Nested datatypes with generalized Mendler iteration: map fusion and the example of the representation of untyped lambda calculus with explicit flattening

Ralph Matthes

Institut de Recherche en Informatique de Toulouse (IRIT)
C. N. R. S. et Université Paul Sabatier (Toulouse III)
118 route de Narbonne, F-31062 Toulouse Cedex 9

Abstract. Nested datatypes are families of datatypes that are indexed over all types such that the constructors may relate different family members. 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, there is no direct support by theorem provers to guarantee termination of functions that traverse these data structures. 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. In subsequent work (accepted for the Journal of Functional Programming), the author introduced an induction principle for the verification of programs obtained from Mendler-style iteration of rank 2, which is one of those schemes, and justified it in the Calculus of Inductive Constructions through an implementation in the theorem prover Coq.
The new contribution is an extension of this work to generalized Mendler iteration (introduced in Abel et al, cited above), leading to a map fusion theorem for the obtained iterative functions. The results and their implementation in Coq are used for a case study on a representation of untyped lambda calculus with explicit flattening. Substitution is proven to fulfill two of the three monad laws, the third only for "hereditarily canonical" terms, but this is rectified by a relativisation of the whole construction to those terms.

1 Introduction

Nested datatypes [1] are families of datatypes that are indexed over all types and where different family members are related by the datatype constructors. Let κ_0 stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. Then, let κ_1 be the kind of type transformations, hence $\kappa_1 := \kappa_0 \to \kappa_0$. A typical example would be *List* of kind κ_1, where *List A* is the type of finite lists with elements from type A. But *List*

is not a nested datatype since the recursive equation for *List*, i. e., $List\,A = 1 + A \times List\,A$, does not relate lists with different indices. A simple example of a nested datatype where an invariant is guaranteed through its definition are the powerlists [2] (or perfectly balanced, binary leaf trees [3]), with recursive equation $PList\,A = A + PList(A \times A)$, where the type $PList\,A$ represents trees of 2^n elements of A with some $n \geq 0$ (that is not fixed) since, throughout this article, we will only consider the least solutions to these equations. The basic example where variable binding is represented through a nested datatype is a typeful de Bruijn representation of untyped lambda calculus, following ideas of [4–6]. The lambda terms with free variables taken from A are given by $Lam\,A$, with recursive equation $Lam\,A = A + Lam\,A \times Lam\,A + Lam(option\,A)$. The first summand gives the variables, the second represents application of lambda terms and the interesting third summand stands for lambda abstraction: An element of $Lam(option\,A)$ (where $option\,A$ is the type that has exactly one more element than A, namely *None*, while the injection of A into $option\,A$ is called *Some*) is seen as an element of $Lam\,A$ through lambda abstraction of that designated extra variable that need not occur freely in the body of the abstraction.

Programming with nested datatypes is possible in the functional programming language Haskell, but this article is concerned with frameworks that guarantee termination of all expressible programs, such as the Coq theorem prover [7] that is based on the Calculus of Inductive Constructions (CIC), presented with details in [8], which only recently (since version 8.1 of Coq) evolved towards a direct support for many nested datatypes that occur in practice, e. g., *PList* and *Lam* are fully supported with recursion and induction principles. Although Coq is officially called the "Coq proof assistant", it is already in itself[1] a functional programming language. This is certainly not surprising since it is based on an extension of polymorphic lambda calculus (system F^ω), although the default type-theoretic system of Coq since version 8.0 is "pCIC", namely the Predicative Calculus of (Co)Inductive Constructions. System F^ω is also the framework of the article with Abel and Uustalu [10] that presents a variety of terminating iteration principles on nested datatypes for a notion of nested datatypes that also allows true nesting, which is not supported by the aforementioned recent extension of CIC. A nested datatype will be called "truly nested" (non-linear [11]) if the intuitive recursive equation for the inductive family has at least one summand with a nested call to the family name, i. e., the family name appears somewhere inside the type argument of a family name occurrence of that summand. Our example throughout this article is lambda terms with explicit flattening [12], with the recursive equation

$$LamE\,A = A + LamE\,A \times LamE\,A + LamE(option\,A) + LamE(LamE\,A)\ .$$

The last summand qualifies *LamE* as truly nested datatype: $LamE\,A$ is the type argument to *LamE*.

[1] Not to speak of the program extraction facility of Coq that allows to obtain programs in OCaml, Scheme and Haskell from Coq developments in an automatic way [9].

Even without termination guarantees, the algebra of programming [13] 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, laws have been important from the beginning [1]. However, no reasoning principles, in particular no induction principles, were studied in [10] on terminating iteration (and coiteration) principles. Newer work by the author [14] integrates rank-2 Mendler iteration into CIC and also justifies an induction principle for them. This is embodied in the system *LNMIt*, the "logic for natural Mendler-style iteration", defined in Section 3.1. This system integrates termination guarantees and calculational verification in one formalism and would also allow dependently-typed programming on top of nested datatypes. Just to recall, termination is also of practical concern with dependent types, namely that type-checking should be decidable: If types depend on object terms, object terms have to be evaluated in order to verify types, as expressed in the convertibility rule. Note, however, that this only concerns evaluation within the definitional equality (i. e., convertibility), henceforth denoted by $\simeq$. Except from the above intuitive recursive equations, $=$ will denote propositional equality throughout: this is the equality type that requires proof and that satisfies the Leibniz principle, i. e., that validity of propositions is not affected by replacing terms by equal (w. r. t. $=$) terms.

The present article is concerned with an extension of *LNMIt* to a system *LNGMIt* that has generalized Mendler-iteration *GMIt*, introduced in [10], in addition to plain Mendler-iteration that is provided by *LNMIt*. Generalized Mendler-iteration is a scheme encompassing generalized folds [11, 3, 15]. In particular, the efficient folds of [15] are demonstrated to be instances of *GMIt* in [10], and the relation to the gfolds of [11] is discussed there. Perhaps surprisingly, *GMIt* could be explained within F^ω through *MIt*. In a sense, this all boils down to the use of a syntactic form of right Kan extensions as the target constructor G^{κ_1} of the polymorphic iterative functions of type $\forall A^{\kappa_0}.\, \mu F A \to G A$, where μF denotes the nested datatype [10, Section 4.3]. (These Kan extension ideas are displayed in more detail using Haskell in [16], but only in a setting that excludes truly nested datatypes although the type system of current Haskell implementations has no problems with them.)

The main theorem of [14] is trivially carried over to the present setting, i. e., just by the Kan extension trick, the justification of *LNMIt* within CIC with impredicative universe $Set =: \kappa_0$ and propositional proof irrelevance is carried over to *LNGMIt*. Impredicativity of κ_0 is needed here since syntactic Kan extensions use impredicative means for κ_0 in order to stay within κ_1. However, *LNMIt* and *LNGMIt* are formulated as extensions of pCIC with its predicative *Set* as κ_0.

The functions that are defined by a direct application of *GMIt* are uniquely determined (up to pointwise propositional equality) by their recursive equation, under a reasonable extensionality assumption. It is shown when these functions are themselves extensional and when they are "natural", and what natural has to mean for them.

By way of the example of lambda terms with explicit flattening—the truly nested datatype *LamE*—the merits of the general theorems about *LNGMIt* will be studied, mainly by a representation of parallel substitution on *LamE* using *GMIt* and a proof of the monad laws for it. One of the laws fails in general, but it can be established for the hereditarily canonical terms. Their inductive definition (using the inductive definition mechanism of pCIC) refers to the notion of free variables that is obtained from the scheme *MIt*. The whole development for *LamE* can be interpreted within the hereditarily canonical terms, and for those, parallel substitution is shown to be a monad.

All the concepts and results have been formalised in the Coq system, also using module functors having as parameter a module type with the abstract specification of *LNGMIt*, in order to separate the impredicative justification from the predicative formulation and its general consequences that do not depend on an implementation/justification. The Coq code is available [17] and is based on [18].

The following section 2.1 introduces to the Mendler style of obtaining terminating recursive programs and develops the notions of free variables and renaming in the case study. It also discusses extensionality and naturality. Section 2.2 presents *GMIt* and defines a representation of substitution for the case study, leading to a list of properties one would like to prove about it. In Section 3.1, the already existing system *LNMIt* with the logic for *MIt* is properly defined, while Section 3.2 defines the new extension *LNGMIt* as a logic for *GMIt* and proves some general results. The question of naturality for functions that are defined through *GMIt* is addressed in Section 4. General results about proving naturality are presented, one of them is map fusion. Section 5 problematizes the results obtained so far in the case study. Hereditary canonicity is the key notion that allows to pursue that case study. Section 6 concludes.

Acknowledgements: To Andreas Abel for all the joint work in this field and some of his LaTeX macros and the figure I reused from earlier joint papers, and to the referees for their helpful advice that I could only partially integrate in view of the length of this article. In an early stage of the present results, I have benefitted from support by the European Union FP6-2002-IST-C Coordination Action 510996 “Types for Proofs and Programs”.

2 Mendler-style Iteration

Mendler-style iteration schemes, originally proposed for positive inductive types [19], come with a termination guarantee, and termination is not based on syntactic criteria (that all recursive calls are done with “smaller” arguments) but just on types (called “type-based termination” in [20]).

2.1 Plain Mendler-style Iteration *MIt*

In order to fit the above intuitive definition of *LamE* into the setting of Mendler-style iteration, the notion of rank-2 functor is needed. Their kind is defined as

$\kappa_2 := \kappa_1 \to \kappa_1$. Any constructor F of kind κ_2 qualifies as rank-2 functor for the moment, and $\mu F : \kappa_1$ denotes the generated family of datatypes. For our example, set

$$LamEF := \lambda X^{\kappa_1} \lambda A^{\kappa_0}.\, A + XA \times XA + X(option\, A) + X(XA)$$

and $LamE := \mu\, LamEF$. In general, there is just one datatype constructor for μF, namely $in : F(\mu F) \subseteq \mu F$, using $X \subseteq Y := \forall A^{\kappa_0}.\, XA \to YA$ for any $X, Y : \kappa_1$ as abbreviation for the respective polymorphic function space. For $LamE$, more clarity comes from the four derived datatype constructors

$$\begin{aligned} &varE : \forall A^{\kappa_0}.\, A \to LamE\, A \ , \\ &appE : \forall A^{\kappa_0}.\, LamE\, A \to LamE\, A \to LamE\, A \ , \\ &absE : \forall A^{\kappa_0}.\, LamE(option\, A) \to LamE\, A \ , \\ &\mathit{flatE} : \forall A^{\kappa_0}.\, LamE(LamE\, A) \to LamE\, A \ , \end{aligned}$$

where, for example, *flatE* is defined as $\lambda A^{\kappa_0} \lambda e^{LamE(LamE\, A)}.\, in\, A\, (inr\, e)$, with right injection *inr* (here, we assume that + associates to the left), and the other datatype constructors are defined by the respective sequence of injections (see [12] or [10, Example 8.1]).[2] From the explanations of *Lam* in the introduction, it is already clear that *varE*, *appE* and *absE* represent the construction of terms from variable names, application and lambda abstraction in untyped lambda calculus (their representation via a nested datatype has been introduced by [5, 6]).

A simple example can be given as follows: Consider the untyped lambda term $\lambda z.\, z\, x_1$ with the only free variable x_1. For future extensibility, think of the allowed set of variable names as $option\, A$ with type variable A. The designated element *None* of $option\, A$ shall be the name for variable x_1. $\lambda z.\, z\, x_1$ is represented by

$$absE(appE\, (varE\, None)\, (varE\, (Some\, None))) \ ,$$

with *None* and *Some None* of type $option(option\, A)$, hence with the shift that is characteristic of de Bruijn representation. Obviously, the representation is of type $\forall A^{\kappa_0}.\, LamE\, (option\, A)$, and it could have been done in a similar way with *Lam* instead of *LamE*.

In [4], a lambda-calculus interpretation of monad multiplication of *Lam* is given that has the type of *flatE* (with *LamE* replaced by *Lam*), but *here*, this is just a formal (non-executed) form of an integration of the lambda terms that constitute its free variable occurrences into the term itself. We call *flatE explicit flattening*. It does not do anything to the term but is another means of constructing terms.

For an example, consider $t := \lambda y.\, y\, \{\lambda z.\, z\, x_1\}\, \{x_2\}$, where the braces shall indicate that the term inside is considered as the *name of a variable*. If these terms-as-variables were integrated into the term, i. e., if t were "flattened", one would obtain $\lambda y.\, y\, (\lambda z.\, z\, x_1)\, x_2$. This is a trivial operation in this example. In

[2] In Haskell 98, one would define *LamE* through the types of its datatype constructors whose names would be fixed already in the definition of *LamE*.

[14], it is recalled that parallel substitution can be decomposed into renaming, followed by flattening. Under the assumption that substitution is a non-trivial operation, flattening and renaming cannot both be considered trivial. Through the explicit form of flattening, its contribution to the complexity of substitution can be studied in detail.

We want to represent t as term of type $\forall A^{\kappa_0}.\, \mathit{LamE}\,(\mathit{option}(\mathit{option}A))$, in order to accommodate the two free variables x_1, x_2. We instantiate the representation above for $\lambda z.\, z\, x_1$ by $\mathit{option}\, A$ in place of A and get a representation as term $t_1 : \mathit{LamE}\,(\mathit{option}(\mathit{option}A))$. x_2 is represented by

$$t_2 := \mathit{varE}\,(\mathit{Some}\,\mathit{None}) : \mathit{LamE}\,(\mathit{option}(\mathit{option}A)) \ .$$

Now, t shall be represented as the term

$$\mathit{flatE}(\mathit{absE}\, t_3) : \mathit{LamE}\,(\mathit{option}(\mathit{option}A)) \ ,$$

hence with $t_3 : \mathit{LamE}\,(\mathit{option}\,(\mathit{LamE}\,(\mathit{option}(\mathit{option}A))))$, defined as

$$t_3 := \mathit{appE}\,\Big(\mathit{appE}\,(\mathit{varE}\,\mathit{None})\,(\mathit{varE}\,(\mathit{Some}\, t_1))\Big)\,(\mathit{varE}\,(\mathit{Some}\, t_2)) \ ,$$

that stands for $y\,\{\lambda z.\, z\, x_1\}\,\{x_2\}$. Finally, we can quantify over the type A.

Mendler iteration of rank 2 [10] can be described as follows: There is a constant

$$\mathit{MIt} : \forall G^{\kappa_1}.\, (\forall X^{\kappa_1}.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G$$

and the iteration rule

$$\mathit{MIt}\, G\, s\, A\,(\mathit{in}\, A\, t) \simeq s\,(\mu F)\,(\mathit{MIt}\, G\, s)\, A\, t \ .$$

In a properly typed left-hand side, t has type $F(\mu F)A$ and s is of type

$$\forall X^{\kappa_1}.\, X \subseteq G \to FX \subseteq G \ .$$

The term s 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.

Our first example of an iterative function on LamE is the function $\mathit{EFV} : \mathit{LamE} \subseteq \mathit{List}$ (EFV is a shorthand for $\mathit{LamEToFV}$) that gives the list of the names of the free variables (with repetitions in case of multiple occurrences). We want to have the following definitional equations that describe the recursive behaviour (we mostly write type arguments as indices in the sequel):

$$\begin{array}{lll}
\mathit{EFV}_A\,(\mathit{varE}_A\, a) & \simeq & [a] \ , \\
\mathit{EFV}_A\,(\mathit{appE}_A\, t_1\, t_2) & \simeq & \mathit{EFV}_A\, t_1 + \mathit{EFV}_A\, t_2 \ , \\
\mathit{EFV}_A\,(\mathit{absE}_A\, r) & \simeq & \mathit{filterSome}_A\,(\mathit{EFV}_{\mathit{option}\,A}\, r) \ , \\
\mathit{EFV}_A\,(\mathit{flatE}_A\, e) & \simeq & \mathit{flatten}(\mathit{map}\, \mathit{EFV}_A\,(\mathit{EFV}_{\mathit{LamE}\,A}\, e)) \ .
\end{array}$$

Here, we denoted by $[a]$ the singleton list that only has a as element and by $+$ list concatenation. Moreover, $\mathit{filterSome} : \forall A^{\kappa_0}.\, \mathit{List}(\mathit{option}\, A) \to \mathit{List}\, A$ removes

all the occurrences of *None* from its argument and also removes the injection *Some* from A to $\mathit{option}\,A$ from the others. This is nothing but saying that the extra element *None* of $\mathit{option}\,A$ is the variable name that is considered bound in $\mathit{absE}_A\,r$, and that therefore all its occurrences have to be removed from the list of free variables. The set of free variables of $\mathit{flatE}_A\,e$ is the union of the sets of free variables of the free variables of e, which are still elements of $\mathit{LamE}\,A$. This is expressed by the usual mapping function

$$\mathit{map} : \forall A^{\kappa_0} \forall B^{\kappa_0}.\,(A \to B) \to \mathit{List}\,A \to \mathit{List}\,B$$

for lists and the operation $\mathit{flatten} : \forall A^{\kappa_0}.\,\mathit{List}(\mathit{List}\,A) \to \mathit{List}\,A$ that concatenates all the lists in its argument to a single list, and we did not mention the types with which the type arguments of *map* and *flatten* are instantiated.[3] We now argue that there is such a function EFV, by showing that it is directly definable as $\mathit{MIt}\,\mathit{List}\,s_{\mathit{EFV}}$ for some closed term

$$s_{\mathit{EFV}} : \forall X^{\kappa_1}.\,X \subseteq \mathit{List} \to \mathit{LamEF}\,X \subseteq \mathit{List}\ ,$$

and therefore, we have the termination guarantee (in [10], a definition of *MIt* within F^ω is given that respects the iteration rule even as reduction from left to right, hence this *is* iteration as is the iteration over the Church numerals of which this is still a generalization). Using an intuitive notion of pattern matching, we define

$$\begin{array}{rl}
s_{\mathit{EFV}} := \lambda X^{\kappa_1} \lambda it^{X \subseteq \mathit{List}} \lambda A^{\kappa_0} \lambda t^{\mathit{LamEF}\ X\ A}.\,\mathrm{match}\,t\,\mathrm{with} & \\
\quad \mid \mathit{inl}(\mathit{inl}(\mathit{inl}\,a^A)) & \mapsto [a] \\
\quad \mid \mathit{inl}(\mathit{inl}(\mathit{inr}(t_1^{XA}, t_2^{XA}))) & \mapsto \mathit{it}_A\,t_1 + \mathit{it}_A\,t_2 \\
\quad \mid \mathit{inl}(\mathit{inr}\,r^{X(\mathit{option}\,A)}) & \mapsto \mathit{filterSome}(\mathit{it}_{\mathit{option}\,A}\,r) \\
\quad \mid \mathit{inr}\,e^{X(XA)} & \mapsto \mathit{flatten}(\mathit{map}\,\mathit{it}_A\,(\mathit{it}_{XA}\,e))\ .
\end{array}$$

For $\mathit{EFV} := \mathit{MIt}\,\mathit{List}\,s_{\mathit{EFV}}$, the required equational specification is obviously satisfied (since the pattern-matching mechanism behaves properly with respect to definitional equality $\simeq$).[4]

The visible reason why Mendler's style can guarantee termination without any syntactic descent (in which way can the mapping over EFV_A be seen as "smaller"?) is the following: the recursive calls come in the form of uses of *it*, which does not have type $\mathit{LamEF} \subseteq \mathit{List}$ but just $X \subseteq \mathit{List}$, and the type arguments of the datatype constructors are replaced by variants that only mention X instead of *LamE*. So, the definitions have to be uniform in that type transformation variable X, but this is already sufficient to guarantee termination (for

[3] It would have been cleaner to use just one function instead, namely the function $\mathit{flat\_map} : \forall A^{\kappa_0} \forall B^{\kappa_0}.\,(A \to \mathit{List}\,B) \to \mathit{List}\,A \to \mathit{List}\,B$, where $\mathit{flat\_map}_{A,B}\,f\,\ell$ is the concatenation of all the B-lists $f\,a$ for the elements a of the A-list ℓ. Note that *flatten* is monad multiplication for the list monad and could also be made explicit by a truly nested datatype.

[4] In Haskell 98, our specification of EFV, together with its type, can be used as a definition, but no termination guarantee is obtained.

the rank-1 case of inductive *types*, this has been discovered in [21] by syntactic means and, independently, by the author with a semantic construction [22]).

A first interesting question about the results of $EFV_A\, t$ is how they behave with respect to renaming of variables. First, define for any type transformation $X : \kappa_1$ the type of its map term as (from now, omit the kind κ_0 from A and B)

$$mon\, X := \forall A \forall B.\, (A \to B) \to X\, A \to X\, B \ .$$

Clearly, $map : mon\, List$, but also renaming $lamE$ will have a type of this form, more precisely, $lamE : mon\, LamE$, and $lamE\, f\, t$ has to represent t after renaming every free variable occurrence a in t by $f\, a$. It would be possible to define $lamE$ by help of *GMIt* introduced in the next section, but it will automatically be available in the systems *LNMIt* and *LNGMIt* that will be described in Section 3. Therefore, we content ourselves in displaying its recursive behaviour (we omit the type arguments to $lamE$):

$$\begin{array}{lcl}
lamE\, f\, (varE_A\, a) & \simeq & varE_B(f\, a) \ , \\
lamE\, f\, (appE_A\, t_1\, t_2) & \simeq & appE_B\, (lamE\, f\, t_1)\, (lamE\, f\, t_2) \ , \\
lamE\, f\, (absE_A\, r) & \simeq & absE_B(lamE\, (option_map\, f)\, r) \ , \\
lamE\, f\, (flatE_A\, e) & \simeq & flatE_B\Big(lamE\, \big(\lambda t^{LamE\, A}.\, lamE\, (\lambda x^A.\, f\, x)\, t\big)\, e\Big) \ .
\end{array}$$

Here, in the second clause, yet another map term occurs, namely the canonical $option\_map : mon\, option$, so that $lamE$ is called with type arguments $option\, A$ and $option\, B$. In the final clause, the outer call to $lamE$ is with type arguments $LamE\, A$ and $LamE\, B$, while the inner one stays with A and B. The right-hand side in the last case is unpleasantly η-expanded, and one would have liked to see $flatE_B(lamE\, (lamE\, f)\, e)$ instead. However, these two terms are not definitionally equal.

For any $X : \kappa_1$ and map term $m : mon\, X$, define the following proposition

$$ext\, m := \forall A \forall B \forall f^{A \to B} \forall g^{A \to B}.\, (\forall a^A.\, f\, a = g\, a) \to \forall r^{X\, A}.\, m\, A\, B\, f\, r = m\, A\, B\, g\, r \ .$$

It expresses that m only depends on the extension of its functional argument, which will be called *extensionality* of m in the sequel. In intensional type theory such as CIC, it does not hold in general.[5] In *LNMIt* and *LNGMIt*, the canonical map term $map_{\mu F}$ that comes with μF is extensional. Hence, $lamE$ of our example will be extensional, and the right-hand side in the last case is propositionally equal to the simpler form considered above.

We can now state the "interesting question", mentioned before: Can one prove

$$\forall A \forall B \forall f^{A \to B} \forall t^{LamE\, A}.\, EFV_B(lamE\, f\, t) = map\, f\, (EFV_A\, t) \ ?$$

This is an instance of the question for polymorphic functions j of type $X \subseteq Y$ whether they behave propositionally as a natural transformation from (X, mX)

[5] There are deep studies [23–25] on a reconciliation of intensional type theory with extensionality for function spaces. However, we will stick with CIC.

to (Y, mY), given map functions $mX : mon\, X$ and $mY : mon\, Y$. Here, the pair (X, mX) is seen as a functor although no functor laws are required (for the moment). The proposition that defines j to be such a natural transformation is

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

The system *LNMIt*, described in Section 3.1, allows to answer the above question by showing $EFV \in \mathcal{N}(lamE,\, map)$. This is in contrast to pure functional programming, where, following [26], naturality is seen as free, namely as a specific instance of parametricity for parametric equality. In intensional type theory such as our *LNMIt* and *LNGMIt* (see Section 3.2), naturality has to be proven on a case by case basis.

By (plain) Mendler iteration *MIt*, one can also define a function $eval : LamE \subseteq Lam$ that evaluates all the explicit flattenings and thus yields the representation of a usual lambda term [14]. In [14], also *eval* is seen in *LNMIt* to be a natural transformation.

2.2 Generalized Mendler-style iteration *GMIt*

We would like to define a representation of substitution on *LamE*. As for *Lam*, the most elegant solution is to define a parallel substitution

$$substE : \forall A \forall B.\, (A \to LamE\, B) \to LamE\, A \to LamE\, B \ ,$$

where for a *substitution rule* $f : A \to LamE\, B$, the term $substE_{A,B}\, f\, t : LamE\, B$ is the result of substituting every variable $a : A$ in the term representation $t : LamE\, A$ by the term $f\, a : LamE\, B$. The operation *substE* would then qualify as Kleisli extension operation of a monad in Kleisli form (a. k. a., bind operation in Haskell).

Evidently, the desired type of *substE* is not of the form $LamE \subseteq G$ for any $G : \kappa_1$. However, it is equivalent (just move the universal quantification over B across an implication) to $LamE \subseteq Ran_{LamE}\, LamE$, with

$$Ran_H\, G := \lambda A.\, \forall B.\, (A \to HB) \to GB$$

for any $H, G : \kappa_1$, which is a syntactic form of a right Kan extension of G along H. This categorical notion has been introduced into the research on nested datatypes in [5], while in [12], it was first used to justify termination of iteration schemes, and in [10], it served as justification of *generalized* Mendler iteration, to be defined next. Its motivation was better efficiency (it covers the efficient folds of [15], see [10]), but visually, this is just hiding of the Kan extension from the user. Technically, this also means a formulation that does not need impredicativity of the universe κ_0 because, only with impredicative κ_0, we have $Ran_H\, G : \kappa_1$. Hence, we stay within pCIC.

The trick is to use the notion of *relativized refined containment* [10]: given $X, H, G : \kappa_1$, define the abbreviation

$$X \leq_H G := \forall A \forall B.\, (A \to HB) \to XA \to GB.$$

Generalized Mendler iteration consists of a constant (the iterator)

$$GMIt : \forall H^{\kappa_1} \forall G^{\kappa_1} . (\forall X^{\kappa_1} . X \leq_H G \to FX \leq_H G) \to \mu F \leq_H G$$

and the generalized iteration rule

$$GMIt\, H\, G\, s\, A\, B\, f\, (in\, A\, t) \simeq s\, (\mu F)\, (GMIt\, H\, G\, s)\, A\, B\, f\, t \ .$$

As mentioned before, *GMIt* can again be justified within F^ω, hence ensuring termination of the rewrite system underlying $\simeq$.

Coming back to *substE*, we note that its desired type is $LamE \leq_{LamE} LamE$, and in fact, we can define $substE := GMIt\, LamE\, LamE\, s_{substE}$ with

$$s_{substE} : \forall X^{\kappa_1} . X \leq_{LamE} LamE \to LamEF\, X \leq_{LamE} LamE \ ,$$

given by (note that we start omitting the type parameters at many places)

$$\begin{array}{l}
\lambda X^{\kappa_1} \lambda it^{X \leq_{LamE} LamE} \lambda A \lambda B \lambda f^{A \to LamE\, B} \lambda t^{LamEF\, X\, A} . \mathrm{match}\, t\, \mathrm{with} \\
\quad \mid inl(inl(inl\, a^A)) \qquad\qquad \mapsto f a \\
\quad \mid inl(inl(inr(t_1^{XA}, t_2^{XA}))) \mapsto appE\, (it_{A,B}\, f\, t_1)\, (it_{A,B}\, f\, t_2) \\
\quad \mid inl(inr\, r^{X(option\, A)}) \qquad \mapsto absE(it_{option\, A, option\, B}\, (liftE\, f)\, r) \\
\quad \mid inr\, e^{X(XA)} \qquad\qquad\quad \mapsto flatE(it_{XA, LamE\, B}(varE_{LamE\, B} \circ (it_{A,B}\, f))\, e)) \ .
\end{array}$$

Here, we used an analogue of lifting for *Lam* in [6],

$$liftE : \forall A \forall B . (A \to LamE\, B) \to option\, A \to LamE(option\, B) \ ,$$

definable by pattern-matching with properties

$$\begin{array}{ll}
liftE_{A,B}\, f\, None & \simeq varE_{option\, B} None \ , \\
liftE_{A,B}\, f\, (Some\, a) & \simeq lamE\, Some\, (f a) \ ,
\end{array}$$

where renaming *lamE* is essential.

Note that $varE_{LamE\, B} \circ (it_{A,B}\, f)$ has type $XA \to LamE(LamE\, B)$ (the infix operator $\circ$ denotes composition of functions). From the point of view of clarity of the definition, we would have much preferred $flatE\, (lamE\, (it_{A,B}\, f)\, e)$ to the term in the last clause of the definition of s_{substE}. It would only type-check *after* instantiating X with *LamE*, hence generalized Mendler iteration cannot accept this alternative. However, a system of sized nested datatypes [27] could assign more informative types to *lamE* in order to solve this problem, but there do not yet exist systematic means of program verification for them.

Our definition only satisfies

$$substE\, f\, (flatE\, e) \simeq flatE(substE(varE \circ (substE\, f))\, e) \ ,$$

to be seen immediately from the generalized iteration rule (assuming again proper $\simeq$-behaviour of pattern matching). Note that $substE\, f\, (varE\, a) \simeq f a$ is already the verification of the first of the three monad laws for the purported monad $(LamE, varE, substE)$ in Kleisli form (where *varE* is the unit of the monad).

The following will be provable about *substE* in the system *LNGMIt*, where we mean the universal (and well-typed) closure of all statements:

1. $(\forall a^A.\, fa = ga) \to substE\, f\, t = substE\, g\, t$
2. $(\forall a^A.\, a \in EFV\, t \to fa = ga) \to substE\, f\, t = substE\, g\, t$
3. $lamE\, g\, (substE\, f\, t) = substE\, (\, (lamE\, g) \circ f)\, t$
4. $substE\, g\, (lamE\, f\, t) = substE\, (g \circ f)\, t$
5. $substE\, g\, (substE\, f\, t) = substE\, ((substE\, g) \circ f)\, t$
6. $EFV(substE\, f\, t) = flatten(map\, (EFV \circ f)\, (EFV\, t))$

The first is extensionality, the second refined extensionality, the third and fourth are the two halves of naturality (number 4 appears to be an instance of map fusion, as studied in [15]), the fifth is one of the other two monad laws, and the last a means to express that EFV is a monad morphism from $LamE$ (that does *not* satisfy the last remaining monad law) to $List$. An easy consequence from it is $b \in EFV(substE\, f\, t) \to \exists a.\, a \in EFV\, t \wedge b \in EFV(fa)$. This consequence and the first five statements are all intuitively true for substitution, renaming and the enumeration of free variables, and they were all known for Lam, hence without explicit flattening. The point here is that also the truly nested datatype $LamE$ can be given a logic that allows such proofs within intensional type theory, hence in a system with static termination guarantee, interactive program construction (in implementations such as Coq) and no need to *represent* the programs in a programming logic: the program's behaviour with respect to $\simeq$ is directly available.

3 Logic for Natural Generalized Mendler-style Iteration

First, we recall *LNMIt* from [14], then we extend it by *GMIt* and its definitional rules in order to obtain its extension *LNGMIt*.

3.1 *LNMIt*

In *LNMIt*, for a nested datatype μF, we require that $F : \kappa_2$ preserves *extensional functors*. In pCIC, we may form for $X : \kappa_1$ the dependently-typed record $\mathcal{E}X$ that contains a map term $m : mon\, X$, a proof e of extensionality of m, i. e., of $ext\, m$, and proofs f_1, f_2 of the first and second functor laws for (X, m), defined by the propositions

$fct_1\, m := \forall A \forall x^{X\,A}.\, m\, A\, A\, (\lambda y.y)\, x = x$,
$fct_2\, m := \forall A \forall B \forall C\, \forall f^{A \to B}\, \forall g^{B \to C}\, \forall x^{X\,A}.\, m\, A\, C\, (g \circ f)\, x = m\, B\, C\, g\, (m\, A\, B\, f\, x)$.

Given a record ef of type $\mathcal{E}X$, Coq's notation for its field m is $m\, ef$, and likewise for the other fields. We adopt this notation instead of the more common $ef.m$. Preservation of extensional[6] functors for F is required in the form of a term of type $\forall X^{\kappa_1}.\, \mathcal{E}\, X \to \mathcal{E}(FX)$, and *LNMIt* is defined to be pCIC with $\kappa_0 := Set$, extended by the constants and rules of Figure 1, adopted from [14]. In

[6] While the functor laws are certainly an important ingredient of program verification, the extensionality requirement is more an artifact of our intensional type theory, as discussed in Section 2.1.

Parameters:

$F \quad : \kappa_2$

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

Constants:

$\mu F \quad : \kappa_1$

$map_{\mu F} : mon(\mu F)$

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

$MIt \quad : \forall G^{\kappa_1}.\, (\forall X^{\kappa_1}.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G$

$\mu FInd \quad : \forall P : \forall A.\, \mu F A \to Prop.\, \Big(\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(m\, ef,\, map_{\mu F})}.$

$\Big(\forall A \forall x^{XA}.\, P_A(j_A\, x)\Big) \to \forall A \forall t^{FXA}.\, P_A(In\, ef\, j\, n\, t)\Big)$

$\to \forall A \forall r^{\mu F A}.\, P_A\, r$

Rules:

$map_{\mu F}\, f\, (In\, ef\, j\, n\, t) \simeq In\, ef\, j\, n\, (m(Fp\mathcal{E}\, ef)\, f\, t)$

$MIt\, s\, (In\, ef\, j\, n\, t) \simeq s\, (\lambda A.\, (MIt\, s)_A \circ j_A)\, t$

$\lambda A \lambda x^{\mu F A}.\, (MIt\, s)_A\, x \simeq MIt\, s$

Fig. 1. Specification of *LNMIt* as extension of pCIC.

LNMIt, one can show the following theorem [14, Theorem 3] about canonical elements: 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:

$$\begin{aligned} m\, ef_{\mu F} &\simeq map_{\mu F}\ , \\ map_{\mu F}\, f\, (InCan\, t) &\simeq InCan(m\, (Fp\mathcal{E}\, ef_{\mu F})\, f\, t)\ , \\ MIt\, s\, (InCan\, t) &\simeq s\, (MIt\, s)\, t\ . \end{aligned}$$

(The proof of this theorem needs the induction rule $\mu FInd$ in order to show that $map_{\mu F}$ is extensional and satisfies the functor laws. These proofs enter $ef_{\mu F}$, and *In* can then be instantiated with $X := \mu F$, $ef := ef_{\mu F}$ and j the identity on μF with its trivial proof of naturality, to yield the desired *InCan*.)

This will now be related to the presentation in Section 2.1: The datatype constructor *In* is way more complicated than our previous *in*, but we get back *in* in the form of *InCan* that only constructs the "canonical elements" of the nested datatype μF. The map term $map_{\mu F}$ for μF, which does renaming in our example of *LamE*, as demonstrated in Section 2.1, is an integral part of the system definition since it occurs in the type of *In*. This is a form of simultaneous induction-recursion [28], where the inductive definition of μF is done simultaneously with the recursive definition of $map_{\mu F}$. The Mendler iterator *MIt* has not been touched at all; there is just a more general iteration rule that also covers non-canonical elements, but for the canonical elements, we get the same behaviour, i. e., the same equation with respect to $\simeq$. The crucial part is the induction principle $\mu FInd$, where *Prop* denotes the universe of propositions (all our propositional equalities and their universal quantifications belong to it). Without access to the argument n that assumes naturality of j as a transforma-

tion from $(X, m\,ef)$ to $(\mu F, map_{\mu F})$, one would not be able to prove naturality of $MIt\,s$, i. e., of iteratively defined functions on the nested datatype μF. The author is not aware of ways how to avoid non-canonical elements and nevertheless have an induction principle that allows to establish naturality of $MIt\,s$ [14, Theorem 1].

The system *LNMIt* can be defined within CIC with impredicative *Set*, extended by the principle of proof irrelevance, i. e., by $\forall P : Prop\,\forall p_1^P \forall p_2^P . p_1 = p_2$. This is the main result of [14], and it is based on an impredicative construction of simultaneous inductive-recursive definitions by Capretta [29] that could be extended to work for this situation. It is also available in the form of a Coq module [18] that allows to benefit from the evaluation of terms in Coq. For this, it is crucial that convertibility in *LNMIt* implies convertibility in that implementation.

The "functor" *LamEF* is easily seen to fulfill the requirement of *LNMIt* to preserve extensional functors (using [14, Lemma 1 and Lemma 2]). As mentioned in Section 2.1, *LNMIt* allows to prove that $EFV \in \mathcal{N}(lamE, map)$, and this is an instance of [14, Theorem 1].

3.2 *LNGMIt*

Let *LNGMIt* be the extension of *LNMIt* by the constant *GMIt* from section 2.2,

$$GMIt : \forall H^{\kappa_1} \forall G^{\kappa_1} . (\forall X^{\kappa_1} . X \leq_H G \to FX \leq_H G) \to \mu F \leq_H G \ ,$$

and the following two rules:

$$GMIt_{H,G}\,s\,f\,(In\,ef\,j\,n\,t) \simeq s\,(\lambda A \lambda B \lambda f^{A \to HB} . (GMIt_{H,G}\,s\,A\,B\,f) \circ j_A)\,f\,t \ ,$$
$$\lambda A \lambda B \lambda f^{A \to HB} \lambda x^{\mu F A} . GMIt_{H,G}\,s\,A\,B\,f\,x \simeq GMIt_{H,G}\,s \ .$$

Theorem [14, Theorem 3] about $ef_{\mu F}$ and *InCan* for *LNMIt* immediately extends to *LNGMIt* and yields the following additional convertibility:

$$GMIt\,s\,f\,(InCan\,t) \simeq s\,(GMIt\,s)\,f\,t \ ,$$

which has this concise form only because of the η-rule for *GMIt* that was made part of *LNGMIt*. Thus, we get back the original behaviour of *GMIt* described in Section 2.2, but with the derived datatype constructor *InCan* instead of the *defining* datatype constructor *in*.

Lemma 1. *The system LNGMIt can be defined within LNMIt if the universe κ_0 of computationally relevant types is impredicative.*

Proof. The proof is nothing but the observation that the embedding of GMIt^ω into MIt^ω of [10, Section 4.3] extends for our situation of a rank-2 inductive constructor μF to non-canonical elements, i. e., the full datatype constructor *In* instead of only *in*, considered in that work: define for $H, G : \kappa_1$ the terms

$$toGRan \quad := \lambda X^{\kappa_1} \lambda h^{X \leq_H G} \lambda A \lambda x^{XA} \lambda B \lambda f^{A \to HB} . h\,A\,B\,f\,x \ ,$$
$$fromGRan := \lambda X^{\kappa_1} \lambda h^{X \subseteq Ran_H\,G} \lambda A \lambda B \lambda f^{A \to HB} \lambda x^{XA} . h\,A\,x\,B\,f \ .$$

These terms establish the logical equivalence of $X \leq_H G$ and $X \subseteq \mathit{Ran}_H\, G$:

$$\begin{array}{ll} \mathit{toGRan} & : \forall X^{\kappa_1}.\, X \leq_H G \to X \subseteq \mathit{Ran}_H\, G\ , \\ \mathit{fromGRan} & : \forall X^{\kappa_1}.\, X \subseteq \mathit{Ran}_H\, G \to X \leq_H G\ . \end{array}$$

Define for a step term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ for $\mathit{GMIt}_{H,G}$ the step term s' for $\mathit{MIt}_{\mathit{Ran}_H\, G}$ as follows:

$$s' := \lambda X^{\kappa_1} \lambda h^{X \subseteq \mathit{Ran}_H\, G}.\, \mathit{toGRan}_{FX}\, (s_X\, (\mathit{fromGRan}_X\, h))\ .$$

Then, we can define

$$\mathit{GMIt}_{H,G}\, s := \mathit{fromGRan}_{\mu F}\, (\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')$$

and readily observe that the main definitional rule for *GMIt* in *LNGMIt* is inherited from that of *MIt* in *LNMIt* and that the other rule is immediate from the definition.[7] Impredicativity of κ_0 is needed to have $\mathit{Ran}_H\, G : \kappa_1$, as mentioned in Section 2.2. □

Corollary 1. *The system LNGMIt can be defined within CIC with impredicative Set, extended by the principle of propositional proof irrelevance, i. e., by* $\forall P : \mathit{Prop}\, \forall p_1^P \forall p_2^P.\, p_1 = p_2$.

Proof. Use the the previous lemma and the main theorem of [14] that states the same property of *LNMIt*.

[14] is more detailed about how much proof irrelevance is needed for the proof.

Lemma 2 (Uniqueness of $\mathit{GMIt}\, s$**).** *Assume* $H, G : \kappa_1$, $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ *and* $h : \mu F \leq_H G$ *(the candidate for being* $\mathit{GMIt}\, s$*). Assume further the following extensionality property of* s *(*s *only depends on the extension of its first function argument, but in a way adapted to the parameter* f*):*

$$\forall X^{\kappa_1} \forall g, h : X \leq_H G.\, (\forall A \forall B \forall f^{A \to HB} \forall x^{XA}.\, g\, f\, x = h\, f\, x) \to \\ \forall A \forall B \forall f^{A \to HB} \forall y^{FXA}.\, s\, g\, f\, y = s\, h\, f\, y\ .$$

Assume finally that h *satisfies the equation for* $\mathit{GMIt}\, s$*:*

$$\forall X^{\kappa_1} \forall \mathit{ef}^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(m\, \mathit{ef},\, \mathit{map}_{\mu F})} \forall A \forall B \forall f^{A \to HB} \forall t^{FXA}. \\ h_{A,B}\, f\, (\mathit{In}\, \mathit{ef}\, j\, n\, t) = s\, (\lambda A \lambda B \lambda f^{A \to HB}.\, (h_{A,B}\, f) \circ j_A)\, f\, t\ .$$

Then, $\forall A \forall B \forall f^{A \to HB} \forall r^{\mu F\, A}.\, h_{A,B}\, f\ r = \mathit{GMIt}\, s\, f\, r$.

Proof. By the induction principle $\mu\mathit{FInd}$, as for [14, Theorem 2].

Given type constructors X, H, G, the type $X \leq_H G$ has an embedded function space, so there is the natural question whether an inhabitant h of $X \leq_H G$

[7] Strictly speaking, we have to define *GMIt* itself, but this can be done just by abstracting over G, H and s that are only parameters of the construction.

only depends on the extension of this function parameter. This is expressed by the proposition (*gext* stands for generalized extensionality)

$$gext\, h := \forall A \forall B \forall f, g : A \to HB.\, (\forall a^A.\, fa = ga) \to \forall r^{XA}.\, h_{A,B}\, f\, r = h_{A,B}\, g\, r \ .$$

The earlier definition of *ext* is the special instance where X and G coincide and where H is the identity type transformation $Id_{\kappa_0} := \lambda A.\, A$.

Given type constructors H, G and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$, we say that *s preserves extensionality* if $\forall X^{\kappa_1} \forall h^{X \leq_H G}.\, gext\, h \to gext(s\, h)$ holds.

Lemma 3 (Extensionality of *GMIt s*). *Assume type constructors H, G and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality in the above sense. Then $GMIt\, s : \mu F \leq_H G$ is extensional, i. e., $gext(GMIt\, s)$ holds.*

Proof. An easy application of $\mu FInd$.

Coming back to the representation *substE* of substitution on *LamE* from Section 2.2, straightforward reasoning shows that s_{substE} preserves extensionality, hence Lemma 3 yields *gext substE*, which proves the first item in the list on page 11. Its refinement, namely the second item in that list,

$$(\forall a^A.\, a \in EFV\, t \to fa = ga) \to substE\, f\, t = substE\, g\, t \ ,$$

needs a direct proof by the induction principle $\mu FInd$, where the behaviour of *EFV* on non-canonical elements plays an important role, but is nevertheless elementary.

4 Naturality in *LNGMIt*

In order to establish an extension of the map fusion law of [15], a notion of naturality for functionals $h : X \leq_H G$ has to be introduced. We first treat the case where H is the identity Id_{κ_0}. In this case, we omit the argument for H from $X \leq_H G$ and only write $X \leq G$. Assume a function $h : X \subseteq G$ and map terms $m_X : mon\, X$ and $m_G : mon\, G$. Figure 2, which is strongly inspired by [12, Figure 1], recalls naturality, i. e., $h \in \mathcal{N}(m_X, m_G)$ is displayed in the form of a commuting diagram (where commutation means pointwise propositional equality of the compositions) for any A, B and $f : A \to B$. The diagonal marked by $h\, f$ in Figure 2 can then be defined by either $(m_G\, f) \circ h_A$ or $h_B \circ (m_X\, f)$, and this yields a functional of type $\forall A \forall B.\, (A \to B) \to XA \to GB$, again called h in [30, Exercise 5 on page 19]. Its type is more concisely expressed as $X \leq G$. The exercise in [30] (there expressed in pure category-theoretic terms) can be seen to establish a naturality-like diagram of the functional h. Namely, also the diagram in Figure 3 commutes for all A, B, C, $f : A \to B$ and $g : B \to C$. Moreover, from a functional h for which the second diagram commutes, one obtains in a unique way a natural transformation h from X to G with h_A being $h\, id_A$. In category theory, this is a simple exercise, but in our intensional setting, this allows to define naturality for any $X, G : \kappa_1$, $m_X : mon\, X$, $m_G : mon\, G$ and $h : X \leq G$.

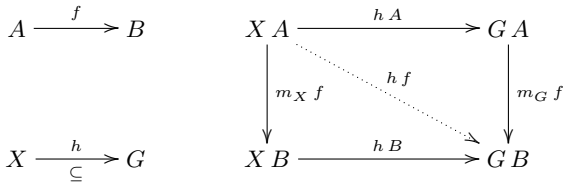

Fig. 2. Naturality of $h : X \subseteq G$

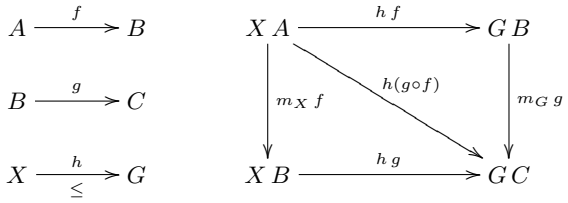

Fig. 3. Naturality of $h : X \leq G$

Definition 1 (Naturality of $h : X \leq G$). *Given $X, G : \kappa_1$, $m_X : mon\, X$, $m_G : mon\, G$ and $h : X \leq G$, the functional h is called natural with respect to m_X and m_G if it satisfies the following two laws:*

1. $\forall A \forall B \forall C \forall f^{A \to B} \forall g^{B \to C} \forall x^{XA}.\, m_G\, g\, (h_{A,B}\, f\, x) = h_{A,C}\, (g \circ f)\, x$
2. $\forall A \forall B \forall C \forall f^{A \to B} \forall g^{B \to C} \forall x^{XA}.\, h_{B,C}\, g\, (m_X\, f\, x) = h_{A,C}\, (g \circ f)\, x$

Mac Lane's exercise [30] can readily be extended to the generality of $X \leq_H G$, with arbitrary H, and a function $h : X \circ H \subseteq G$, but with less pleasing diagrams. We therefore content ourselves with an algebraic description of the parts we need for *LNGMIt*.

Definition 2 (Naturality of $h : X \leq_H G$). *Given $X, H, G : \kappa_1$ and $h : X \leq_H G$, define the two parts of naturality of h as follows: If $m_H : mon\, H$ and $m_G : mon\, G$, define the first part $gnat_1\ m_H\, m_G\, h$ by*

$$\forall A \forall B \forall C \forall f^{A \to HB} \forall g^{B \to C} \forall x^{XA}.\, m_G\, g\, (h_{A,B}\, f\, x) = h_{A,C}\, ((m_H\, g) \circ f)\, x \ .$$

If $m_X : mon\, X$, define the second part $gnat_2\ m_X\, h$ by

$$\forall A \forall B \forall C \forall f^{A \to B} \forall g^{B \to HC} \forall x^{XA}.\, h_{B,C}\, g\, (m_X\, f\, x) = h_{A,C}\, (g \circ f)\, x \ .$$

Since Id_{κ_0} has the map term $\lambda A \lambda B \lambda f^{A \to B} \lambda x^{A}.\, fx$, Definition 1 is an instance of Definition 2.

The backwards direction of Mac Lane's exercise for our generalization is now mostly covered by the following lemma.

Lemma 4. *Given $X, H, G : \kappa_1$, $m_X : mon\, X$, $m_H : mon\, H$, $m_G : mon\, G$ and $h : X \leq_H G$ such that $gnat_1\ m_H\, m_G\, h$ and $gnat_2\, m_X\, h$ hold, the function $h^{\subseteq} := \lambda A \lambda x^{X(HA)}.\, h_{HA,A}\, (\lambda y^{HA}.\, y)\, x : X \circ H \subseteq G$ is natural: $h^{\subseteq} \in \mathcal{N}(m_X \star m_H,\, m_G)$. Here, $m_X \star m_H$ denotes the canonical map term for $X \circ H$, obtained from m_X and m_H.*

Proof. Elementary.

Thus, finally, one can define and argue about functions of type $(\mu F) \circ H \subseteq G$ through $(GMIt\, s)^{\subseteq}$.

Lemma 5 (First part of naturality of *GMIt s*). *Given $H, G : \kappa_1$, map terms $m_H : mon\, H$, $m_G : mon\, G$ and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality. Assume further*

$$\forall X^{\kappa_1} \forall h^{X \leq_H G}.\, \mathcal{E}\, X \to gext\, h \to gnat_1\ m_H\, m_G\, h \to gnat_1\ m_H\, m_G\, (s\, h)\ .$$

Then, $GMIt\, s$ satisfies the first part of naturality, i. e., $gnat_1\ m_H\, m_G\, (GMIt\, s)$.

Proof. Induction with $\mu FInd$. The proof does *not* use the naturality of argument j, provided by the context of the induction step. Preservation of extensionality is used in order to apply Lemma 3 for the function representing the recursive calls, because that function becomes the h of the main assumption on s.

An an instance of this lemma, one can prove the third item in the list on page 11 on properties of *substE*.

Theorem 1 (Second part of naturality of *GMIt s*—map fusion). *Given $H, G : \kappa_1$ and a term $s : \forall X^{\kappa_1}.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality. Assume further*

$$\forall X^{\kappa_1} \forall h^{X \leq_H G} \forall ef^{\mathcal{E}\, X}.\, gext\, h \to gnat_2\, (m\, ef)\, h \to gnat_2\, (m\, (Fp\mathcal{E}\ ef))\, (s\, h)\ .$$

Then, $GMIt\, s$ satisfies the second part of naturality, i. e., $gnat_2\ map_{\mu F}\, (GMIt\, s)$.

Proof. Induction with $\mu FInd$. Again, we have to use Lemma 3 for the function

$$h := \lambda A \lambda B \lambda f^{A \to HB}.\, (GMIt_{H,G}\, s\, A\, B\, f) \circ j_A$$

representing the recursive calls in the right-hand side of the rule for *GMIt* in the definition of *LNGMIt*. Since we also have to provide a proof of $gnat_2\, (m\, ef)\, h$, we crucially need naturality of j that comes with the induction principle.

Although the proof is quite simple (again, see the full proof in the Coq development [17]), this is the main point of the complicated system *LNGMIt* with its inductive-recursive nature: ensure naturality to be available for j inside the inductive step of reasoning on μF. One might wonder whether this theorem could be an instance of [14, Theorem 1], using the definition of *GMIt* in Lemma 1 for impredicative κ_0. This is not true, due to problems with extensionality: Proving

propositional equality between functions rarely works in intensional type theory such as CIC, and the use of $Ran_H\ G$ in the construction of Lemma 1 introduces values of function type.

As an instance of this theorem, one can prove the fourth item in the list on page 11 on properties of *substE*. The fifth item (the interchange law for substitution that is one of the monad laws) can then be proven by the induction principle $\mu FInd$, using extensionality and both parts of naturality (hence, the items 1, 3 and 4 that are based on Lemma 3, Lemma 5 and Theorem 1) in the case for the representation of lambda abstraction (recall that *liftE* is defined by help of *lamE*).

5 Completion of the Case Study on Substitution

The last item on page 11 in the list of properties of *substE* can be proven by the induction principle $\mu FInd$ without any results about *LamE*, just with several preparations about lists, also using naturality of *EFV* in the proof of the case for the representation of lambda abstraction. Thus, that property list can be considered as finished.

We are not yet fully satisfied: The last monad law is missing, namely

$$\forall A \forall t^{LamE\ A}.\ substE\ varE_A\ t = t\ \ .$$

Any proof attempt breaks due to the presence of non-canonical terms in *LNGMIt*. We call any term of the form $InCan\ t$ with $t : F(\mu F)A$ a canonical term in $\mu F\, A$, but since this notion is not recursively applied to the subterms, we cannot hope to prove the above monad law for all the canonical terms in the family *LamE* either.

The following is an ad hoc notion for our example. For the truly nested datatype *Bush* of "bushes" with $Bush\ A = 1 + A \times Bush(Bush\ A)$, a similar notion has been studied by the author in [14, Section 4.2], also introducing a "canonization" function that transforms any bush into a hereditarily canonical bush and that does not change hereditarily canonical bushes with respect to propositional equality.

Definition 3 (Hereditarily canonical term). *Define the notion of hereditarily canonical elements of the nested datatype LamE, the predicate can* : $\forall A.\ LamE\ A \to Prop$*, inductively by the following four closure rules:*

- $\forall A \forall a^A.\ can\ (varE\ a)$
- $\forall A \forall t_1^{LamE\ A} \forall t_2^{LamE\ A}.\ can\ t_1 \to can\ t_2 \to can(appE\ t_1\ t_2)$
- $\forall A \forall r^{LamE(option\ A)}.\ can\ r \to can\ (absE\ r)$
- $\forall A \forall e^{LamE(LamE\ A)}.\ can\ e \to (\forall t^{LamE\ A}.\ t \in EFV\, e \to can\ t) \to can\ (flatE\ e)$

This definition is strictly positive and, formally, infinitely branching. However, there are always only finitely many t that satisfy $t \in EFV\, e$. System pCIC does not need this latter information for having induction principles for *can*,

and *LNGMIt* comprises pCIC, but this is not the part that is under study here. Therefore, all proofs by induction on *can* are not considered to be of real interest for this article. Except for the information which results are used in these proofs.

Note once again the simultaneous inductive-recursive structure that is avoided here: If only hereditarily canonical elements were to be considered from the beginning, one would have to define their free variables simultaneously since the last clause of the definition refers to them at a negative position.

5.1 Results for Hereditarily Canonical Terms

Using refined extensionality of *substE* (property number 2 in the list on page 11) in the induction step for $\mathit{flatE}\, e$, induction on *can* provides the relativization of the missing monad law to hereditarily canonical terms:

$$\forall A \forall t^{\mathit{LamE}\, A}.\, \mathit{can}\, t \to \mathit{substE}\, \mathit{varE}_A\, t = t \ .$$

Renaming *lamE* preserves hereditary canonicity:

$$\forall A \forall B \forall f^{A \to B} \forall t^{\mathit{LamE}\, A}.\, \mathit{can}\, t \to \mathit{can}(\mathit{lamE}\, f\, t) \ .$$

This is proven by induction on *can*, and the crucial *flatE* case needs the following identification of free variables of $\mathit{lamE}\, f\, t$:

$$\forall A \forall B \forall f^{A \to B} \forall t^{\mathit{LamE}\, A} \forall b^{B}.\, b \in \mathit{EFV}(\mathit{lamE}\, f\, t) \to \exists a^{A}.\, a \in \mathit{EFV}\, t \wedge b = fa \ ,$$

which is nearly an immediate consequence of naturality of *EFV*.

Analogously, *substE* preserves hereditary canonicity:

$$\forall A \forall B \forall f^{A \to \mathit{LamE}\, B} \forall t^{\mathit{LamE}\, A}.\\ (\forall a^{A}.\, a \in \mathit{EFV}\, t \to \mathit{can}(fa)) \to \mathit{can}\, t \to \mathit{can}(\mathit{substE}\, f\, t) \ .$$

Again, this is proven by induction on *can*, and again, the crucial case is with $\mathit{flatE}\, e$, for which free variables of $\mathit{substE}\, f\, t$ have to be identified, but this has already been mentioned as a consequence of property number 6 in the list on page 11.

As an immediate consequence of the last monad law, preservation of hereditary canonicity by *lamE* and the second part of naturality of *substE* (item 4 of the list, proven by map fusion), one can see *lamE* as a special instance of *substE* for hereditarily canonical elements:

$$\forall A \forall B \forall f^{A \to B} \forall t^{\mathit{LamE}\, A}.\, \mathit{can}\, t \to \mathit{lamE}\, f\, t = \mathit{substE}\, (\mathit{varE}_B \circ f)\, t \ .$$

From this, evidently, we get the more perspicuous equation for $\mathit{substE}\, f\, (\mathit{flatE}\, e)$, discussed on page 10, but only for hereditarily canonical e and only with propositional equality:

$$\forall A \forall B \forall f^{A \to \mathit{LamE}\, B} \forall e^{\mathit{LamE}(\mathit{LamE}\, A)}.\, \mathit{can}\, e \to \\ \mathit{substE}\, f\, (\mathit{flatE}\, e) = \mathit{flatE}(\mathit{lamE}\, (\mathit{substE}\, f)\, e) \ .$$

5.2 Hereditarily Canonical Terms as a Nested Datatype

Define $\mathit{LamEC} := \lambda A.\, \{t : \mathit{LamE}\ A \mid \mathit{can}\ t\} : \kappa_1$. The set comprehension notation stands for the inductively defined `sig` of Coq (definable within pCIC, hence within *LNGMIt*) which is a strong sum in the sense that the first projection $\pi_1 : \mathit{LamEC} \subseteq \mathit{LamE}$ yields the element t and the second projection the proof of $\mathit{can}\, t$.

Thus, we encapsulate hereditary canonicity already in the family *LamEC*. We will present *LamEC* as a truly nested datatype, but not one that comes as a μF from *LNGMIt*.

It is quite trivial to define datatype constructors

$$\begin{aligned} &\mathit{varEC} : \forall A.\, A \to \mathit{LamEC}\ A\ , \\ &\mathit{appEC} : \forall A.\, \mathit{LamEC}\ A \to \mathit{LamEC}\ A \to \mathit{LamEC}\ A\ , \\ &\mathit{absEC} : \forall A.\, \mathit{LamEC}(\mathit{option}\ A) \to \mathit{LamEC}\ A \end{aligned}$$

from their analogues in *LamE*. For the construction of

$$\mathit{flatEC} : \forall A.\, \mathit{LamEC}(\mathit{LamEC}\ A) \to \mathit{LamEC}\ A\ ,$$

the problem is as follows: Assume $e : \mathit{LamEC}(\mathit{LamEC}\ A)$. Then, its first projection, $\pi_1 e$, is of type $\mathit{LamE}(\mathit{LamEC}\ A)$. Therefore, the first projection of $\mathit{flatEC}\, e$ has to be

$$t := \mathit{flatE}(\mathit{lamE}\ (\pi_1)_A\ (\pi_1 e)) : \mathit{LamE}\ A\ ,$$

with the renaming with $(\pi_1)_A : \mathit{LamEC}\ A \to \mathit{LamE}\ A$ inside. Thanks to the preservation of hereditary canonicity by *lamE* and the identification of the variables of renamed terms, canonicity of t can be established.

Since *flatEC* is doing something with its argument, we cannot think of *LamEC* as being generated from the four datatype constructors. We see this more as a semantical construction whose properties can be studied. However, there is still the operational kernel available in the form of the definitional equality $\simeq$.

From preservation of hereditary canonicity by *lamE* and *substE*, one can easily define $\mathit{lamEC} : \mathit{mon}\, \mathit{LamEC}$ and

$$\mathit{substEC} : \forall A \forall B.\, (A \to \mathit{LamEC}\ B) \to \mathit{LamEC}\ A \to \mathit{LamEC}\ B\ .$$

The list of free variables is obtained through $\mathit{ECFV} : \mathit{LamEC} \subseteq \mathit{List}$, defined by composing *EFV* with π_1, which is then also natural. Therefore, one can immediately transfer the identification of free variables of $\mathit{lamE}\ f\ t$ and $\mathit{substE}\ f\ t$ to *lamEC* and *substEC*.

In order to have "real" results, proof irrelevance has to be assumed for the proofs of hereditary canonicity. From propositional proof irrelevance, as used in Corollary 1, it immediately follows that π_1 is injective:

$$\forall A \forall t_1, t_2 : \mathit{LamEC}\ A.\, \pi_1\, t_1 = \pi_1\, t_2 \to t_1 = t_2\ .$$

This is the only addition to *LNGMIt* that we adopt here. Then, all the properties of the list in Section 2.2 can be transferred to *substEC*, the recursive description (now only with propositional equality) of *lamE* can be carried over to *lamEC* that makes *LamEC* an extensional functor, and also the results of Section 5.1 that were relativized to hereditarily canonical terms now hold unconditionally for *lamEC* and *substEC*. Finally, a monad structure has been obtained. Once again, all the proofs are to be found in the Coq scripts [17].

6 Conclusions and Future Work

Recursive programming with Mendler-style iteration is able to cover intricate nested datatypes with functions whose termination is far from being obvious. But termination is not the only property of interest. A calculational style of verification that is based on generic results such as naturality criteria is needed on top of static analysis. The system *LNGMIt* and the earlier system *LNMIt* from which it is derived are an attempt to combine the benefits from both paradigms: the rich dependently-typed language secured by decidable type-checking and termination guarantees on one side and the laws that are inspired from category theory on the other side.

LNGMIt can prove naturality in many cases, with a notion of naturality that encompasses map fusion. However, the system is heavily based on the unintuitive non-canonical datatype constructor *In* which makes reasoning on paper somewhat laborious. This can be remedied by intensive use of computer aided proof development. The ambient system for the development of the metatheory and the case study is the Calculus of Inductive Constructions that is implemented by the Coq system. Proving and programming can both be done interactively. Therefore, *LNGMIt*, through its implementation in Coq, can effectively aid in the construction of terminating programs on nested datatypes and to establish their equational properties.

Certainly, the other laws in, e. g., [15] should be made available in our setting as well. Clearly, not only (generalized) iteration should be available for programs on nested datatypes. The author experiments with primitive recursion in Mendler style, but does not yet have termination guarantees [31].

An alternative to *LNGMIt* with its non-canonical elements could be a dependently-typed approach from the very beginning. This could be done by indexing the nested datatypes additionally over the natural numbers as with sized nested datatypes [27] where the size corresponds to the number of iterations of the datatype "functor" over the constantly empty family. But one could also try to define functions directly for all powers of the nested datatype (suggested to me by Nils Anders Danielsson) or even define all powers of it simultaneously (suggested to me by Conor McBride). The author has presented preliminary results at the TYPES 2004 meeting about yet another approach where the indices are finite trees that branch according to the different arguments that appear in the recursive equation for the nested datatype (based on ideas by Anton Setzer and Peter Aczel).

References

1. Bird, R., Meertens, L.: Nested datatypes. In Jeuring, J., ed.: Mathematics of Program Construction, MPC'98, Proceedings. Volume 1422 of Lecture Notes in Computer Science., Springer Verlag (1998) 52–67
2. Bird, R., Gibbons, J., Jones, G.: Program optimisation, naturally. In Davies, J., Roscoe, B., Woodcock, J., eds.: Millenial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symp. in Honour of Professor Sir Anthony Hoare, Palgrave (2000)
3. Hinze, R.: Efficient generalized folds. In Jeuring, J., ed.: Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal. (2000)
4. Bellegarde, F., Hook, J.: Substitution: A formal methods case study using monads and transformations. Science of Computer Programming **23** (1994) 287–311
5. Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming **9**(1) (1999) 77–91
6. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In Flum, J., Rodríguez-Artalejo, M., eds.: Computer Science Logic, 13th International Workshop, CSL '99, Proceedings. Volume 1683 of Lecture Notes in Computer Science., Springer Verlag (1999) 453–468
7. Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.1. Project LogiCal, INRIA. (2006) System available at `coq.inria.fr`.
8. Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I (1996)
9. Letouzey, P.: A new extraction for Coq. In Geuvers, H., Wiedijk, F., eds.: TYPES 2002 Post-Conference Proceedings. Volume 2646 of Lecture Notes in Computer Science., Springer Verlag (2003) 200–219
10. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333**(1–2) (2005) 3–66
11. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2) (1999) 200–222
12. Abel, A., Matthes, R.: (Co-)iteration for higher-order nested datatypes. In Geuvers, H., Wiedijk, F., eds.: TYPES 2002 Post-Conference Proceedings. Volume 2646 of Lecture Notes in Computer Science., Springer Verlag (2003) 1–20
13. Bird, R., de Moor, O.: Algebra of Programming. Volume 100 of International Series in Computer Science. Prentice Hall (1997)
14. Matthes, R.: An induction principle for nested datatypes in intensional type theory. Journal of Functional Programming (2008) To appear.
15. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Aspects of Computing **16**(1) (2004) 19–35
16. Johann, P., Ghani, N.: Initial algebra semantics is enough! In Ronchi Della Rocca, S., ed.: Typed Lambda Calculi and Applications (TLCA) 2007, Proceedings. Volume 4583 of Lecture Notes in Computer Science., Springer Verlag (2007) 207–222
17. Matthes, R.: Coq development for "Nested datatypes with generalized Mendler iteration: map fusion and the example of the representation of untyped lambda calculus with explicit flattening". `http://www.irit.fr/~Ralph.Matthes/Coq/MapFusion/` (January 2008)
18. Matthes, R.: Coq development for "An induction principle for nested datatypes in intensional type theory". `http://www.irit.fr/~Ralph.Matthes/Coq/InductionNested/` (January 2008)

19. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y., IEEE Computer Society Press (1987) 30–36
20. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical Structures in Computer Science **14** (2004) 97–141
21. Uustalu, T., Vene, V.: A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In Haveraaen, M., Owe, O., eds.: Selected Papers of the 8th Nordic Workshop on Programming Theory (NWPT '96). Volume 248 of Research Reports, Department of Informatics, University of Oslo. (May 1997) 237–246
22. Matthes, R.: Naive reduktionsfreie Normalisierung (translated to English: naive reduction-free normalization). Slides of talk on December 19, 1996, given at the Bern Munich meeting on proof theory and computer science in Munich, available at the author's homepage (December 1996)
23. Hofmann, M.: Extensional concepts in intensional type theory. PhD thesis, University of Edinburgh (1995) Available as report ECS-LFCS-95-327.
24. Altenkirch, T.: Extensional equality in intensional type theory. In: 14th Annual IEEE Symposium on Logic in Computer Science (LICS 1999), IEEE Computer Society (1999) 412–420
25. Oury, N.: Extensionality in the calculus of constructions. In Hurd, J., Melham, T.F., eds.: Theorem Proving in Higher Order Logics. Proceedings. Volume 3603 of Lecture Notes in Computer Science., Springer Verlag (2005) 278–293
26. Wadler, P.: Theorems for free! In: Proceedings of the fourth international conference on functional programming languages and computer architecture, Imperial College, London, England, September 1989, ACM Press (1989) 347–359
27. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Doktorarbeit (PhD thesis), LMU München (2006)
28. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. The Journal of Symbolic Logic **65**(2) (2000) 525–549
29. Capretta, V.: 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) (March 2004)
30. Mac Lane, S.: Categories for the Working Mathematician. second edn. Volume 5 of Graduate Texts in Mathematics. Springer Verlag (1998)
31. Matthes, R.: Recursion on nested datatypes in dependent type theory. In Beckmann, A., Dimitracopoulos, C., Löwe, B., eds.: Logic and Theory of Algorithms. Volume 5028 of Lecture Notes in Computer Science., Springer Verlag (2008) To appear.