In^+ ef\ map_{\mu^+ F} MI^+ (\lambda A \lambda t^{GA}. j_A t) nt) (Inchk\ ef\ j\ n\ k\ t)) .$$

First show $j' \in \mathcal{N}(mef, map_{\mu F})$. For this, one has to remove the outer quantifiers and then use proof-irrelevance in showing the equation only for the first projections. But this follows by a short calculation from our naturality proof n . Let n_1 be this proof of $j' \in$

$\mathcal{N}(mef, map_{\mu F})$. We deduce $P_A(Inef\ j'\ n_1\ t)$ from the general assumption s (the induction step of $\mu FInd$) and our assumption H .

It also holds that

$$In^+ ef\ map_{\mu+F} MIt^+ (\lambda A \lambda t : GA.\ j_A\ t)\ nt$$

is equal to the first projection of $Inef\ j'\ n_1\ t$. This is a simple calculation for the “j argument”, and we identify all naturality proofs in our system. Since we identify elements of μFA with the same first component, the two arguments of P_A in this proof development are equal, hence we may pass from the validity of the second such statement to that of the first one. Again, all the details can be found in the Coq development (Matthes, 2008). \square

6 Example: Explicit Flattening

In the following, we will illustrate the use of $LNMI$ with the example of a representation by nested datatypes of untyped lambda-terms. The original form (for credits, see the previously cited article by Abel et al) can be treated directly in Coq since version 8.1, which is the current one. Only the extension by an explicit flattening rule (again, see that article for more information) goes beyond direct representability in Coq.

In Coq with predicative Set , one can now declare $Lam : \kappa_1$ as an inductive family just by giving the types of its constructors (in a context $A : Set$):

$$\begin{aligned} var & : A \rightarrow Lam\ A , \\ app & : Lam\ A \rightarrow Lam\ A \rightarrow Lam\ A , \\ abs & : Lam(option\ A) \rightarrow Lam\ A . \end{aligned}$$

Then, $Lam\ A$ represents the untyped lambda-terms, where the variable names are taken from the type A . Lambda-abstraction is represented by abs , thus the name of the bound variable is just taken to be the additional element in $option\ A$. We insist on the freedom in the type A that can even be $Lam\ A'$ for some A' . A full formalization of pure type systems in Coq based on a restriction of the admissible types A to initial segments of the natural numbers has been obtained by Adams (2006).¹⁸ We may mostly follow his development for the definition of substitution

$$subst : \forall A, B. (A \rightarrow Lam\ B) \rightarrow Lam\ A \rightarrow Lam\ B ,$$

where for a *substitution rule* $f : A \rightarrow Lam\ B$, the term $subst\ f\ t : Lam\ B$ is the result of substituting every variable $a : A$ in the term representation $t : Lam\ A$ by the term $f\ a : Lam\ B$. The mapping function $lam : mon\ Lam$ does just variable renaming. It is easy to establish extensionality and functoriality of lam and extensionality of $subst$ in its argument f . One may even prove that $subst\ f\ t$ only depends on the values $f\ a$ for the a 's that freely occur in t (a notion to be defined inductively). The most interesting properties of $subst$ are the following:

$$\forall A, B, C \forall f^A \rightarrow B \forall g^B \rightarrow Lam\ C \forall t^{Lam\ A} . subst\ g\ (lam\ f\ t) = subst\ (g \circ f)\ t ,$$

¹⁸ Other formalizations were earlier (Altenkirch & Reus, 1999; McBride, 1999) and the point of this example is the essential use of nested Lam that is not considered elsewhere.

An induction principle for nested datatypes in intensional type theory 25

$$\forall A, B, C \forall f^{A \rightarrow Lam B} \forall g^{B \rightarrow C} \forall t^{Lam A}. lam\ g\ (subst\ f\ t) = subst\ (lam\ g \circ f)\ t .$$

An instance of the first property that goes well beyond Adams' work is as follows:

$$\forall A, B \forall f^{A \rightarrow Lam B} \forall t^{Lam A}. subst\ (\lambda x^{Lam B}. x)\ (lam\ f\ t) = subst\ f\ t .$$

Here, we used extensionality of *subst* in order to get from $(\lambda x.x) \circ f$ to f as argument to *subst*. Note that the term *lam f t* has type $Lam(Lam B)$. The function $subst(\lambda x.x) : Lam(Lam B) \rightarrow Lam B$ “flattens” this term in that it integrates the lambda-terms that constitute its free variable occurrences into the term itself. Note that this equation is just standard category-theoretic knowledge about the equivalence of monad representations with monad multiplication (here, the flattening operation) and binding (here, the substitution operation) and that this viewpoint has already been taken for the representation of untyped lambda-calculus by Bellegarde & Hook (1994).

With the above properties, one can easily establish the three monad laws for *subst*. We can also show naturality of *subst* in the following extended sense:

$$\forall X^{\kappa_1} \forall m^{mon X} \forall j^{X \subseteq Lam}. j \in \mathcal{N}(m, lam) \rightarrow (\lambda A. subst\ j_A) \in \mathcal{N}(lam \star m, lam) ,$$

where $lam \star m$ is the canonical map term for $Lam \circ X$ that is implicit in the last case of Lemma 1 on page 13. This naturality lemma will be needed below.

While all of the described development goes smoothly in the current Coq version — for the details, see the Coq scripts (Matthes, 2008)—we now have to make use of *LNMI*. We extend the untyped lambda-calculus by an explicit notion of flattening, i. e., a term former to indicate flattening that is not carried out, just like explicit substitution. The corresponding datatype functor is thus

$$F := \lambda X \lambda A. A + XA \times XA + X(option A) + X(XA) ,$$

while the one for *Lam* would be the same F , with the last summand removed. We assume that $+$ associates to the left. From Lemma 2, one easily produces the corresponding Fp^e .

There are two options for the use of *LNMI*: the axiomatic one that only takes the specification in Section 3 (in Coq, this is done by putting the whole development into a “functor” that depends on an argument module of a module type that contains the description of *LNMI*), and the implementation according to Section 5. In the first case, predicative *Set* suffices, but the equations for the behaviour of $map_{\mu F}$ and *MIt* can only be used as propositional equality. In the second case, one needs impredicative *Set*, and Coq applies the equations implicitly when evaluating expressions. However, the implementation details are not encapsulated and might be exploited in the development. The Coq scripts that are available for this article illustrate both approaches.

We will call *LamE* the fixed point μF for the F above, and *lamE* the map term $map_{\mu F}$. The interesting new canonical datatype constructor (*InCan* codes together four datatype constructors),

$$flat : \forall A. LamE(LamEA) \rightarrow LamEA ,$$

is obtained by composing *InCan* with the right injection *inr*.

We define a function $eval : LamE \subseteq Lam$ that evaluates all the explicit flattenings and thus yields the representation of a usual lambda-term by

$$eval := MIt(\lambda X^{\kappa_1} \lambda it^{X \subseteq Lam} \lambda A \lambda t^{FXA}. match\ t\ with\ \dots | inr\ e \mapsto subst\ it_A\ (it_{XA}\ e)) .$$

Note that $e : X(XA)$, $it_{XA} e : Lam(XA)$, $it_A : XA \rightarrow LamA$ and consequently $subst\ it_A (it_{XA} e) : LamA$. Theorem 3 immediately gives

$$eval_A(flat_A e) \simeq subst\ eval_A (eval_{LamEA} e) .$$

This is a new algorithm and substantially different from earlier work (Abel *et al.*, 2005). Intuitively, it takes the argument e of type $LamE(LamEA)$ and first ignores the complex structure of the variables when evaluating the explicit flattenings, arriving at a term of type $Lam(LamEA)$. Then, each of the freely occurring variables, that are in fact lambda-terms with explicit flattenings, has to be substituted by the corresponding evaluated lambda-term.

Theorem 1 allows to prove that $eval$ is a natural transformation from $lamE$ to lam . The case that pertains to the $flat$ constructor is a simple consequence of naturality of $subst$, defined above (together with extensionality of lam and mef). As a corollary of naturality, using the “instance of the first property” of $subst$ above, we get

$$eval_A(flat_A e) = subst(\lambda x^{LamA}.x)(eval_{LamA}(lamE\ eval_A e)) .$$

The algorithmic idea of the right-hand side is as follows: Take the argument e of type $LamE(LamEA)$ and first concentrate on the terms-as-variables in $LamEA$: Rename them via $lamE$, according to the function $eval_A : LamEA \rightarrow LamA$. This yields a term of type $LamE(LamA)$. In a second time, evaluate the “outer structure”, ignoring the complex structure of the variables. Now, flatten out this term of type $Lam(LamA)$, see our discussion above.

We might wonder whether $eval$ could have been defined so that the previous propositional equality were even forced to be convertibility. One would first try to replace the term $subst\ it_A (it_{XA} e)$ in the definition of $eval$ by

$$subst(\lambda x^{LamA}.x)(it_{LamA}(lamE\ it_A e)) ,$$

but this would only type-check if $lamE$ were replaced by a map term $m : mon\ X$ for X , but no such m is available in our version of the Mandler iterator. Currently, one would need to resort to sized nested datatypes (Abel, 2006) for such a program, but there do not yet exist induction principles for reasoning on programs with sized nested datatypes.

As a less immediate application of naturality of $eval$, we can analyse half-explicit substitution

$$esubst : \forall A, B. (A \rightarrow LamEB) \rightarrow LamEA \rightarrow LamEB$$

defined by $esubst\ f\ t := flat(lamE\ f\ t)$, where only the renaming is done, and the flattening is left explicit. Trivially, one can embed Lam into $LamE$ by a function emb . The question is then whether $subst\ f\ t$ with $f : A \rightarrow LamB$ and $t : LamA$ can be calculated by evaluating

$$eval_B(esubst(emb_B \circ f)(emb_A t)) ,$$

and this can be answered in the affirmative (for propositional equality) by using naturality of $eval$. One also needs extensionality and the first property of $subst$ and that $eval$ is a left-inverse of emb , but those are all proven by induction on Lam , thus not taking profit from $LNMI$ in this Coq development (Matthes, 2008).

7 Conclusions

It is now possible to combine the following benefits:

- *termination* of all functions following the recursion schemes
- recursion schemes are *type-based* and not syntax-driven
- genericity: no specific shape of the datatype functors required
- no continuity properties required
- includes *truly nested* datatypes
- *categorical laws* for program verification
- program execution *within the convertibility relation of Coq*

In practice, one often uses refined forms of iteration that are also known under the names of efficient folds (Hinze, 2000; Martin *et al.*, 2004). More general forms in the spirit of Mendler’s style can be studied, e. g., the system $MI_{=}^{\omega}$ (Abel *et al.*, 2005). Its iterator can be expressed by $MI_{}$ of this article, but only at the expense of some right Kan extension as target type constructor G . Although our Theorem 1 would apply, its application condition would speak about equality of functions, and that is usually not provable. With some more refined notions of extensionality and more careful use of quantification, it is nevertheless possible to prove a theorem for $MI_{}$ that will yield a naturality property for $MI_{=}$ that precisely captures the map fusion law. This can even be extended to treat $GMI_{}$, a more liberal form of $MI_{=}$, also introduced in that article.

Certainly, more and more difficult examples have to be verified. It does not seem possible to extend the construction with the inductive-recursive definition from iteration to *primitive recursion* or just to add an inversion operation. A further goal would be reasoning principles for *conventional style* without non-canonical elements that can treat truly nested datatypes.

Finally, it should be mentioned that the present work does not restrict the system to one single nested datatype or only the introduction of one such datatype after the other. All the constructions are fully parametric so that arbitrary interleaving of such families is admissible, although the author is not aware of natural examples where a nested datatype sits inside the definition of another “real” nested datatype in the sense of different family indices in the recursive equation.

Acknowledgements: I am grateful to Chantal Berline who took the time to discuss possible analogies of the presence of non-canonical elements in $LNMI_{}$ and non-standard models. I am also very grateful to the anonymous referees that forced me to deepen my understanding of what $LNMI_{}$ can do with canonical elements, and for all their detailed and valuable comments.

References

- Abel, Andreas. (2006). *A polymorphic lambda-calculus with sized higher-order types*. Doktorarbeit (PhD thesis), LMU München.
- Abel, Andreas, Matthes, Ralph, & Uustalu, Tarmo. (2005). Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, **333**(1–2), 3–66.
- Adams, Robin. (2006). Formalized metatheory with terms represented by an indexed family of types. *Pages 1–16 of: Filiâtre, Jean-Christophe, Paulin-Mohring, Christine, & Werner, Benjamin (eds),*

- Types for proofs and programs, TYPES 2004, international workshop, revised selected papers.* Lecture Notes in Computer Science, vol. 3839. Springer Verlag.
- Altenkirch, Thorsten. (1999). Extensional equality in intensional type theory. *Pages 412–420 of: 14th annual IEEE symposium on Logic in Computer Science (LICS 1999)*. IEEE Computer Society.
- Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda terms using generalized inductive types. *Pages 453–468 of: Flum, Jörg, & Rodríguez-Artalejo, Mario (eds), Computer Science Logic, 13th international workshop, CSL '99, proceedings*. Lecture Notes in Computer Science, vol. 1683. Springer Verlag.
- Bellegarde, Françoise, & Hook, James. (1994). Substitution: A formal methods case study using monads and transformations. *Science of computer programming*, **23**, 287–311.
- Bertot, Yves, & Castéran, Pierre. (2004). *Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions*. Texts in Theoretical Computer Science. Springer Verlag.
- Bird, Richard, & de Moor, Oege. (1997). *Algebra of programming*. International Series in Computer Science, vol. 100. Prentice Hall.
- Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of: Jeuring, Johan (ed), Mathematics of Program Construction, MPC'98, proceedings*. Lecture Notes in Computer Science, vol. 1422. Springer Verlag.
- Bird, Richard, & Paterson, Ross. (1999a). Generalised folds for nested datatypes. *Formal Aspects of Computing*, **11**(2), 200–222.
- Bird, Richard S., & Paterson, Ross. (1999b). De Bruijn notation as a nested datatype. *Journal of Functional Programming*, **9**(1), 77–91.
- Capretta, Venanzio. (March 2004). *A polymorphic representation of induction-recursion*. Note of 9 pages available on the author's web page (a second 15 pages version of May 2005 has been seen by the present author).
- Coq Development Team. (2006). *The Coq Proof Assistant Reference Manual Version 8.1*. Project LogiCal, INRIA. System available at coq.inria.fr.
- Coquand, Thierry, & Paulin, Christine. (1990). Inductively defined types. *Pages 50–66 of: Martin-Löf, Per, & Mints, Grigori (eds), COLOG-88, international conference on Computer Logic*. Lecture Notes in Computer Science, vol. 417. Springer Verlag.
- Dybjer, Peter. (2000). A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, **65**(2), 525–549.
- Dybjer, Peter, & Setzer, Anton. (2003). Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, **124**, 1–47.
- Girard, Jean-Yves. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII.
- Hinze, Ralf. (2000). Efficient generalized folds. Jeuring, Johan (ed), *Proceedings of the second workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal*.
- Hofmann, Martin. (1995). *Extensional concepts in intensional type theory*. Ph.D. thesis, University of Edinburgh. Available as report ECS-LFCS-95-327.
- Martin, Clare, Gibbons, Jeremy, & Bayley, Ian. (2004). Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, **16**(1), 19–35.
- Matthes, Ralph. (1998). *Extensions of system F by iteration and primitive recursion on monotone inductive types*. Doktorarbeit (PhD thesis), University of Munich. Available via the homepage <http://www.irit.fr/Ralph.Matthes/>.
- Matthes, Ralph. (2001). Monotone inductive and coinductive constructors of rank 2. *Pages 600–614 of: Fribourg, Laurent (ed), Proceedings of CSL 2001*. Lecture Notes in Computer Science, vol. 2142. Springer Verlag.

- Matthes, Ralph. (2006). Verification of programs on truly nested datatypes in intensional type theory. McBride, Conor, & Uustalu, Tarmo (eds), *Workshop on Mathematically Structured Functional Programming (MSFP '06)*, Kuressaare, Estonia, July 2, 2006. Electronic Workshops in Computing (eWiC), ISSN 1477-9358. British Computer Society. 12 pages, freely available through <http://www.bcs.org/server.php?show=nav.7927>.
- Matthes, Ralph. (January 2008). *Coq development for "An induction principle for nested datatypes in intensional type theory"*. <http://www.irit.fr/~Ralph.Matthes/Coq/InductionNested/>.
- McBride, Conor. (1999). *Independently typed functional programs and their proofs*. Ph.D. thesis, University of Edinburgh.
- Mendler, Nax P. (1987). Recursive types and type constraints in second-order lambda calculus. *Pages 30–36 of: Proceedings of the second annual IEEE symposium on Logic in Computer Science, Ithaca, N.Y.* IEEE Computer Society Press.
- Oury, Nicolas. (2005). Extensionality in the calculus of constructions. *Pages 278–293 of: Hurd, Joe, & Melham, Thomas F. (eds), Theorem Proving in Higher Order Logics. proceedings.* Lecture Notes in Computer Science, vol. 3603. Springer Verlag.
- Uustalu, Tarmo. (1998). *Natural deduction for intuitionistic least and greatest fixedpoint logics, with an application to program construction*. Ph.D. thesis, Royal Institute of Technology, Kista, Sweden.
- Uustalu, Tarmo, & Vene, Varmo. (May 1997). A cube of proof systems for the intuitionistic predicate μ -, ν -logic. *Pages 237–246 of: Haverdaen, Magne, & Owe, Olaf (eds), Selected papers of the 8th Nordic workshop on Programming Theory (NWPT '96)*. Research Reports, Department of Informatics, University of Oslo, vol. 248.
- Uustalu, Tarmo, & Vene, Varmo. (2002). Least and greatest fixed points in intuitionistic natural deduction. *Theoretical Computer Science*, **272**, 315–339.
- Werner, Benjamin. (2006). On the strength of proof-irrelevant type theories. *Pages 604–618 of: Furbach, Ulrich, & Shankar, Natarajan (eds), Automated reasoning, third international joint conference, IJCAR 2006*. Lecture Notes in Computer Science, vol. 4130. Springer Verlag.

