Abstract

A definitional extension LNGMIIt of the Calculus of Inductive Constructions (CIC), that underlies the proof assistant Coq, is presented that allows also to program with nested datatypes that are not legal data type definitions of CIC since they are “truly nested”. LNGMIIt ensures termination of recursively defined functions that follow iteration schemes in the style of N. Mendler. Characteristically for them, termination comes from polymorphic typing instead of structural requirements on recursive calls.

LNGMIIt comes with an induction principle and generalized Mendler-style iteration that allows a very clean representation of substitution for an untyped lambda calculus with explicit flattening, as an extended case study.

On the generic level, a notion of naturality adapted to generalized Mendler-style iteration is developed, and criteria for it established, in particular a map fusion theorem for the obtained iterative functions.

Concerning the case study, substitution is proven to fulfill two of the three monad laws, the third one only for “hereditarily canonical” terms, but this is rectified by a relativization of the whole construction to those terms. All the generic results and the case study have been fully formalized with the Coq system.

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, i.e., there is at least one datatype constructor that relates family members with different indices. Let $\kappa_0$ stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. Then, let $\kappa_1$ be the kind of type transformations, i.e., $\kappa_1 := \kappa_0 \rightarrow \kappa_0$. A typical example for a type transformation is $\text{List}$ of kind $\kappa_1$, where $\text{List} A$ is the type of finite lists with elements from type $A$. Therefore, $\text{List}$ itself is a family of datatypes that is indexed over all types. But $\text{List}$ is not a nested datatype since the recursive type equation for $\text{List}$, i.e., $\text{List} A = 1 + A \times \text{List} A$, does not relate lists with different indices.\(^1\) A simple example of a nested datatype where an invariant is guaranteed through

\(^1\) Families that are uniformly parameterised, such as $\text{List}$, are not excluded from our further treatment, but the efforts spent on covering nested datatypes in the rigorous sense would not be needed for those “degenerate” cases.
its definition are the powerlists [2] (or perfectly balanced, binary leaf trees [3]), with
recursive type 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). Clearly, this is only true
if we take the least solution of the recursive type equation. Throughout this article, we
will only consider least fixed points, i.e., our nested datatypes are \textit{inductive} families.

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, 5, 6].
The lambda terms where the names of the free variables are taken from \( A \) are given by
\( \text{Lam} A \), with recursive type equation

\[
\text{Lam} A = A + \text{Lam} A \times \text{Lam} A + \text{Lam}(\text{opt} A).
\]

The first summand gives the variables, the second represents application of lambda terms
and the interesting third summand stands for lambda abstraction. It uses the option type
\( \text{opt} A \) that has exactly one more element than \( A \), namely \( \text{None} \), while the injection of
\( A \) into \( \text{opt} A \) is called \( \text{Some} \). The idea is that an element of \( \text{Lam}(\text{opt} A) \) is seen as an
element of \( \text{Lam} A \) through lambda abstraction of the extra variable with the designated
name \( \text{None} \) in \( \text{opt} A \). Note that we do not assume that this variable occurs freely in the
body of the abstraction. The type \( A \) is only the name space for the variables, not the
set of names of variables that effectively occur, and it can be infinite and even any type
in \( \kappa_0 \), including types of the form \( \text{Lam} B \).

Programming with nested datatypes is possible in the functional programming lan-
guage Haskell (and Haskell is the language in which the example programs of [1] and
many other papers since then, up to the recent [7], are presented), but this article is
concerned with frameworks that guarantee termination of all expressible programs, such
as the Coq proof assistant [8] (for a textbook on Coq, see [9]) that is based on the
Calculus of Inductive Constructions (CIC), presented with details in [10], 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 \( \text{Lam} \) are fully supported with recursion
and induction principles. Although Coq is officially called a “proof assistant”, it is
already in itself\(^2\) a functional programming language. This is certainly not surprising
since it is based on an extension of polymorphic lambda calculus (system \( F\omega \)), although
the default type-theoretic system of Coq since version 8.0 is “pCIC”, the Predicative
Calculus of (Co)Inductive Constructions. System \( F\omega \) is also the framework of the article
with Abel and Uustalu [12] 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. We call a nested datatype
\textit{truly nested} (non-linear [13]) if the recursive type 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. In
other words, not only does the right-hand side of the recursive type equation refer to the
family name with a type argument \( B \) different from the type variable \( A \) on the left-hand
side, but the type expression \( B \) even mentions the family name itself.

\(^2\)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 [11].
Our example throughout this article is lambda terms with explicit flattening \[14\], with the recursive type equation

\[
\text{Lam} A = A + \text{Lam} A \times \text{Lam} A + \text{Lam}(\text{opt} A) + \text{Lam}(\text{Lam} A)
\]

The last summand qualifies \text{Lam} as truly nested datatype: \text{Lam} A is the type argument to \text{Lam}. It is clear that true nesting depends on the fact that we speak about families of types that are indexed over all (mono-)types and not just over all elements of a given type, such as the natural numbers. Only then, self-composition, like the informal \text{Lam} \circ \text{Lam} that is used in the last summand above, is possible, and it is even the smallest pattern for true nesting.

Even without termination guarantees, the algebra of programming \[15\] 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 \[12\] on terminating iteration (and coiteration) principles. Newer work by the author \[16\] integrates rank-2 Mendler-style iteration into CIC and also justifies an induction principle for the nested datatypes that have this iteration scheme. This is embodied in the system \text{LNMIt}, the “logic for natural Mendler-style iteration”, defined in Section 4.1. 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 informal recursive type 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. \(\simeq\)) terms.

The present article is concerned with an extension of \text{LNMIt} to a system \text{LNGMIt} that has generalized Mendler-style iteration \text{GMIt}, introduced in \[12\], in addition to plain Mendler-style iteration that is provided by \text{LNMIt}. Ordinary Mendler-style iteration does not allow a direct definition of the substitution operation on \text{Lam} while generalized Mendler-style iteration fits perfectly well for that purpose. Generalized Mendler-style iteration is a scheme encompassing generalized folds \[13, 3, 17\]. In particular, the efficient folds of \[17\] are demonstrated to be instances of \text{GMIt} in \[12\], and the relation to the \text{gfolds} of \[13\] is discussed there. Perhaps surprisingly, \text{GMIt} could be explained within \text{F\omega} through \text{MIT}. In a sense, this all boils down to the use of a syntactic form of right Kan extensions as the target constructor \(G^{\kappa_1}\) of the polymorphic iterative functions of type \(\forall A^{\kappa_0}.\mu F A \rightarrow GA\), where \(\mu F\) denotes the nested datatype \[12, \text{Section 4.3}\].

The main theorem of \[16\] is trivially carried over to the present setting, i.e., just by the Kan extension trick, the justification of \text{LNMIt} within CIC with impredicative universe \text{Set} \(=:\kappa_0\) and propositional proof irrelevance is carried over to \text{LNGMIt} (Theorem 3). Impredicativity of \(\kappa_0\) is needed since syntactic Kan extensions use impredicative means for \(\kappa_0\) in order to stay within \(\kappa_1\). However, \text{LNMIt} and \text{LNGMIt} are formulated as extensions of pCIC with its predicative \text{Set} as \(\kappa_0\).
The functions that are defined by a direct application of $\text{GMIt}$ are uniquely determined (up to pointwise propositional equality) by their recursive equation (Theorem 4), under a reasonable extensionality assumption. It is shown when these functions are themselves extensional (Theorem 5) and when they are “natural” (Theorems 6 and 7), and what natural has to mean for them (Definition 2), since they have types of the form

$$\forall A^\kappa \forall B^\kappa . (A \to HB) \to X A \to GB$$

for type transformations $X, H, G$. For the usual polymorphic function spaces described by $\forall A^\kappa . X A \to GA$, naturality is well-established, but for our situation, naturality does not seem to have been defined before.

By way of the example of lambda terms with explicit flattening—the truly nested datatype $Lam$—the merits of the general theorems about $LNGMIt$ will be studied, mainly by a representation of parallel substitution on $Lam$ 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 $Lam$ can be interpreted within the hereditarily canonical terms, and for those, parallel substitution is shown to give rise to 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 [18] and is based on [19].

The following section 2.1 presents notational conventions and repeats technical content from the introduction. Section 2.2 introduces to Mendler’s style of obtaining terminating recursive programs and develops the notions of free variables and renaming in the case study, leading to the question of naturality. Section 2.3 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, an appropriate notion of naturality is defined for the functions that are instances of the iteration scheme in $GMIt$. In Section 4.1, the already existing system $LNMIt$ with the logic for $MIt$ is properly defined, while Section 4.2 defines the new extension $LNGMIt$ as a logic for $GMIt$ and proves the general results mentioned above. This culminates in general criteria that guarantee naturality, 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 where the originally desired results are obtained for a variant of the truly nested datatype where all terms have to come with proof of hereditary canonicity. Section 6 concludes.

The present article is a thoroughly revised and extended version of the conference paper [20]. In particular, there are now proofs of all numbered lemmas and theorems, with the exception of Theorem 1 and Theorem 9, that belong to the realm of the case study and whose proof techniques do not seem necessary to present. Section 5 was rather sketchy in the conference version, and there was nothing related to the present section 5.3. Still, the reader is invited to consult the Coq code [18], in particular for the case study.

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 thoughtful advice that led to many changes in the presentation and also to new
material in the Coq scripts. In an early stage of the present results, I have benefitted from support by the European Union FP6 IST Coordination Action 510996 “Types for Proofs and Programs”.

2. Mendler-style Iteration

Mendler-style iteration schemes, originally proposed for positive inductive types [21], 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 [22]).

2.1. Notation

Here, we fix notation. This is not meant as an introduction to pCIC. The kinds we use for programming are $\kappa_0$ which stands for the universe of computationally relevant types (in Coq, this is $\text{Set}$), the kind $\kappa_1 := \kappa_0 \to \kappa_0$ of type transformations, and the kind $\kappa_2 := \kappa_1 \to \kappa_1$ of rank-2 type transformations. Types in $\kappa_0$ are denoted by $A$, $B$ and $C$. Type transformations in $\kappa_1$ are denoted by $X$, $Y$, $G$ and $H$. For variables instead of composite expressions, we use the same names $A,B,C$ and $X,Y,G,H$, respectively. $F$ will always stand for a rank-2 type transformation, i.e., an element of kind $\kappa_2$. With this naming convention, we will never have to give kinding annotations in the sequel. The universe $\kappa_0$ has the following constants and operations: the singleton type $1$, the unary operation $\text{opt}$ for the option type (with term constants $\text{None}$ and $\text{Some}$, as described in the introduction), the binary connectives $+$, $\times$ and $\to$ for disjoint sums, products and function spaces. On the term side, the left injection into a sum is denoted by $\text{inl}$ and the right injection by $\text{inr}$, the pair of $t_1$ and $t_2$ in a product by $(t_1,t_2)$ and the projections $\pi_1$ and $\pi_2$. Moreover, there is universal quantification over variables of kind $\kappa_0$. However, unless one assumes that $\kappa_0$ is impredicative, this leads out of $\kappa_0$. In pCIC, $\forall A.B$ has kind $\text{Type}$, which is a predicative universe (in fact, a cumulative hierarchy of universes). $\text{Type}$ is the universe of all types that can be assigned to terms, and it also has the function-space constructor $\to$. Universal quantification over variables of kind $\kappa_0$ or $\kappa_1$ does not lead out of $\text{Type}$.

We assume the type transformation $\text{List}$ with the usual semantics of homogeneous lists (the terms $[]$ and $a :: \ell$ will denote the empty list and the list consisting of the first element $a$ and rest $\ell$). The type transformation $\text{Lam}$ will not be introduced officially, but remarks will be made that refer to its intuitive semantics given in the introduction. We have $\lambda A.B$ as a type transformation ($A$ is a type variable, and $B$ is a type in $\kappa_0$). We also have $\lambda X.G$ as a rank-2 type transformation (with $X$ a variable in $\kappa_1$ and $F$ an expression in $\kappa_1$). Application is written as juxtaposition, and $XA$ is a type in $\kappa_0$, and $FX$ is a type transformation in $\kappa_1$.

We use $X \subseteq Y := \forall A.XA \to YA$ for any type transformations $X,Y$ as abbreviation for the respective polymorphic function space, hence $X \subseteq Y$ is a type in $\text{Type}$.

On the term level, we assume $\lambda$-abstraction over typed term variables and over variables of kind $\kappa_0$ and $\kappa_1$. We also assume application of terms $t$ to terms, types and type transformations, depending on $t$’s type.

Intensional/definitional equality is denoted by $\simeq$. In Coq, this is convertibility and not visible to the user. It acts algorithmically and cannot be inspected, let alone extended.
The type \((\lambda A.B)C\) is definitionally equal to \(B[A := C]\), denoting the capture-free substitution of type variable \(A\) by type \(C\) in type \(B\). Analogously for type transformations. Renaming of bound variables is even below the level of definitional equality: expressions are viewed up to \(\alpha\)-equivalence. The same rules hold for abstracted terms that receive arguments through application, and for the names of bound term variables.

Propositional/Leibniz equality is denoted by \(=\). Only terms \(t_1, t_2\) of definitionally equal type can be used to form the proposition \(t_1 = t_2\). Propositions have kind \(\text{Prop}\). The universe \(\text{Prop}\) is closed under universal quantification and thus impredicative. However, before Section 4, we only quantify propositions over variables that have a type in kind \(\kappa_0\) or variables \(A, B, C\) of kind \(\kappa_0\). On propositions, \(\rightarrow\) means implication.

2.2. Plain Mendler-style Iteration MIT

In order to fit the informal definition of \(\text{Lam}\), given in the introduction, into the setting of Mendler-style iteration, the notion of rank-2 functor is needed. Any constructor \(F\) of kind \(\kappa_2\) qualifies as rank-2 functor for the moment, and \(\mu F : \kappa_1\) denotes the generated family of datatypes. For our example, set

\[
LamF := \lambda X\lambda A. A + XA \times XA + X(\text{opt } A) + X(XA),
\]

which has kind \(\kappa_2\), and \(Lam := \mu LamF\). In general, there is just one datatype constructor for \(\mu F\), namely \(\text{in } F(\mu F) \subseteq \mu F\). For \(Lam\), more clarity comes from the four derived datatype constructors

\[
\begin{align*}
\text{var} & : \forall A. A \rightarrow \text{Lam } A, \\
\text{app} & : \forall A. \text{Lam } A \rightarrow \text{Lam } A \rightarrow \text{Lam } A, \\
\text{abs} & : \forall A. \text{Lam}(\text{opt } A) \rightarrow \text{Lam } A, \\
\text{flat} & : \forall A. \text{Lam}(\text{Lam } A) \rightarrow \text{Lam } A,
\end{align*}
\]

where, for example, \(\text{flat}\) is defined as \(\lambda A \lambda e. \text{Lam}(\text{Lam } A)\), \(\text{in } A (\text{inr } e)\), with right injection \(\text{inr}\) (here, we assume that \(+\) associates to the left), and the other datatype constructors are defined by the respective sequence of injections (see [14] or [12, Example 8.1]).\(^3\) From the explanations of \(\text{Lam}\) in the introduction, it is already clear that \(\text{var, app}\) and \(\text{abs}\) 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 \(\text{opt } A\) with type variable \(A\). The designated element \(\text{None}\) of \(\text{opt } A\) shall be the name for variable \(x_1\). Then, \(\lambda z. z x_1\) is represented by

\[
\text{abs(app (var None) (var (Some None)))},
\]

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

\(^3\)In Haskell 98, one would define \(\text{Lam}\) through the types of its datatype constructors whose names would be fixed already in the definition of \(\text{Lam}\). In Coq, one can do this for \(\text{Lam}\), but for \(\text{Lam}\), this will be rejected as being “non-strictly positive”.

6
In [4], a lambda-calculus interpretation of monad multiplication of $\text{Lam}$ is given that has the type of flat (with $\text{Lam}$ replaced by $\text{Lam}$), 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 the flat datatype constructor 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 [16], 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 $\forall A. \text{Lam} (\text{opt} (\text{opt} A))$, in order to accommodate the two free variables $x_1, x_2$. We instantiate the representation above for $\lambda z. z x_1$ by $\text{opt} A$ in place of $A$ and get a representation as term $t_1 : \text{Lam} (\text{opt} (\text{opt} A))$. $x_2$ is represented by

$$t_2 := \text{var} (\text{Some None}) : \text{Lam} (\text{opt} (\text{opt} A)) .$$

Now, $t$ shall be represented as the term

$$\text{flat} (\text{abs} t_3) : \text{Lam} (\text{opt} (\text{opt} A)) ,$$

hence with $t_3 : \text{Lam} (\text{opt} (\text{Lam} (\text{opt} (\text{opt} A))))$, defined as

$$t_3 := \text{app} \left( \text{app} (\text{var None}) (\text{var} (\text{Some} t_1)) \right) (\text{var} (\text{Some} t_2)) ,$$

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

Mendler-style iteration of rank 2 [12] can be described as follows: There is a constant

$$\text{Mit} : \forall G. (\forall X. X \subseteq G \rightarrow FX \subseteq G) \rightarrow \mu F \subseteq G$$

and the iteration rule

$$\text{Mit} G s A (\text{in} At) \simeq s (\mu F) (\text{Mit} G s) A t .$$

In a properly typed left-hand side – since in is of type $F(\mu F) \subseteq \mu F$ – term $t$ has type $F(\mu F)A$ and $s$ is of type

$$\forall X. X \subseteq G \rightarrow FX \subseteq G .$$

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 $\mu F$, to a function from $FX$ to $G$.

Our first example of an iterative function on $\text{Lam}$ is the function $\text{FV} : \text{Lam} \subseteq \text{List}$ that gives the list of the names of the free variables (with repetitions in case of multiple

\footnote{For a recursive removal of all explicit flattenings, in order to obtain results in the family $\text{Lam}$, see [16, Section 6].}
occurrences, thus \(FV\) is rather a projection from \(Lam\) to lists). We want to have the following definitional equations that describe the recursive behaviour (we mostly write type arguments as indices in the sequel):

\[
FV_A (\text{var}_A a) \simeq [a],
\]

\[
FV_A (\text{app}_A t_1 t_2) \simeq FV_A t_1 + FV_A t_2,
\]

\[
FV_A (\text{abs}_A r) \simeq \text{filterSome}_A (FV_{\text{opt}_A} r),
\]

\[
FV_A (\text{flat}_A e) \simeq \text{flat}_{\text{map}} FV_A (FV_{\text{Lam}_A} e).
\]

Here, we denoted by \([a]\) the singleton list that only has \(a\) as element and by \(+\) list concatenation. Moreover, \(\text{filterSome} : \forall A. \text{List}(\text{opt}_A) \to \text{List} A\) removes all the occurrences of \(\text{None}\) from its argument and also removes the injection \(\text{Some}\) from \(A\) into \(\text{opt}_A\) from the others. In symbols:

\[
\text{filterSome}_A [] \simeq []
\]

\[
\text{filterSome}_A (\text{None} :: \ell) \simeq \text{filterSome}_A \ell
\]

\[
\text{filterSome}_A (\text{Some} a :: \ell) \simeq a :: \text{filterSome}_A \ell
\]

The \(\text{abs}\) clause is interpreted as follows: the extra element \(\text{None}\) of \(\text{opt}_A\) is the variable name that is considered bound in \(\text{abs}_A r\), and that therefore all its occurrences have to be removed from the list of free variables.

The set of free variables of \(\text{flat}_A e\) is the union of the sets of free variables of the free variables of \(e\), which are still elements of \(\text{Lam}_A\). In terms of lists, this is expressed by concatenation of all the lists \(FV_A t\), in the order of appearance of the terms \(t\) in the list \(FV_{\text{Lam}_A} e\). This is achieved by the function

\[
\text{flat}_{\text{map}} : \forall A B. (A \to \text{List} B) \to \text{List} A \to \text{List} B,
\]

which is the “bind” operation of the list monad, and is defined by

\[
\text{flat}_{\text{map}}_{A,B} f [] \simeq []
\]

\[
\text{flat}_{\text{map}}_{A,B} f (a :: \ell) \simeq fa + \text{flat}_{\text{map}}_{A,B} f \ell
\]

In our example, we calculate

\[
FV_{\text{opt}} (\text{Lam}(\text{opt}(\text{opt}_A))) t_3 \simeq \text{None} :: \text{Some} t_1 :: \text{Some} t_2 :: []
\]

\[
FV_{\text{Lam}(\text{opt}(\text{opt}_A))} (\text{abs}_A t_3) \simeq t_1 :: t_2 :: []
\]

\[
FV_{\text{opt}(\text{opt}_A)} (\text{flat}(\text{abs}_A t_3)) \simeq FV_{\text{opt}(\text{opt}_A)} t_1 + FV_{\text{opt}(\text{opt}_A)} t_2 \simeq \text{None} :: \text{SomeNone} :: []
\]

However, the example does not show that variable names may occur repeatedly in the resulting list.

We now argue that there is such a function \(FV\), by showing that it is directly definable as \(MIt\ \text{List} s_{FV}\) for some closed term

\[
s_{FV} : \forall X. X \subseteq \text{List} \to \text{Lam}FX \subseteq \text{List},
\]

and therefore, we have the termination guarantee (in [12], a definition of \(MIt\) within \(F^w\) is given that respects the iteration rule even as reduction from left to right, hence

\[5\]We see that, for truly nested datatypes, nested recursive calls of functions appear quite naturally.
this is iteration as is the iteration over the Church numerals of which this is still a generalization). We will use an intuitive notion of pattern matching and even go beyond what Coq allows in giving names to the sequences of injections that correspond to \( \text{var, app, abs and flat} \):

\[
\begin{align*}
\text{var}^- a & := \text{inl} (\text{inl} (\text{inl} a)) \\
\text{app}^- t_1 t_2 & := \text{inl} (\text{inr} (t_1, t_2)) \\
\text{abs}^- r & := \text{inl} (\text{inr} r) \\
\text{flat}^- e & := \text{inr} e
\end{align*}
\]

We define \( s_{FV} := \lambda X \lambda t X \subseteq \text{List} X \lambda A \lambda t \text{LamF} X A \) match \( t \) with

\[
\begin{align*}
| \text{var}^- a^A & \mapsto [a] \\
| \text{app}^- t_1^X t_2^X A & \mapsto \text{it}_A t_1 + \text{it}_A t_2 \\
| \text{abs}^- t_1^X \text{opt} A & \mapsto \text{filterSome} (\text{it}_{\text{opt}} A r) \\
| \text{flat}^- e^X (X A) & \mapsto \text{flat_map} \text{it}_A (\text{it}_{X A} e)
\end{align*}
\]

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

The visible reason why Mendler’s style can guarantee termination without any syntactic descent (in which way can the flat-mapping over \( FV_A t \) be seen as “smaller”?\(^7\)) is the following: the recursive calls come in the form of uses of \( \text{it} \), which does not have type \( \text{LamF} \subseteq \text{List} \) but just \( X \subseteq \text{List} \), and the type arguments of the datatype constructors are replaced by variants that only mention \( X \) instead of \( \text{Lam} \). So, the definitions have to be uniform in that type transformation variable \( X \), but this is already sufficient to guarantee termination (for the rank-1 case of inductive \( \text{types} \), this has been discovered in [23] by syntactic means and, independently, by the author with a semantic construction [24]).

A first interesting question about the results of \( FV_A t \) is how they behave with respect to renaming of variables. First, define for any type transformation \( X \) the type of its \text{map term} as

\[\text{mon } X := \forall A \forall B. (A \to B) \to X A \to X B.\]

This is monotonicity of \( X \), expressed in logical terms (we never require syntactic positivity). Clearly, the name “map term” comes from the well-known case \( X := \text{List} \), with function \( \text{map} : \text{mon List} \) that maps its function argument over all the elements of its list argument. However, also the \text{renaming} operation \( \text{lam} \) will have a type of this form, more precisely, \( \text{lam} : \text{mon Lam} \), and \( \text{lam} f \) \text{t} has to represent \( t \) after renaming every free variable occurrence of name \( a \) in \( t \) by the variable of name \( f a \). It would be possible to define \( \text{lam} \) by help of \( \text{GMIIt} \) introduced in the next section, but it will automatically be available in the systems \( \text{LNMIIt} \) and \( \text{LNGMIIt} \) that will be described in Section 4. Therefore, we content ourselves in displaying its recursive behaviour (we omit the type arguments to \( \text{lam} \)):

\[
\begin{align*}
\text{lam } f \ (\text{var} \ A \ a) & \simeq \text{var}_B (f a) , \\
\text{lam } f \ (\text{app} \ A \ t_1 \ t_2) & \simeq \text{app}_B (\text{lam} f t_1) (\text{lam} f t_2) , \\
\text{lam } f \ (\text{abs} \ A \ r) & \simeq \text{abs}_B (\text{lam} (\text{opt_map} f) r) , \\
\text{lam } f \ (\text{flat} \ A \ e) & \simeq \text{flat}_B (\text{lam} (\text{lam} f) e) .
\end{align*}
\]

\(^6\)In Haskell 98, our specification of \( FV \), together with its type, can be used as a definition, but no termination guarantee is obtained.

\(^7\)See the discussion about “sized types” in the conclusion.
Here, in the third clause, yet another map term occurs, namely the canonical map \( \text{opt\_map} : \text{mon\_opt} \) that lifts any function of type \( A \rightarrow B \) to one of type \( \text{opt\_A} \rightarrow \text{opt\_B} \). Notice that \( \text{lams} \) is called with type arguments \( \text{opt\_A} \) and \( \text{opt\_B} \). In the final clause, the outer call to \( \text{lams} \) is with type arguments \( \text{Lam\_A} \) and \( \text{Lam\_B} \), while the inner one stays with \( A \) and \( B \). Thus, the free variables of \( e : \text{Lam}((\text{Lam\_A})) \), that have names in \( \text{Lam\_A} \), are renamed according to \( \text{lams} f : \text{Lam\_A} \rightarrow \text{Lam\_B} \), and the outermost datatype constructor is preserved, after appropriately changing its type argument.

We can now state the “interesting question”, mentioned before: can one prove

\[
\forall A \forall B \forall f A \rightarrow B. \forall t \text{Lam\_A}. \text{FV}_A (\text{lam}\_f t) = \text{map} f (\text{FV}_A t) ?
\]

This is an instance of the question for polymorphic functions \( h \) of type \( X \subseteq G \) whether they behave propositionally as a natural transformation from \( (X, m_X) \) to \( (G, m_G) \), given map functions \( m_X : \text{mon\_X} \) and \( m_G : \text{mon\_G} \). Here, the pair \( (X, m_X) \) is seen as a functor although no functor laws are required (for the moment). Being such a natural transformation means that the following holds, see also Figure 1 on page 14:

\[
\forall A \forall B \forall f A \rightarrow B \forall t X A. h_B (m_X A B f t) = m_G A B f (h_A t).
\]

The system \( \text{LNMIt} \), described in Section 4.1, allows to answer the above question by showing that \( \text{FV} \) is a natural transformation from \( (\text{Lam}, \text{lam}) \) to \( (\text{List}, \text{map}) \), i.e., where \( X := \text{Lam}, m_X := \text{lam}, G := \text{List}, m_G := \text{map} \) and \( h := \text{FV} \). This is in contrast to pure functional programming, where, following [25], naturality can come for free: in pure polymorphic lambda-calculus, when taking parametric equality in the law that describes naturality, the naturality property becomes a specific instance of parametricity that holds universally in that system. In intensional type theory such as our \( \text{LNMIt} \) and \( \text{LNGMIt} \) (see Section 4.2), naturality has to be proven on a case by case basis. Still, our Theorems 6 and 7 below give uniform naturality criteria for recursive functions that are defined by generalized Mendler-style iteration, independently of the nested datatype on which they are defined.

By (plain) Mendler-style iteration \( \text{MIt} \), one can also define a function \( \text{eval} : \text{Lam} \subseteq \text{Lam} \) that evaluates all the explicit flattenings and thus yields the representation of a usual lambda term [16]. In [16], also \( \text{eval} \) is seen in \( \text{LNMIt} \) to be a natural transformation, w. r. t. renaming for \( \text{Lam} \) and \( \text{Lam} \), respectively.

2.3. Generalized Mendler-style Iteration \( \text{GMIIt} \)

We would like to define a representation of substitution on \( \text{Lam} \). As for \( \text{Lam} \), the most elegant solution is to define a parallel substitution

\[
\text{subst} : \forall A \forall B. (A \rightarrow \text{Lam\_B}) \rightarrow \text{Lam\_A} \rightarrow \text{Lam\_B},
\]

where for a substitution rule \( f : A \rightarrow \text{Lam\_B} \), the term \( \text{subst}_{A,B} f t : \text{Lam\_B} \) is the result of substituting every variable of name \( a : A \) in the term representation \( t : \text{Lam\_A} \) by the term \( f a : \text{Lam\_B} \). The operation \( \text{subst} \) would then qualify as Kleisli extension operation of a monad in Kleisli form (a. k. a. bind operation in Haskell).8

---

8We say “would” since one of the monad laws cannot be established in \( \text{LNGMIt} \), see the remedy in Section 5.
Evidently, the desired type of subst is not of the form Lam \subseteq G for any G. However, it is logically equivalent to a type of this form, using the following definition:

\[ \text{Ran}_H G := \lambda A. \forall B. (A \rightarrow HB) \rightarrow GB \]

for any H, G. Here, we \( \lambda \)-abstract over a type in kind Type. This has not been covered by Section 2.1 on notation since it will only be used with impredicative \( \kappa_0 \) in the proof of Proposition 1. In pCIC, Ran\( _H \) G has kind \( \kappa_0 \rightarrow \text{Prop} \).

For our example of substitution, we could use Ran\( \text{Lam} \text{Lam} \), based on the following evident logical equivalence:

\[ (\forall A. \text{Lam} A \rightarrow (\text{Ran}\text{Lam}\text{Lam}) A) \Leftrightarrow (\forall A \forall B. (A \rightarrow \text{Lam} B) \rightarrow \text{Lam} A \rightarrow \text{Lam} B) \]

where only the universal quantification over B has to be moved across the implication Lam A \rightarrow \cdot and the two premisses Lam A and A \rightarrow Lam B are interchanged.

Ran\( _H \) G 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 [14], it was first used to justify termination of iteration schemes, and in [12], it served as justification of generalized Mendler-style iteration, to be defined next. Its motivation was better efficiency (it covers the efficient folds of [17], see [12]), 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 \( \kappa_0 \) because, only with impredicative \( \kappa_0 \), we have Ran\( _H \) G : \( \kappa_1 \). Hence, we will not use Ran\( _H \) G for programming and stay within pCIC.

The trick is to use the notion of relativized refined containment [12]: given X, H, G : \( \kappa_1 \), define the abbreviation

\[ X \leq_H G := \forall A \forall B. (A \rightarrow HB) \rightarrowXA \rightarrow GB. \]

Generalized Mendler-style iteration consists of a constant (the iterator)

\[ \text{GMIt} : \forall H \forall G. (\forall X. X \leq_H G \rightarrow FX \leq_H G) \rightarrow \mu F \leq_H G \]

and the generalized iteration rule

\[ \text{GMIt } H \ G \ s \ A \ B \ f \ (\text{in } At) \simeq s (\mu F) (\text{GMIt } H \ G \ s) \ A \ B \ f \ t. \]

As mentioned before, GMIt can again be justified within F\( ^\omega \), hence ensuring termination of the rewrite system underlying \( \simeq \).

Coming back to subst, we note that its desired type is Lam \subseteq Lam Lam, and in fact, we can define subst := GMIt Lam Lam \( s\text{subst} \) with

\[ s_{\text{subst}} : \forall X. X \leq_{Lam} \text{Lam} \rightarrow \text{Lam} \text{F } X \leq_{Lam} \text{Lam} \]

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

\[ \lambda X \lambda itX \leq_{Lam} \lambda A \lambda B \lambda fA \rightarrow_{Lam} B \lambda LamFA \rightarrow \lambda XA, \text{match } t \text{ with } \]

\[ | \text{var} - aA^3 \implies f a | \text{app} - itXA f tX ^A \implies \text{app} (it_{A,B} f t_1) (it_{A,B} f t_2) | \text{abs} - rX(\text{opt} A) \implies \text{abs} (it_{\text{opt}A, \text{opt}B} (\text{lift } f) r) | \text{flat} - cX(XA) \implies \text{flat} (it_{XA, \text{Lam} B} (\text{var} \text{Lam} B \circ (it_{A,B} f)) e) . \]

11
Here, we used an analogue of lifting for \textit{Lam} in [6],

\[
\text{lift} : \forall A \forall B. (A \rightarrow \text{Lam} B) \rightarrow \text{opt} A \rightarrow \text{Lam}(\text{opt} B)
\]
definable by pattern-matching, with properties

\[
\begin{align*}
\text{lift}_{A,B} f \text{ None} & \simeq \text{var}_{\text{opt} B} \text{ None} , \\
\text{lift}_{A,B} f (\text{Some} a) & \simeq \text{lam} \text{ Some} (fa),
\end{align*}
\]

where renaming \textit{lam} is essential.

Note that \textit{var}_{\text{Lam} B} \circ (\text{it}_{A,B} f) has type \(X A \rightarrow \text{Lam}(\text{Lam} B)\) (the infix operator \(\circ\) denotes composition of functions). From the point of view of clarity of the definition, we would have much preferred \(\text{flat} (\text{lam} (\text{it}_{A,B} f) e)\) to the term in the last clause of the definition of \(s_{\text{subst}}\). It would express that substitution is only carried out on the terms-as-variables in the argument \(e : \text{Lam}(\text{Lam} A)\) of \(\text{flat} e\), without touching the outer term structure. But that right-hand side would only type-check \textit{after} instantiating \(X\) with \(\text{Lam}\), hence generalized Mendler-style iteration cannot accept this alternative. However, one could rectify this by applying an extra hypothetical transformation \(j : X \subseteq \text{Lam}\) to \(e\), i.e., by taking \(\text{flat} (\text{lam} (\text{it}_{A,B} f) (j_{X,A} e))\) as right-hand side in the last case of the pattern-matching definition of \(s_{\text{subst}}\), assuming \(j\) would be instantiated by the polymorphic identity on \(\text{Lam}\) in an appropriately modified iteration rule. Having an extra \(j\) is the idea of Mendler-style recursion that was already present in the original article [21] (for positive inductive \textit{types} only). But for recursion, strong normalization is harder to establish than for iteration [26]. For non-generalized Mendler-style recursion, the author has given a logical system [27] in the spirit of system \(\text{LNMIt}\), defined in Section 4.1, but there does not yet exist a justification analogous to Theorem 3 for that system. Therefore, we stay with our definition of \(\text{subst}\) above that would be executable in pure higher-order polymorphic \(\lambda\)-calculus, as shown in [12].

Our definition satisfies

\[
\text{subst} f (\text{flat} e) \simeq \text{flat}(\text{subst}(\text{var} \circ (\text{subst} f)) e)
\]
to be seen immediately from the generalized iteration rule (assuming again proper \(\simeq\)-behaviour of pattern matching). Intuitively, this also only does the substitution according to substitution rule \(f\) on the terms-as-variables in the argument \(e\), but the original terms-as-variables of type \(\text{Lam} A\) are not only renamed into the resulting terms of type \(\text{Lam} B\), but they are substituted by the terms of type \(\text{Lam}(\text{Lam} B)\) that happen to be variables with those resulting terms as names.

Note that \(\text{subst} f (\text{var} a) \simeq fa\) is already the verification of the first of the three monad laws for the purported monad \((\text{Lam}, \text{var}, \text{subst})\) in Kleisli form (where \(\text{var}\) is the unit of the monad), and the other recursive rules are as expected:

\[
\begin{align*}
\text{subst} f (\text{app} t_1 t_2) & \simeq \text{app} (\text{subst} f t_1) (\text{subst} f t_2) , \\
\text{subst} f (\text{abs} r) & \simeq \text{abs} (\text{subst} (\text{lift} f) r).
\end{align*}
\]

The results beyond convertibility about \(\text{subst}\) are collected in the following theorem.

**Theorem 1.** In system \(\text{LNGMIt}\), to be defined later in this article, one can prove the following about the representation \(\text{subst}\) of substitution for lambda terms with explicit flattening, where we mean the universal (and well-typed) closure of all statements:
1. \((\forall a. \, fa = ga) \to subst \, ft = subst \, gt\)
2. \((\forall a. \, a \in FV \to fa = ga) \to subst \, ft = subst \, gt\)
3. \(lam \, g \, (subst \, ft) = subst \, ((lam \, g) \circ f) \, t\)
4. \(subst \, g \, (lam \, f \, t) = subst \, (g \circ f) \, t\)
5. \(subst \, g \, (subst \, ft) = subst \, ((subst \, g) \circ f) \, t\)
6. \(FV \, (subst \, ft) = flat\_map \, (FV \circ f) \, (FV \, t)\)

The first says that subst f only depends on the extension of f, the second refines this to the values of f on the names of the freely occurring variables (the proposition a ∈ ℓ for lists ℓ is defined by iteration over ℓ), the third and fourth are the two halves of naturality, defined in the next section (number 4 appears to be an instance of map fusion, as studied in [17]), the fifth is one of the other two monad laws, and the last one a means to express that FV is a monad morphism from Lam (that does not satisfy the last remaining monad law, i.e., subst varA(LamA) = t is not provable, see Section 5) to List. An easy consequence from it is \(b \in FV\,(subst \, ft) \to \exists a. \, a \in FV \, t \land b \in FV \,(fa)\), where we use existential quantification and conjunction in kind Prop. This consequence and the first five statements are all intuitively true for substitution, renaming and the set of free variables, and they were all known for Lam, hence without explicit flattening. The point here is that also the truly nested datatype Lam 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 ≃ is directly available, without any need for equational reasoning. For example, the term \(GMH \, H \, G \, s \, A \, B \, f \,(in \, A \, t)\) is definitionally equal to \(s \, Lam \,(GMH \, H \, G \, s) \, A \, B \, f \, t\) and can therefore be replaced by the latter anywhere, including in type expressions, and any implementation of the type-checker will do this silently – without any logical reasoning steps. Moreover, the proof of any equation \(t_1 = t_2\) is just by reflexivity of propositional equality in case \(t_1 \simeq t_2\). This is even the basis of proofs by reflection, see, e.g. [9]. Finally, any implementation will allow to compute a (typically unique) normal form with respect to ≃ of any expression, again without any extra guidance by the user.

3. Naturality for Generalized Maps

In order even to state an extension of the map fusion law of [17] to our situation, 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 \(Id_{\alpha_x}\). In this case, we omit the argument for \(H\) from \(X \leq_H G\) and only write \(X \leq G\). This is still a generalization of the type of map functions, since \((mon \, X) \simeq (X \leq X)\).

Assume a function \(h : X \subseteq G\) and map terms \(m_X : mon \, X\) and \(m_G : mon \, G\). Figure 1, which is strongly inspired by [14, Figure 1], recalls naturality, i.e., naturality of \(h\) w.r.t. \(m_X\) and \(m_G\) 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 \(hf\) in Figure 1 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 [28, Exercise 5 on page 19]. Its type is more concisely expressed as \(X \leq G\). The exercise in [28] (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 2 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 $h \, \text{id}_A$. In category theory, this is a simple exercise, but in our intensional setting, this allows to define naturality for any $X, G : \kappa_1, m_X : \text{mon} \, X, m_G : \text{mon} \, G$ and $h : X \subseteq G$.

**Definition1 (Naturality of $h : X \subseteq G$).** Given $X, G : \kappa_1, m_X : \text{mon} \, X, m_G : \text{mon} \, G$ and $h : X \subseteq G$, the functional $h$ is called natural with respect to $m_X$ and $m_G$ if it satisfies the following two laws:

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

Mac Lane’s exercise [28] can readily be extended to the generality of $X \subseteq_H G$, with arbitrary $H$, and a function $h : X \circ H \subseteq G$, but with less pleasing diagrams. We therefore only give an equational description of the parts we need for LNGMR.

**Definition2 (Naturality of $h : X \subseteq_H G$).** Given $X, H, G : \kappa_1$ and $h : X \subseteq_H G$, define the two parts of naturality of $h$ as follows: If $m_H : \text{mon} \, H$ and $m_G : \text{mon} \, G$, define the first part $\text{gnat}_1 \, m_H \, m_G \, h$ by

$$\forall A \forall B \forall C \forall f^A \to H \forall g^B \to C \forall x^A. m_G \, g \, (h_{A,B} \, f \, x) = h_{A,C} \, ((m_H \, g) \, \circ \, f) \, x .$$
Given Lemma 2. covered by the following lemma. If \( m \) and \( g \) are such that \( \text{gnat}_2 \text{subst} \) is natural with respect to \( m \) and \( g \) such that \( \text{gnat}_2 \text{lam subst} \) without the quantifiers, and the fourth item, \( \text{subst} (g \circ f) t \), is the unquantified \( \text{gnat}_2 \text{lam subst} \).

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

**Lemma 2.** Given \( X, H, G : \kappa_1 \), \( m_X : \text{mon} X \), \( m_H : \text{mon} H \), \( m_G : \text{mon} G \) and \( h : X \leq_H G \) such that \( \text{gnat}_2 \), \( m_H m_G \) and \( \text{gnat}_2 \) \( m_H \) \( m_G \) hold, the function

\[
h^\leq := \lambda A \lambda x (H(A)) \, h_{H,A,A} (\lambda y. y) \, x : X \circ H \subseteq G
\]

is natural with respect to \( m_X \star m_H \) and \( m_G \). Here, \( m_X \star m_H \) denotes the canonical map term for \( X \circ H \), obtained from \( m_X \) and \( m_H \), namely with \( (m_X \star m_H) \, f \, x \simeq m_X (m_H \, f) \, x \).

**Proof.** Assume types \( A, B \) and terms \( f : A \rightarrow B \) and \( x : X(HA) \). We have to show

\[
m_G \, f \, \left( h_A^\leq (x) \right) = h_B^\leq (m_X (m_H \, f) \, x) .
\]

The l. h. s. is \( m_G \, f \, \left( h_{H,A,A} (\lambda y. y) \, x \right) = h_{H,B,B} ((m_H \, f) \circ (\lambda y. y)) \, x \) by \( \text{gnat}_2 \). The r. h. s. is \( h_{H,B,B} ((\lambda y. y) \, (m_X (m_H \, f) \, x)) = h_{H,A,B} ((\lambda y. y) \circ (m_H \, f)) \, x \) by \( \text{gnat}_2 \). We conclude since we have the following convertibility: \( (m_H \, f) \circ (\lambda y. y) \simeq \lambda y. m_H \, f \, y \simeq (\lambda y. y) \circ (m_H \, f) \).

Thus, finally, one can define and argue about functions of type \( (\mu F) \circ H \subseteq G \) through \( (GMIlt) \subseteq \). For example, \( (\text{subst})^{\subseteq} : \text{Lam} \circ \text{Lam} \subseteq \text{Lam} \) would be the multiplication operation of the monad \( (\text{Lam}, \text{var}, \text{subst}) \) (but, as mentioned before, we will not be able to establish all the monad laws). Unlike \( \text{flat} \), this is now implicit flattening that carries out the flattening operation.

**4. Logic for Natural Generalized Mendler-style Iteration**

First, we reproduce the definition of \( LNMIlt \) from [16], then we extend it by \( GMIlt \) and its definitional rules in order to obtain its extension \( LNGMIlt \).

The following definitions should be readable on the basis of the notations of Section 2.1, in particular the definitions of \( \kappa_1, \kappa_2 \), \( \text{Type} \subseteq \) and \( \leq \) and the conventions on the kinds of \( A, G, X \). We frequently used the symbol \( \circ \) for function composition, and also the definition \( \text{mon} X := \forall A \forall B \forall f : A \rightarrow B \rightarrow X A \rightarrow X B \), which is a type of kind \( \text{Type} \).

First of all, we need to capture the concept of extensionality for map terms: for any \( X : \kappa_1 \) and map term \( m : \text{mon} \, X \), define the following proposition

\[
\text{ext} \, m := \forall A \forall B \forall f : A \rightarrow B \forall g : A \rightarrow B \, . \, (\forall a : A \, . \, f \, a = g \, a) \rightarrow \forall x : X A \, . \, m \, A \, B \, f \, r = m \, A \, B \, g \, r .
\]

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
4.1. LNMlt

does not hold in general, and we will incorporate it into the constructions where needed, so there is no need to impose it axiomatically.

We also formally represent naturality: given type transformations \(X, G: \kappa_1\), map terms \(m_F: \text{mon} X\) and \(m_G: \text{mon} G\) and a term \(h: X \subseteq G\), the proposition expressing \(h\)'s naturality w.r.t. \(m_X, m_G\) is

\[ h \in N(m_X, m_G) := \forall A \forall B \forall f^A \rightarrow B \forall x^{X^A}. h_B (m_X A B f t) = m_G A B f (h_A) \, . \]

4.1. LNMlt

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

\[ fct_1 m := \forall A \forall x^{X^A}. m AA (\lambda y.y) x = x \, , \]
\[ fct_2 m := \forall A \forall B \forall C \forall f^A \rightarrow B \forall g^B \rightarrow C \forall x^{X^A}. m AC (g \circ f) x = m BC g (m AB f x) \, . \]

Given a record \(e\) of type \(\text{ex} X\), Coq's notation for its field \(m = m e\), and likewise for the other fields. We adopt this notation instead of the more common \(e m\). Preservation of extensional\(^9\) functors by \(F\) is required in the form of a term of type \(\forall X. \text{ex} X \rightarrow \text{ex} (F X)\), and LNMlt is defined to be pCIC with \(\kappa_0 := \text{Set}\), extended by the constants and rules of Figure 3, adopted from [16]. It has to be admitted that this specification goes beyond the explained notions from Section 2.1: the type of \(\text{In}\) has kind Type since Type is even

\(^9\)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 mentioned above.
closed under universal quantification of variables of any type of kind $\text{Type}$ and under
adding a proposition as premise since $\text{Prop}$ is included in $\text{Type}$. The kind of $\mu \text{FInd}$’s
type is $\text{Prop}$ since $\text{Prop}$ is also closed under universal quantification over variables with
a complex type of kind $\text{Type}$ such as $\forall A, \mu F A \rightarrow \text{Prop}$ ($\mu F A$ also has kind $\text{Type}$ since
$\text{Set}$ is included in $\text{Type}$ and $\text{Prop}$ is of kind $\text{Type}$). The application of $\mathcal{P}$ of that type to
$A$ and $r : \mu F A$ is denoted by $\mathcal{P}_A r$, which is a proposition.

In $\text{LNMIt}$, one can show the following theorem [16, Theorem 3] about canonical
elements: There are terms $\text{ef}_\mu F : \exists \mu F$ and $\text{InCan} : F(\mu F) \subseteq \mu F$, the canonical
datatype constructor that constructs canonical elements, such that the following convertibilities hold:

\[
\begin{align*}
\text{map}_\mu F \circ \text{LNMIt} & \simeq \text{map}_\mu F, \\
\text{map}_\mu F \circ (\text{InCan} t) & \simeq \text{InCan}(m \circ \text{FpE} \circ \text{ef}_\mu F) \circ f \circ t, \\
\text{MIt} s \circ (\text{InCan} t) & \simeq s \circ (\text{MIt} s) \circ t.
\end{align*}
\]

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

This will now be related to the presentation in Section 2.2: the datatype constructor
$\text{In}$ is way more complicated than our previous $\text{in}$, but we get back $\text{in}$ in the form of
$\text{InCan}$. The map term $\text{map}_\mu F$ for $\mu F$, which does renaming in our example of $\text{Lam}$, as
demonstrated in Section 2.2, is an integral part of the system definition since it occurs
in the type of $\text{In}$. This is a form of simultaneous induction-recursion [29], where the
inductive definition of $\mu F$ is done simultaneously with the recursive definition of $\text{map}_\mu F$:
notice that the type $j \in \mathbb{N}(m \circ \text{ef}, \text{map}_\mu F)$ of the third term argument $n$ of $\text{In}$ is

\[
\forall A \forall B \forall f A \rightarrow B \forall \mu F X A . j B (m \circ \text{ef} A B f x) = \text{map}_\mu F A B f (j A x),
\]

hence this presents only conditions on $\text{map}_\mu F$ on $j$-images $j A x$ that are considered to have entered $\mu F$ “before” $\text{In} \circ j \circ n \circ t$, which is also the intuition behind the induction rule $\mu \text{FInd}$. Still, $\text{map}_\mu F$ enters even the type of $\text{In}$ and can therefore not be defined
“afterwards” by using $\text{MIt}$.

Notice also that the definitional rule for $\text{map}_\mu F$ is not even recursive, but only does
pattern-matching on the single datatype constructor $\text{In}$ of $\mu F$. So, $\text{LNMIt}$ does not use
the full power of simultaneous induction-recursion, but uses it in a very polymorphic
manner that is not captured by the foundational work such as [29].

The Mendler-style iterator $\text{MIt}$ 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 $\mu \text{FInd}$. Without access to the argument $n$ that assumes naturality
of $j$ as a transformation from $(X, m \circ \text{ef})$ to $(\mu F, \text{map}_\mu F)$, one would not be able to prove
naturality of $\text{MIt} s$, i.e., of iteratively defined functions on the nested datatype $\mu F$. The
author is not aware of ways how to avoid non-canonical elements and nevertheless have

\[10\]By taking the identity for $j$ in the second definitional rule of $\text{LNMIt}$, one obtains after $\beta$-reduction
an $\eta$-expansion of $(\text{MIt} s) A$, and this has to be remedied by the somewhat undesirable third definitional
rule of $\text{LNMIt}$. Since it is an $\eta$-like rule, one would rather not like to require it. In the Coq development,
this is avoided by defining $\text{MIt}$ by appropriate $\eta$-expansions of a preliminary constant of the same type.
an induction principle that allows to establish naturality of \( \text{MIt} \) s \([16, \text{Theorem 1}]\) (but see the final discussion in the conclusions).

The system \( \text{LNMIt} \) can be defined within CIC with impredicative \( \text{Set} \), extended by the principle of proof irrelevance, i.e., by \( \forall P : \text{Prop}\forall p_1^P \forall p_2^P, p_1 = p_2 \). This is the main result of [16], and it is based on an impredicative construction of simultaneous inductive-recursive definitions by Capretta \([30]\) that could be extended to work for this situation. It is also available \([19]\) in the form of a Coq module that allows to benefit from the evaluation of terms in Coq. For this, it is crucial that convertibility in \( \text{LNMIt} \) implies convertibility in that implementation.

The “functor” \( \text{LamF} \) is easily seen to fulfill the requirement of \( \text{LNMIt} \) to preserve extensional functors (using \([16, \text{Lemma 1 and Lemma 2}]\)). One can also directly program a term of type \( \forall X. \text{mon} X \rightarrow \text{mon} (\text{LamF} X) \) and verify that extensionality and both functor laws are preserved. The definition would be

\[
\lambda X \lambda m \text{mon} X \lambda A \lambda B \lambda f A \rightarrow B \lambda t \text{LamF} X A. \text{match} t \text{with} \begin{cases} 
\var{a}^{-} & \mapsto \var{a}^{-}(fa) \\
\text{app}^{-} t_X^A t_X^A & \mapsto \text{app}^{-}(m_{A,B} f t_1)(m_{A,B} f t_2) \\
\text{abs}^{-} r X^{(\text{opt} A)} & \mapsto \text{abs}^{-}\left(m_{\text{opt} A, \text{opt} B} (\text{opt}_\text{-map} f) r\right) \\
\text{flat}^{-} e X^A & \mapsto \text{flat}^{-}\left((m \ast m)_{A,B} f e\right)
\end{cases}
\]

with the operation \( \ast \) on map terms, described in Lemma 2. Also from this definition, one would immediately derive the recursive behaviour of \( \text{lam} \) that is shown near the end of Section 2.2. As mentioned there, \( \text{LNMIt} \) allows to prove that \( \text{FV} \in \mathcal{N}(\text{lam}, \text{map}) \), and this is an instance of \([16, \text{Theorem 1}]\).

4.2. \( \text{LNGMIt} \)

Let \( \text{LNGMIt} \) be the extension of \( \text{LNMIt} \) by the constant \( \text{GMIt} \) from Section 2.3 plus the two definitional rules of Figure 4. For completeness, we recall the abbreviation \( X \leq_H G := \forall A \forall B. \ (A \rightarrow HB) \rightarrowXA \rightarrow GB \) from Section 2.3. On the logical side, nothing changes with respect to \( \text{LNMIt} \).

Theorem \([16, \text{Theorem 3}]\) about \( \text{ef}_\ast \) and \( \text{InCan} \) for \( \text{LNMIt} \) immediately extends to \( \text{LNGMIt} \) and yields the following additional convertibility:

\[
\text{GMIt} s f (\text{InCan} t) \simeq s (\text{GMIt} s) f t ,
\]

which has this concise form only because of the \( \eta \)-like rule for \( \text{GMIt} \) that was made part of \( \text{LNGMIt} \) (the second rule in Figure 4). Thus, we get back the original behaviour of \( \text{GMIt} \) described in Section 2.3, but with the derived datatype constructor \( \text{InCan} \) instead of
of the defining datatype constructor in. For our case study, we are guaranteed that \( \text{lam} \) is extensional and satisfies the two functor laws—by the generic construction.

**Proposition 1.** The system LNGMIt can be defined within LNMIt if the universe \( \kappa_0 \) of computationally relevant types is impredicative.

**Proof.** The proof is nothing but the observation that the embedding of GMIt\( ^\kappa \) into Mit\( ^\kappa \) of [12, Section 4.3] extends for our situation of a rank-2 inductive constructor \( \mu F \) to non-canonical elements, i.e., the full datatype constructor \( \text{In} \) instead of only \( \text{in} \), considered in that work: define for \( H,G \):

\[
\text{LeqRan} := \lambda X \lambda h X \leq_H G \lambda A \lambda x X A \lambda B \lambda f A \rightarrow B f x ,
\]

\[
\text{RanLeq} := \lambda X \lambda h X \subseteq \text{Ran}_H G \lambda A \lambda B \lambda f A \rightarrow B f \lambda x X A . h A x B f .
\]

These terms establish the logical equivalence of \( X \leq_H G \) and \( X \subseteq \text{Ran}_H G \):

\[
\text{LeqRan} : \forall X. X \leq_H G \rightarrow X \subseteq \text{Ran}_H G ,
\]

\[
\text{RanLeq} : \forall X. X \subseteq \text{Ran}_H G \rightarrow X \leq_H G .
\]

Define for a step term \( s : \forall X. X \leq_H G \rightarrow FX \leq_H G \) for GMIt\( _{H,G} \) the step term \( s' \) for Mit\( _{\text{Ran}_H G} \) as follows:

\[
s' := \lambda X \lambda h X \subseteq \text{Ran}_H G . \text{LeqRan}_{FX} (s_X (\text{RanLeq}_X h)) .
\]

Then, we can define

\[
\text{GMIt}_{H,G} s := \text{RanLeq}_{\mu F} (\text{Mit}_{\text{Ran}_H G} s')
\]

and readily observe that the main definitional rule for GMIt in LNGMIt (the first rule in Figure 4) is inherited from that of Mit in LNMIt:

\[
\text{GMIt}_{H,G} s f (\text{In ef j n t}) \simeq \text{RanLeq}_{\mu F} (\text{Mit}_{\text{Ran}_H G} s') f (\text{In ef j n t})
\]

\[
\simeq \text{Mit}_{\text{Ran}_H G} s' (\text{In ef j n t}) f
\]

\[
\simeq s' (\lambda A . (\text{Mit}_{\text{Ran}_H G} s') A \circ j_A) t f
\]

\[
\simeq \text{LeqRan}_{FX} (s_X (\text{RanLeq}_X (\lambda A . (\text{Mit}_{\text{Ran}_H G} s') A \circ j_A))) t f
\]

\[
\simeq s_X (\text{RanLeq}_X (\lambda A . (\text{Mit}_{\text{Ran}_H G} s') A \circ j_A))) f t ,
\]

and the first term argument to \( s \) is then convertible with

\[
\lambda A \lambda B \lambda f A \rightarrow B f \lambda x X A . \text{Mit}_{\text{Ran}_H G} s' A (j_A x) B f
\]

and further with \( \lambda A \lambda B \lambda f A \rightarrow B f . (\text{RanLeq}_{\mu F} (\text{Mit}_{\text{Ran}_H G} s') A B f) \circ j_A . \)

The \( \eta \)-like rule for GMIt is immediate from the definition since \( \text{RanLeq}_{\mu F} h \beta \)-reduces for any term \( h \) to a term that has the following prefix \( \lambda A \lambda B \lambda f \lambda x . \) Therefore, \( \eta \)-expansion with those four variables can be eliminated just by \( \beta \)-reduction.\(^{11}\) Impredicativity of \( \kappa_0 \) is needed to have \( \text{Ran}_H G : \kappa_1 \), as mentioned in Section 2.3.

\(^{11}\)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.
Theorem 3. The system LNGMI \textit{It} can be defined within CIC with impredicative Set, extended by the principle of propositional proof irrelevance, i. e., by admitting the axiom \( \forall P : \text{Prop} \forall p_1 \forall p_2 : p_1 = p_2 \).

\textbf{Proof.} Use the previous proposition and the main theorem of [16] that states the same property of LNM\textit{It}.

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

Theorem 4 (Uniqueness of GM\textit{It}). Assume type transformations \( H, G : \kappa_1 \) and terms \( s : \forall X. X \leq_H G \rightarrow F X \leq_H G \) and \( h : \mu F \leq_H G \) (the candidate for being GM\textit{It}s). Assume further the following extensionality property of \( s \) (only depends on the extension of its first function argument, but in a way adapted to the parameter \( f \)):
\[ \forall X \forall g, h : X \leq_H G. (\forall A \forall B \forall f^{A \rightarrow H B} \forall x^{A X}. g f x = h f x) \rightarrow \forall A \forall B \forall f^{A \rightarrow H B} \forall y^{F X A}. s g f y = s h f y . \]
Assume finally that \( h \) satisfies the equation for GM\textit{It}s:
\[ \forall X e f X X j \forall X \forall e f X \forall H \forall j \subseteq X (e f, \text{map}_x, e) \forall A \forall B \forall f^{A \rightarrow H B} \forall x^{F X A}. h A B f (\text{In ef j n t}) = s (\lambda A \lambda B \lambda f^{A \rightarrow H B}. (h A B f) \circ j A) f t . \]
Then, \( \forall A \forall B \forall f^{A \rightarrow H B} \forall \mu F A. h A B f \circ r = (\text{GMIt s}) A B f r . \)

\textbf{Proof.} By the induction principle \( \mu F \text{Ind} \), as for [16, Theorem 2]. It seems nevertheless appropriate to show the argument. The induction principle is used with the property
\[ P := \lambda A \lambda B A \forall \forall B \forall f^{A \rightarrow H B}. h A B f \circ r = (\text{GMIt s}) A B f r , \]
where the quantification of the parameters \( B \) and \( f \) is compulsory already for typing purposes. Then assume the appropriate \( X, e f, j, n \). The inductive hypothesis is
\[ \forall A \forall x^{A X} \forall B \forall f^{A \rightarrow H B}. h A B f (j A x) = (\text{GMIt s}) A B f (j A x) . \]
We assume further \( A, B, f^{A \rightarrow H B}, t^{F X A} \) and have to show
\[ h A B f (\text{In ef j n t}) = (\text{GMIt s}) A B f (\text{In ef j n t}) . \]
Applying the equational hypothesis on \( h \) and the computation rule for GM\textit{It} yields the following equivalent equation:
\[ s (\lambda A \lambda B \lambda f^{A \rightarrow H B}. (h A B f) \circ j A) f t = s (\lambda A \lambda B \lambda f^{A \rightarrow H B}. ((\text{GMIt s}) A B f) \circ j A) f t . \]
The extensionality assumption on \( s \) finishes the proof if we can show
\[ \forall A \forall B \forall f^{A \rightarrow H B} \forall x^{A X}. ((h A B f) \circ j A) x = (((\text{GMIt s}) A B f) \circ j A) x , \]
but this is, up to the order of quantifiers, convertible with the induction hypothesis. □

Given type transformations \( X, H, G \), the type \( X \leq_H G \) has an embedded function space, so there is the natural question whether an inhabitant \( h \) of \( X \leq_H G \) only depends on the extension of this function parameter. This is expressed by the proposition
\[ \forall x^{h : X \rightarrow H B}. (\forall a^{A}. f a = g a) \rightarrow \forall x^{A X}. h A B f r \circ r = h A B g r . \]
The name \textit{gext} stands for \textit{generalized extensionality.} The earlier definition of \textit{ext} is the special instance where \(X\) and \(G\) coincide and where \(H\) is the identity type transformation 

\[ I_d \_ \_ \_ \_ \_ := \lambda A.A. \]

Given type transformations \(H,G\) and a term \(s : \forall X. X \leq H G \rightarrow FX \leq H G\), we say that \(s\) \textit{preserves extensionality if} \(\forall X \forall hX \leq n^G. \text{gext} h \rightarrow \text{gext} (sh)\) holds.

The following statement has no precursor in \textit{LMNIt.}

**Theorem 5 (Extensionality of \textit{GMIt}s).** Assume type transformations \(H, G\) and a term \(s : \forall X. X \leq H G \rightarrow FX \leq H G\) that preserves extensionality in the above sense. Then \(\text{GMIt}s : \mu F \leq H G\) is extensional, i.e., \(\text{gext}(\text{GMIt}s)\) holds.

**Proof.** An easy application of \(\mu\text{Fin}d\). The predicate we need is

\[ P := \lambda A \exists \nu F \forall A \forall B \forall f g \, A^\rightarrow H B \rightarrow (\forall a^A. fa = ga) \rightarrow (\text{GMIt}s)_{A,B} f r = (\text{GMIt}s)_{A,B} g r . \]

Then assume the appropriate \(X, ef, j, n\) and the inductive hypothesis \(\forall A \forall X A, P_A(j_A \, x)\). Given \(A, i^{FXA}\), we want to show \(P_A(\text{In} \, ef \, j \, n \, t)\). So, assume \(B, g, g^\rightarrow H B\) such that \(\forall a^A. fa = ga\). Our aim is to show \((\text{GMIt}s)_{A,B} f (\text{In} \, ef \, j \, n \, t) = (\text{GMIt}s)_{A,B} g (\text{In} \, ef \, j \, n \, t)\).

Both sides are convertible by help of the first rule for \(\text{GMIt}\). Since \(s\) preserves extensionality, it suffices to show generalized extensionality for the common first term argument of \(s\) in both reducts, i.e., \(\text{gext}(\lambda A \exists \nu f A^\rightarrow H B) \cdot ((\text{GMIt}s)_{A,B} f) \circ j_A\). Up to order of quantifiers and convertibility, this is just the induction hypothesis.

Coming back to the representation \(\text{subst}\) of substitution on \(\text{Lam}\) from Section 2.3, straightforward reasoning shows that \(\text{subst}\) preserves extensionality, hence Theorem 5 yields \(\text{gext subst}\), which proves the first item of Theorem 1. Its refinement, namely the second item of Theorem 1,

\[ (\forall a^A. a \in \text{FV} \, t \rightarrow fa = ga) \rightarrow \text{subst} \, f \, t = \text{subst} \, g \, t , \]

needs a direct proof by the induction principle \(\mu\text{Fin}d\), where the behaviour of \(\text{FV}\) on non-canonical elements plays an important role, but is nevertheless elementary.

**Theorem 6 (First part of naturality of \textit{GMIt}s).** Given \(H, G : \kappa_1\), map terms \(m_H : \text{mon} \, H, m_G : \text{mon} \, G\) and a term \(s : \forall X. X \leq H G \rightarrow FX \leq H G\) that preserves extensionality. Assume further

\[ \forall X \forall hX \leq n^G \, X \rightarrow \text{gext} h \rightarrow \text{gnat}_1 \, m_H \, m_G \, h \rightarrow \text{gnat}_1 \, m_H \, m_G \, (sh) . \]

Then, \(\text{GMIt}s\) satisfies the first part of naturality, i.e., \(\text{gnat}_1 \, m_H \, m_G \, (\text{GMIt} s)\).

**Proof.** Induction with \(\mu\text{Fin}d\) for the predicate

\[ \lambda A \lambda r^A F \forall B \forall C \forall f A^\rightarrow H B \forall g B^\rightarrow C. m_G g ((\text{GMIt}s)_{A,B} f \, r) = (\text{GMIt}s)_{A,C} ((m_H g) \circ f) \, r . \]

As usual, assume the appropriate \(X, ef, j, n, A, t, B, C, f, g\). We have to show

\[ m_G g ((\text{GMIt}s)_{A,B} f (\text{In} \, ef \, j \, n \, t)) = (\text{GMIt}s)_{A,C} ((m_H g) \circ f) (\text{In} \, ef \, j \, n \, t) . \]

We want to use Theorem 5 for the function \(h : X \leq H G,\) defined by

\[ h := \lambda A \lambda B \lambda f A^\rightarrow H B . ((\text{GMIt}s)_{A,B} f) \circ j_A , \]
which represents the recursive calls in the right-hand side of the rule for GMIt in the
definition of LNGMIt. Generalized extensionality of h follows straightforwardly without
any assumptions on j from \text{gext}(GMIt s), which comes from Theorem 5 that uses our
assumption that s preserves extensionality.

Our original goal is convertible by the first rule for GMIt with
\[ m_G \ g (s \ h \ t) = s \ h ((m_H \ g) \circ f) \ t \]
hence we only have to show \( gnat_I m_H m_G (s \ h) \). The main assumption of the theorem
is made for that: \( ef \) has type \( \mathcal{E} X \), \text{gext} h has been shown above, and the induction
hypothesis provides \( gnat_I m_H m_G h \). \hfill \Box

The proof did not use the naturality of argument \( j \), provided by the context of the
induction step.

As an instance of this theorem, one can prove the third item in Theorem 1.

\textbf{Theorem 7 (Second part of naturality of GMIt s—map fusion).} Given \( H, G : \kappa_1 \)
and a term \( s : \forall \ X \ \mathcal{X} \ \xi \leq_H G \rightarrow \mathcal{X} \mathcal{X} \ \xi \leq_H G \) that preserves extensionality. Assume further
\( \forall \ X \forall h X \leq_H G \forall \xi \mathcal{E} X \ \text{gext} h \rightarrow gnat_2 (m \ ef) \ h \rightarrow gnat_2 (m (Fp \mathcal{E} ef)) (s \ h) \).

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

\textbf{Proof.} Induction with \( \mu \text{FInd} \), quite analogously to the previous proof. The induction
predicate is
\[ \lambda A \lambda r \mu FA \forall B \forall C \forall f A \rightarrow B \forall g B \rightarrow HC \ (GMIt s)_{B,C} g (\text{map}_{\mu F} f \ r) = (GMIt s)_{A,C} (g \circ f) \ r . \]
As before, assume the appropriate \( X, ef, j, n, A, t, B, C, f, g \). We have to show
\[ (GMIt s)_{B,C} g (\text{map}_{\mu F} f (\ln ef j n t)) = (GMIt s)_{A,C} (g \circ f) (\ln ef j n t) . \]
The left-hand side enjoys a rule application for \( \text{map}_{\mu F} \) and is thus convertible with
\[ (GMIt s)_{B,C} g (\ln ef j n (m (Fp \mathcal{E} ef) f t)) . \]
We reuse the function \( h \) that occurred in the proof of the previous theorem. Again,
thanks to preservation of extensionality by s, we can use Theorem 5 and infer even \text{gext} h. We can now apply the computation rule for GMIt on both sides and arrive at the convertible proposition
\[ s \ h \ g (m (Fp \mathcal{E} ef) f t) = s \ h (g \circ f) \ t , \]
hence we have to show \( gnat_2 (m (Fp \mathcal{E} ef)) (s \ h) \), but here, the main assumption of the
theorem applies if we can show \( gnat_2 (m \ ef) \ h \). We now crucially need naturality of \( j \)
that comes as assumption \( n \) of \( j \in \mathcal{N}(m \ ef, \text{map}_{\mu F}) \) with the induction principle. Assume
\( A, B, C, f, g, x \) and calculate
\[ h_{B,C} g (m \ ef \ x) \simeq (GMIt s)_{B,C} g (j_B (m \ ef \ f \ x)) = (GMIt s)_{B,C} g (\text{map}_{\mu F} f (j_A \ x)) = (GMIt s)_{A,C} (g \circ f) (j_A \ x) \simeq h_{A,C} (g \circ f) \ x . \]
The first = uses naturality of \( j \), the second applies the induction hypothesis. \hfill \Box
Although the proof is rather simple, this is the main point of the complicated system $\text{LNGMIt}$ with its inductive-recursive nature: ensure naturality to be available for $j$ inside the inductive step of reasoning on $\mu F$. One might wonder whether this theorem could be an instance of [16, Theorem 1], using the definition of $\text{GMI}t$ in Proposition 1 for impredicative $\kappa_0$. 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 $\text{Ran}_G$ in the construction of Proposition 1 introduces values of function type.

As an instance of this theorem, one can prove the fourth item of Theorem 1. By the way, Lemma 2 then yields that $(\text{subst})^\subseteq : \text{Lam} \circ \text{Lam} \subseteq \text{Lam}$ is natural with respect to $\text{lam} \ast \text{lam}$ and $\text{lam}$. The fifth item (the interchange law for substitution that is one of the monad laws) can now be proven by the induction principle $\mu \text{FInd}$, using extensionality and both parts of naturality (hence, the items 1, 3 and 4 that are based on Theorem 5, Theorem 6 and Theorem 7) in the case for the representation of lambda abstraction (recall that $\text{lift}$ is defined by help of $\text{lam}$).

5. Completion of the Case Study on Substitution

The last item of Theorem 1 on properties of $\text{subst}$ can be proven by the induction principle $\mu \text{FInd}$ without any results about $\text{Lam}$, just with several preparations about lists, also using naturality of $\text{FV}$ in the proof of the case for the representation of lambda abstraction. Thus, the proof of Theorem 1 can be considered as finished.

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

$$\forall A \forall \text{Lam}^A. \text{subst} \text{var}_A t = t,$$

which has an $\eta$-like flavour since $\text{subst} \text{var}_A$ must then have any term representation $t$ as image. We called every term of the form $\text{InCan} t$ with $t : F(\mu F)A$ a canonical term in $\mu FA$: this makes the terms of the form $\text{app} t_1 t_2$ or $\text{abs} r$ or $\text{flate} c$ canonical terms that govern the results of substitution in all but the variables case, as can be seen from the definition of $\text{s}_{\text{subst}}$. Because of the presence of non-canonical terms in $\text{LNGMIt}$, we therefore cannot hope to prove the last monad law for all terms. And we cannot even hope to do so only for the canonical terms in the family $\text{Lam}$ since this notion is not recursively applied to the subterms.

The following is an ad hoc notion for our example. For the truly nested datatype $\text{Bush}$ of “bushes” with $\text{Bush} A = 1 + A \times \text{Bush}(\text{Bush} A)$, a similar notion has been studied by the author in [16, 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 $\text{Lam}$, the predicate $\text{can} : \forall A. \text{Lam} A \rightarrow \text{Prop}$, inductively by the following four closure rules:

- $\forall A \forall a^A. \text{can} (\text{var} a)$
- $\forall A \forall t_1^\text{Lam} A. \forall t_2^\text{Lam} A. \text{can} t_1 \rightarrow \text{can} t_2 \rightarrow \text{can}(\text{app} t_1 t_2)$
- $\forall A \forall r^\text{Lam} (\text{opt} A). \text{can} r \rightarrow \text{can}(\text{abs} r)$
∀A∀e.Lam(Lam A). can e → (∀t.Lam A.t ∈ FV e → can t) → can (flat e).

Hence, can is closed under all the term formation rules of Lam, except for flat, where the extra assumption for hereditary canonicity of flat e is that the names of the free variables of e : Lam(Lam A) are already hereditarily canonical terms in Lam A. This definition is strictly positive and, formally, infinitely branching. However, there are always only finitely many t that satisfy t ∈ FV 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, except for the example proof of Lemma 8 below, are not considered to be of real interest for this article. We will only record which former results enter these proofs.

Note 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 recursively since the last clause of the definition of can refers to them at a negative position. But the flat case of FV would work out well since we would be allowed to assume that FV_{Lam A} e and FV_A t would have been defined before for every t ∈ FV_{Lam A} e, thus ensuring well-definedness of flat_map FV_A (FV_{Lam A} e).

Here comes a digression on the notion of hereditarily canonical terms that is an answer to a very interesting question by one of the referees. Lemma 8 continues with the main line of thought.

The problem with can is that it is not generically derivable from LamF. The definition suggested by the referee takes as an additional argument a predicate that should be fulfilled by the names of all free variables:

\[\text{can}_2 : \forall A. (A \rightarrow \text{Prop}) \rightarrow \text{Lam A} \rightarrow \text{Prop}\]

is inductively defined by the following closure rules:

* \[\forall A\forall P_A \rightarrow \text{Prop}.A.a \rightarrow \text{can}_2 P (\text{var} a)\]

* \[\forall A\forall P_A \rightarrow \text{Prop}.\forall t_1 \text{Lam A}. \forall t_2 \text{Lam A}. \text{can}_2 P t_1 \rightarrow \text{can}_2 P t_2 \rightarrow \text{can}_2 P (\text{app} t_1 t_2)\]

* \[\forall A\forall P_A \rightarrow \text{Prop}.\forall t_1 \text{Lam A}. \forall t_2 \text{Lam A}. \text{can}_2 (\text{optpred} P) r \rightarrow \text{can}_2 P (\text{abs} r)\]

* \[\forall A\forall P_A \rightarrow \text{Prop}.\forall e \text{Lam (Lam A)}. \text{can}_2 (\text{can}_2 P) e \rightarrow \text{can}_2 P (\text{flat} e)\]

Here optpred P denotes the obvious lifting of P to a predicate on opt A, which holds on None and is Pa on Some a.

This is a truly nested inductive definition of a family of predicates: note the change of parameter P that involves can2 itself in the last closure rule. Therefore, can2 is not admitted as an inductive definition in CIC, and Coq will refuse it, just as Lam itself, if it were defined as an inductive datatype of Coq. However, one can study can2 axiomatically: just define constants corresponding to the closure rules. An induction principle is also easy to assume as an extra constant: it expresses that can2 is minimal among all terms of its type (with respect to pointwise implication) that satisfy these closure rules. However, these are only axioms without any consistency guarantee by Coq. A variant can2’ of the
same type as \( \text{can}_2 \) can be defined in CIC. It has the same first three closure rules (just with \( \text{can}'_2 \) instead of \( \text{can}_2 \)), but the last rule is as follows:

\[
\forall A \forall P A \rightarrow \text{Prop} \forall e \text{Lam}(\text{Lam} A). \text{can}'_2 U e \rightarrow (\forall t \text{Lam} A. \ t \in \text{FV} e \rightarrow \text{can}_2' P t) \rightarrow \text{can}'_2 P (\text{flat} e)
\]

Here, \( U \) denotes the “universal” predicate that is always true. This variant is very close to the original \( \text{can} \), and it is again not generic. It is easy to see that \( \text{can}'_2 P t \) implies \( \text{can} t \) and that \( \text{can} t \) implies \( \text{can}'_2 U t \), again with \( U \) the universal predicate.

By the help of some extra lemmas, one can show that \( \text{can}_2 P t \) and \( \text{can}'_2 P t \) are logically equivalent propositions for all \( P, t \). However, for both directions, the induction principle for \( \text{can}_2 \) enters, and that has only been assumed. The details are to be found in the Coq development [18].

5.1. Results for Hereditarily Canonical Terms

The relativization of the missing monad law to hereditarily canonical terms is true:

**Lemma 8.** \( \forall A \forall t \text{Lam} A. \text{can} t \rightarrow \text{subst var}_A t = t \).

**Proof.** Induction on \( \text{can} t \). The variable case even holds with convertibility:

\[
\text{subst var}_A (\text{var} A a) \simeq \text{var}_A a
\]

Case \( \text{app} A t_1 t_2 \): by induction hypothesis, \( \text{subst var}_A t_i = t_i \) for \( i = 1, 2 \). One concludes by observing

\[
\text{subst var}_A (\text{app} A t_1 t_2) \simeq \text{app}_A (\text{subst var}_A t_1) (\text{subst var}_A t_2)
\]

Case \( \text{abs} A r \): by induction hypothesis, \( \text{subst var}_A r = r \). We calculate

\[
\text{subst var}_A (\text{abs} A r) \simeq \text{abs}_A (\text{subst} (\text{lift var}_A) r)
\]

We conclude from extensionality of \( \text{subst} \) (item 1 in Theorem 1) since case analysis on \( \text{opt} A \) shows: \( \forall t \text{opt} A. \ \text{lift var}_A a = \text{var}_A \text{opt} A a \).

Case \( \text{flat} A e \): by induction hypothesis, \( \text{subst var}_A \text{Lam} A e = e \). We know

\[
\text{subst var}_A (\text{flat} A e) \simeq \text{flat}_A (\text{subst} (\text{var}_A \text{Lam} A \circ (\text{subst var}_A)) e),
\]

which allows us to conclude by refined extensionality of \( \text{subst} \) (property number 2 of Theorem 1), if we can show

\[
\forall t \text{Lam} A. t \in \text{FV} \text{Lam} A e \rightarrow (\text{var}_A \text{Lam} A \circ (\text{subst var}_A)) t = \text{var}_A \text{Lam} A t
\]

but for those \( t \), that have to be hereditarily canonical in order to make \( \text{flat} A e \) hereditarily canonical, the induction hypothesis provides \( \text{subst var}_A t = t \).

Another result that cannot be proven in general is a refinement of extensionality of \( \text{lam} f t \) in its function argument \( f \) in considering the “renaming rule” \( f \) only on the free variable names of \( t \). We are only able to show the following relativization by induction on the predicate \( \text{can} \) and elementary reasoning about free variable names:

\[
\forall A \forall f, g A \rightarrow B \forall t \text{Lam} A. \text{can} t \rightarrow (\forall a. a \in \text{FV} t \rightarrow f a = g a) \rightarrow \text{lam} f t = \text{lam} g t
\]
It seems that the relativization to \textit{can} is necessary since \textit{lam}, i.e., \textit{map}_{\mu F} for our fixed \( F := \text{LamF} \), is too deeply integrated into the system \textit{LNGMIt} to be amenable to further reasoning that would go beyond the intended and established fact that \textit{map}_{\mu F} is an extensional functor.

We now address extra closure properties of \textit{can}. Renaming \textit{lam} preserves hereditary canonicity:

\[
\forall A \forall B \forall f A \rightarrow B. \forall t. \text{Lam}(A) \rightarrow \text{can}(\text{Lam}(B)).
\]

This is proven by induction on \textit{can}, and the crucial \textit{flat} case needs the following identification of free variables of \textit{lam}:

\[
\forall A \forall B \forall f A \rightarrow B. \forall a. a \in \text{FV}(t) \land b = fa.
\]

which is nearly an immediate consequence of naturality of \textit{FV}.

Analogously, \textit{subst} preserves hereditary canonicity:

\[
\forall A \forall B \forall f A \rightarrow \text{Lam}(B) \forall t. \text{Lam}(A). (\forall a. a \in \text{FV}(t) \rightarrow \text{can}(fa)) \rightarrow \text{can}(t) \rightarrow \text{can}(\text{subst}(f)t).
\]

Again, this is proven by induction on \textit{can}, and again, the crucial case is with \textit{flat} for which free variables of \textit{subst} have to be identified, but this has already been mentioned as a consequence of property number 6 of Theorem 1.

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

\[
\forall A \forall B \forall f A \rightarrow B. \forall t. \text{can}(t) \rightarrow \text{lam}(t) = \text{subst}(\text{var}(B) \circ f)(t).
\]

From this, evidently, we get the more perspicuous equation for \textit{subst} (\textit{flat} e), discussed on page 12, but only for hereditarily canonical \textit{e} and only with propositional equality:

\[
\forall A \forall B \forall f A \rightarrow B \forall t. \text{can}(t). \forall e. \text{can}(t) \rightarrow \text{can}(\text{subst}(f)(t) = \text{flat}(\text{lam}(\text{subst}(f))e).
\]

5.2. Hereditarily Canonical Terms as a Nested Datatype

Define \( \text{Lam}' := \lambda A. \{ t : \text{Lam}(A) \mid \text{can} t \} : \kappa_1 \). The set comprehension notation stands for the inductively defined \textit{sig} of Coq (definable within pCIC, hence within \textit{LNGMIt}) which is a strong sum in the sense that the first projection \( \pi_1 : \text{Lam}' \subseteq \text{Lam} \) yields the element \text{t} and the second projection the proof of \text{can} \text{t}. Less Coq-specifically, one can describe \text{Lam}' by a non-recursive inductive definition, with single defining clause

\[
\forall A \forall t. \text{Lam}(A). \text{can} t \rightarrow \text{Lam}' A,
\]

and \( \pi_1 \) can be defined by pattern-matching on this construction.

Thus, we encapsulate hereditary canonicity already in the family \text{Lam}''. We will present \text{Lam}'' as a truly nested datatype, but not one that comes as a \( \mu F \) from \textit{LNGMIt}.

Since \textit{can} follows the term structure in the cases other than for \textit{flat}, it is quite trivial to define datatype constructors

\[
\begin{align*}
\text{var}' & : \forall A. A \rightarrow \text{Lam}' A, \\
\text{app}' & : \forall A. \text{Lam}' A \rightarrow \text{Lam}' A \rightarrow \text{Lam}' A, \\
\text{abs}' & : \forall A. \text{Lam}'(\text{opt} A) \rightarrow \text{Lam}' A
\end{align*}
\]
from their analogues in \textit{Lam}. The construction of
\[
\text{\texttt{\textit{flat}' : } \forall A. \mathit{Lam}'(\mathit{Lam}' A) \rightarrow \mathit{Lam}' A}
\]
is as follows: Assume \(e : \mathit{Lam}'(\mathit{Lam}' A)\). Then, its first projection, \(\pi_1 e\), is hereditarily canonical and of type \(\mathit{Lam}(\mathit{Lam}' A)\). Therefore, the first projection of \(\text{\texttt{\textit{flat}'_A e}}\) is taken to be
\[
\hat{e} := \text{\texttt{\textit{flat}}}_A(\text{\texttt{\textit{lam}}} (\pi_1 e)) : \mathit{Lam} A ,
\]
with the renaming with \((\pi_1)_A : \mathit{Lam}' A \rightarrow \mathit{Lam} A\) inside. Thanks to the preservation of hereditary canonicity by \texttt{lam}, the argument \(\text{\texttt{\textit{lam}}} (\pi_1 e)\) to \texttt{\textit{flat}} \(A\) in \(\hat{e}\) is hereditarily canonical. Thanks to the identification of the variables of renamed terms, any free variable of that argument can be identified (up to \(\equiv\)) as \(\pi_1 t\) for a \(t \in \mathit{FV}'_{\mathit{Lam}', A}(\pi_1 e)\), hence is in turn hereditarily canonical. In conclusion, we get \(\text{\texttt{can} } \hat{e}\), which allows to finish the definition of \(\text{\texttt{\textit{flat}'_A e}}\) by pairing the term \(\hat{e}\) with the proof of \(\text{\texttt{can} } \hat{e}\). By construction, \(\pi_1(\text{\texttt{\textit{flat}'_A e}}) \equiv \text{\texttt{\textit{flat}}} (\text{\texttt{\textit{lam} } } \pi_1 \pi_1 e)\).

Since \(\text{\texttt{\textit{flat}'}}\) is doing something with its argument (even if this is only renaming in order to get rid of canonicity information), we cannot think of \(\mathit{Lam}'\) 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 definitional equality \(\equiv\).

From preservation of hereditary canonicity by \texttt{lam} and \texttt{subst}, one can easily define \(\text{\texttt{\textit{lam}'}} : \forall \mathit{mon} \mathit{Lam}'\) and \(\text{\texttt{subst'}} : \forall A \forall B. (A \rightarrow \mathit{Lam}' B) \rightarrow \mathit{Lam}' A \rightarrow \mathit{Lam}' B\), such that \(\pi_1(\text{\texttt{\textit{lam}' } f \ } t) \equiv \text{\texttt{lam} } f (\pi_1 t)\) and \(\pi_1(\text{\texttt{subst'} } f \ } t) \equiv \text{\texttt{subst} } (\pi_1 \circ f) (\pi_1 t)\).

The list of free variables is obtained through \(\mathit{FV}' : \mathit{Lam}' \subseteq \mathit{List}\), defined by pre-composing \(\mathit{FV}\) with \(\pi_1\), and \(\mathit{FV}'\) is then natural with respect to \(\mathit{lam}\) and \(\mathit{map}\) (which immediately follows from naturality of \(\mathit{FV}\)). One readily verifies the analogues to the recursive description (w.r.t. \(\equiv\)) of \(\mathit{FV}\):

\[
\begin{align*}
\text{\texttt{\textit{FV}' (var' } a) & \equiv [a] , \\
\text{\texttt{\textit{FV}' (app} t_1 t_2) & = \text{\texttt{FV}}' t_1 + \text{\texttt{FV}}' t_2 , \\
\text{\texttt{\textit{FV}' (abs} r) & = \text{\texttt{filterSome}}_A (\text{\texttt{FV}}'_A r) , \\
\text{\texttt{\textit{FV}' (\text{\texttt{flat}}_A e}) & = \text{\texttt{\textit{flat}}} \text{\texttt{\textit{map}}}_A (\text{\texttt{FV}}'_A e) .
\end{align*}
\]

It is even trivial to lift item 6 of Theorem 1 to \(\mathit{Lam}'\), yielding
\[
\text{\texttt{\textit{FV}}'(\text{\texttt{subst'} } f \ } t) = \text{\texttt{\textit{flat}}} \text{\texttt{\textit{map}}}_A (\text{\texttt{FV}}' \circ f) (\text{\texttt{FV}}' t) .
\]

One can reprove from naturality of \(\mathit{FV}'\) and the preceding equation or simply transfer (using the two equations two paragraphs above) the identification of free variables of \(\mathit{lam} f \ } t\) and \(\mathit{subst'} f \ } t\) to \(\mathit{lam}'\) and \(\mathit{subst}'\), yielding
\[
\begin{align*}
\forall A \forall B \forall f^A \forall t^B \forall \mathit{lam}' A \forall \mathit{lam}' B, \ b \in \text{\texttt{FV}}'(\mathit{lam}' f t) & \rightarrow \exists a^A. a \in \text{\texttt{FV}}' t \land b = fa , \\
\forall A \forall B \forall f^A \forall \mathit{lam}' \mathit{lam}' B, \ b \in \text{\texttt{FV}}'(\mathit{subst'} f t) & \rightarrow \exists a^A. a \in \text{\texttt{FV}}' t \land b \in \text{\texttt{FV}}'(fa) .
\end{align*}
\]

These first results about \(\mathit{Lam}'\) are rather trivial since they do not concern equality of term representations, but just of their variable names. In order to have “real” results, one has to establish equations between elements of type \(\mathit{Lam}' A\) for some type \(A\). Propositional
equality (=) will not genuinely be adequate since the proofs of hereditary canonicity will hardly agree in non-trivial equations one may wish to prove. One may now axiomatically identify all proofs of hereditary canonicity of a given term \( t : \text{Lam} A \), which amounts to proof irrelevance, or work with a “user-defined” equivalence \( \equiv_A \) on \( \text{Lam} A \), which we will do: define for \( t, u : \text{Lam} A \)

\[
t \equiv_A u : \iff \pi_1 \equiv \pi_1 u .
\]

Evidently, this is a (uniform) family of equivalence relations, for every type \( A \), and we omit the index when there is no risk of confusion.

Assuming proof-irrelevance, recalled in the formulation of Theorem 3, we would be sure that \( t \equiv u \) if and only if \( t = u \). The “if” direction is trivial for reflexive \( \equiv \), but the other allows to replace \( t \) by \( u \) in every context and not only those that are proven to be compatible with the equivalence. If the reader is ready to accept the principle of proof-irrelevance throughout the development (and not just in some confined places for the justification of some induction principle, as is done for the proof of Theorem 3), she or he may always read \( = \) where \( \equiv \) is used in the sequel. We remark that Coq supports working with such equivalences and, more generally, setoids very well in the context of nested datatypes since the most recent version 8.2 of Coq, thanks to Matthieu Sozeau.

Therefore, also from a practical perspective, it did not seem necessary to impose proof-irrelevance in the most recent version 8.2 of Coq, thanks to Matthieu Sozeau. Working with such equivalences and, more generally, setoids very well in the context of nested datatypes since the most recent version 8.2 of Coq, thanks to Matthieu Sozeau.

Finally, a monad structure has been obtained.

For completeness, we also mention the recursive description of \( \text{lam}' \) and \( \text{subst}' \) parallel to that of \( \text{lam} \) and \( \text{subst} \), and some more properties of \( \text{lam}' \) that come from \( \text{lam} \).

The following recursive description is the strict analogue of that for \( \text{lam} \), but with \( \equiv \) in place of \( \simeq \).

\[
\begin{align*}
\text{lam}' f (\text{var}'_A a) & \equiv \text{var}'_B (fa) , \\
\text{lam}' f (\text{app}'_A t_1 t_2) & \equiv \text{app}'_B (\text{lam}' f t_1) (\text{lam}' f t_2) , \\
\text{lam}' f (\text{abs}'_A r) & \equiv \text{abs}'_B (\text{lam}' (\text{opt_map} f) r) , \\
\text{lam}' f (\text{flat}'_A e) & \equiv \text{flat}'_B (\text{lam}' (\text{lam}' f) e) .
\end{align*}
\]
The proof of the last equation needs extensionality and the second functor law for \( \text{lam'} \) and satisfies the two functor laws, but everything w. r. t. \( \equiv \):

- \((\forall a. a \in \text{FV} t \to f a = g a) \to \text{lam'} f t \equiv \text{lam'} g t\)
- \(\text{lam'} (\lambda a. a) t \equiv t\) and \(\text{lam'} (g \circ f) t \equiv \text{lam'} (\text{lam'} f t)\)

We close this collection of results with the strict analogue of the recursive description for \( \text{subst} \), for which we have to provide a dedicated version of lifting, namely

\[
\text{lift'} : \forall A \forall B. (A \to \text{Lam'} B) \to \text{opt } A \to \text{Lam'} (\text{opt } B)
\]

that we define such that

\[
\begin{align*}
\text{lift'}_{A,B} f \text{ None} & \equiv \text{var'}_{\text{opt } B} \text{ None} , \\
\text{lift'}_{A,B} f (\text{Some } a) & \equiv \text{lam'} \text{ Some } (f a) .
\end{align*}
\]

Observe that, generally, \(\pi_1 (\text{lift'} f a) = \text{lift } (\pi_1 \circ f) a\). Unsurprisingly, we arrive at

\[
\begin{align*}
\text{subst'} f (\text{var'} a) & \equiv f a , \\
\text{subst'} f (\text{app'} t_1 t_2) & \equiv \text{app'} (\text{subst'} f t_1) (\text{subst'} f t_2) , \\
\text{subst'} f (\text{abs'} r) & \equiv \text{abs'} (\text{subst'} (\text{lift'} f r)) , \\
\text{subst'} f (\text{flat'} e) & \equiv \text{flat'} (\text{subst'} (\text{var'} e)) e ,
\end{align*}
\]

where the last equation is mostly obsolete in view of property 7 of Theorem 9 (and would even follow from it using also property 6 from that Theorem).

Once again, all the proofs are to be found in the Coq scripts [18].

5.3. Canonization and Exhaustivity for the Constructors of Lam'

Since the datatype constructors \(\text{var'}, \text{app'}, \text{abs'}, \text{flat'}\) of \(\text{Lam'}\) have been defined after the definition of \(\text{Lam'}\) and not \(\text{Lam'}\) defined through them, as it would be usual for an inductive family, a natural question is whether every term of type \(\text{Lam'} A\) is “of one of the forms” \(\text{var'} a\), \(\text{app'} t_1 t_2\), \(\text{abs'} r\) or \(\text{flat'} e\). Being of one of the forms is not meant definitionally, i.e., not with convertibility \(\equiv\), since this would require a syntactic analysis of closed terms. However, we need not be satisfied with \(\equiv\), except for the \(\text{flat'}\) case. Recall that, under proof-irrelevance, the difference between \(=\) and \(\equiv\) becomes immaterial.

**Theorem 10.** The constructors of \(\text{Lam'}\) are exhaustive in the following sense:

\[
\forall A \forall t : \text{Lam'} A. (\exists a. t = \text{var'} a) \lor (\exists t_1, t_2. t = \text{app'} t_1 t_2) \lor (\exists r. t = \text{abs'} r) \lor (\exists e. t \equiv \text{flat'} e) .
\]

We will give a proof where we keep in mind what would be needed to finish it, and only then introduce the ideas how to do that. The first step is to decompose \(t : \text{Lam'} A\) into \(t' := \pi_1 t\) and the proof \(p\) of \(\text{can'} t\). The overall structure of the proof of the theorem is an induction on \(p\). Naturally, we will not be able to profit from the induction hypothesis for exhaustivity. The cases for variables, application and abstraction are extremely simple since the definition of those constructors just paired the terms with the corresponding closure properties of \(\text{can}\). This is why we can even have \(=\) in these cases.
Case flat': We are given $e_0 : \text{Lam}(\text{Lam} A)$ and $p : \text{can} e_0$ and know that all free variable names of $e_0$ are hereditarily canonical. We have to find a term $e : \text{Lam}'(\text{Lam}' A)$ such that flat $e_0$, paired with the can proof from the flat-clause and the given data, is $\equiv$-equal to flat $e$, i.e., flat $e_0 = \pi_1(\text{flat'} e)$. We need to come up with a term $e_1 : \text{Lam}(\text{Lam}' A)$ in can so that $e_1$, paired with the canonicity proof, can be proposed as the desired term $e$. The idea is to obtain $e_1$ through renaming with a function $\text{CAN} : \text{Lam} \subseteq \text{Lam}'$, i.e., $e_1 := \lambda \text{CAN} e_0$. Since lam preserves hereditary canonicity, this is admissible. We must show $\text{flat} e_0 = \pi_1(\text{flat'} e)$, but the right-hand side is $\text{flat} (\text{lam} \pi_1 (\pi_1 e))$ by construction of flat'. With $\pi_1 e \simeq e_1 \simeq \lambda \text{CAN} e_0$, it suffices to show $e_0 = \lambda \pi_1 (\text{lam} \text{CAN} e_0)$. On the other hand, the left-hand side is equal to $\text{lam} \pi_1 (\pi_1 e_0)$, and by the second functor law for lam, the right-hand side is equal to $\text{lam} (\pi_1 \circ \text{CAN}) e_0$. We can use the refined extensionality for lam since $e_0$ is hereditarily canonical. It remains to show for every $t \in \text{FV} e_0$, that $t = \pi_1(\text{CAN} t)$. We know that those $t$ are in can, thus, in order to complete the proof, it suffices to construct a function $\text{CAN} : \text{Lam} \subseteq \text{Lam}'$ such that $\pi_1(\text{CAN} t) = t$ for any hereditarily canonical $t$.

A canonization function can be defined for Lam since its underlying datatype “functor” LamF is monotone in the following sense (compare with Section 4.2 in [16], where bushes were treated similarly): there is an operation $\mathcal{M}$ of type

$$
\forall X \forall Y. \text{mon} Y \rightarrow X \subseteq Y \rightarrow \text{LamF} X \subseteq \text{LamF} Y .
$$

It can be defined as follows by pattern-matching:

$$
\mathcal{M} := \lambda X \lambda Y \lambda m. m \text{mon} Y \lambda f X \subseteq Y \lambda a X \lambda t X A. \text{match} t \text{ with } \begin{cases} \text{var}^- a A & \mapsto \text{var}^- a \\
\text{app}^- t X A t X A & \mapsto \text{app}^- (f A t_1) (f A t_2) \\
\text{abs}^- f X (\text{opt} A) & \mapsto \text{abs}^- (f \text{opt} A r) \\
\text{flat}^- e X (X A) & \mapsto \text{flat}^- (m f A (f X A e)) . \end{cases}
$$

The function that maps lambda terms into hereditarily canonical elements is then generically defined by plain Mendler-style iteration:

$$
\text{Ltc} := \text{MIt} \text{Lam} \left( \lambda X \lambda it X \subseteq \text{Lam} \lambda a X \lambda t X A . \text{InCan}(\mathcal{M} X (\text{Lam} \text{it} t)) \right) .
$$

Notice that InCan is always used to generate the result, which is the first desideratum of canonization. Only the behaviour on canonical elements will be described recursively here:

$$
\begin{align*}
\text{Ltc} (\text{var} a) & \simeq \text{var} a , \\
\text{Ltc} (\text{app} t_1 t_2) & \simeq \text{app} (\text{Ltc} t_1) (\text{Ltc} t_2) , \\
\text{Ltc} (\text{abs} r) & \simeq \text{abs} (\text{Ltc} r) , \\
\text{Ltc} (\text{flat} A e) & \simeq \text{flat} A (\text{lam} \text{Ltc} A (\text{Ltc} A e)) .
\end{align*}
$$

From these rules, induction on can can show that

$$
\forall A \forall t \forall \text{Lam} A . \text{can} t \rightarrow \text{Ltc} t = t .
$$

However, the first functor law for lam and also its refined extensionality are needed for the proof.

For the following important observation, we only offer a proof in the Coq scripts (30 lines of proof):

$$
\forall A \forall t \forall \text{Lam} A . \text{can}(\text{Ltc} t) .
$$
The proof goes by the induction principle $\mu F_{\text{Ind}}$.

The ultimate canonization function $CAN : Lam \subseteq Lam'$ we wanted to have for our proof above, pairs $Ltct$ with the just obtained proof of its hereditary canonicity, for each argument $t$. Then, $\pi_1(CAN \ t) \simeq Ltct$, hence $\pi_1(CAN \ t) = t$ for any hereditarily canonical $t$.

This finishes the proof of Theorem 10. \hfill \Box

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 $LNMIgt$ 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., [17] 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’s style [27], but does not yet have termination guarantees in the context of a useful induction principle like $\mu F_{\text{Ind}}$, that was used intensively in the present article.

A natural question is if nested families of co-inductive types could be treated in a similar fashion. Coq allows to define them if no true nesting is needed, but reasoning is still quite cumbersome, due to a restrictive syntactic guardedness criterion. Programming with truly nested co-inductive datatypes in polymorphic lambda-calculus is possible as well along the lines of [12], but the corresponding logical aspects have not yet been developed. Plain coiteration, however, seems to be of limited applicability, even if the form of the co-recursive calls is not restricted by syntactic conditions, which is the advantage of Mendler’s style.

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 [31] 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 type equation for the nested datatype (based on ideas by Anton
Setzer and Peter Aczel).

In private communication, Andreas Abel has shown to me how to use sized types
not only for the termination guarantee for example programs on truly nested datatypes,
but also for reasoning along the same lines. This is part of the development of Agda2
from Chalmers University. In contrast to my above proposal to add natural numbers
as indices, e.g., inside a Coq development, which unfortunately needs a lot of index
calculations when combining structures for true nesting, the work by Andreas Abel [31]
has a rich subtyping system that allows to pass from annotated versions of the datatypes,
to be seen as approximations, to the unrestricted datatype, thus avoiding most of the
index handling I alluded to before. Moreover, there is no need for non-canonical elements
in that approach. The theoretical status of this verification system is not yet clear to
me. While the cited thesis develops at great depth the foundations in the framework of
higher-order parametric polymorphism, the integration with rich notions of dependent
types with universe hierarchy, and hopefully, simultaneous inductive-recursive definitions,
does not yet seem to be mastered at this level. It should finally be mentioned that “sized
types” are a form of Mendler’s style, but that size information can even be kept after
the definition, while the recursive schemes for Mendler’s style do not add anything to
the typing system, which is leaner but less flexible. The non-intrusiveness into the typing
system made it possible to embed all of the developments of this article directly into the
ambient type theory of Coq, which has been the system of the author’s choice.

References

52–67.
cock (Eds.), Millenial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft
Symp. in Honour of Professor Sir Anthony Hoare, Palgrave, 2000.
[4] F. Bellegarde, J. Hook, Substitution: A formal methods case study using monads and transforma-
[6] T. Altenkirch, B. Reus, Monadic presentations of lambda terms using generalized inductive types,
in: J. Flum, M. Rodríguez-Artalejo (Eds.), Computer Science Logic, 13th International Workshop,
453–468.
[7] P. Johann, N. Ghani, A principled approach to programming with nested types in Haskell, Higher-
[10] C. Paulin-Mohring, Définitions inductives en théorie des types d’ordre supérieur, Habilitation à
diriger les recherches, Université Claude Bernard Lyon I (1996).


[30] V. Capretta, 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).