Map Fusion for Nested Datatypes in Intensional Type Theory

Ralph Matthes

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

Abstract

A definitional extension *LNGMIt* 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". *LNGMIt* 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.

LNGMIt 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 κ_0 stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. Then, let κ_1 be the kind of type transformations, i. e., $\kappa_1 := \kappa_0 \to \kappa_0$. A typical example for a type transformation is *List* of kind κ_1, where *List* A is the type of finite lists with elements from type A. Therefore, *List* itself is a family of datatypes that is indexed over all types. But *List* is not a *nested* datatype since the recursive type equation for *List*, i. e., *List* $A = 1 + A \times$ *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 *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 *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 $\underline{Lam}\,A$, with recursive type equation

$$\underline{Lam}\,A = A + \underline{Lam}\,A \times \underline{Lam}\,A + \underline{Lam}(opt\,A).$$

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 $opt\,A$ that has exactly one more element than A, namely *None*, while the injection of A into $opt\,A$ is called *Some*. The idea is that an element of $\underline{Lam}(opt\,A)$ is seen as an element of $\underline{Lam}\,A$ through lambda abstraction of the extra variable with the designated name *None* in $opt\,A$. 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 κ_0, including types of the form $\underline{Lam}\,B$.

Programming with nested datatypes is possible in the functional programming language 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 $\underline{Lam}$ 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^ω), although the default type-theoretic system of Coq since version 8.0 is "pCIC", the Predicative Calculus of (Co)Inductive Constructions. System F^ω 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 *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

$$Lam\, A = A + Lam\, A \times Lam\, A + Lam(opt\, A) + Lam(Lam\, A) \ .$$

The last summand qualifies *Lam* as truly nested datatype: *Lam A* is the type argument to *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 $Lam \circ Lam$ 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 *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. $=$) terms.

The present article is concerned with an extension of *LNMIt* to a system *LNGMIt* that has generalized Mendler-style iteration *GMIt*, introduced in [12], in addition to plain Mendler-style iteration that is provided by *LNMIt*. Ordinary Mendler-style iteration does not allow a direct definition of the substitution operation on *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 *GMIt* in [12], and the relation to the gfolds of [13] is discussed there. Perhaps surprisingly, *GMIt* could be explained within F^ω through *MIt*. In a sense, this all boils down to the use of a syntactic form of right Kan extensions as the target constructor G^{κ_1} of the polymorphic iterative functions of type $\forall A^{\kappa_0}.\, \mu F A \to G A$, where μF denotes the nested datatype [12, 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 *LNMIt* within CIC with impredicative universe $Set =: \kappa_0$ and propositional proof irrelevance is carried over to *LNGMIt* (Theorem 3). Impredicativity of κ_0 is needed since syntactic Kan extensions use impredicative means for κ_0 in order to stay within κ_1. However, *LNMIt* and *LNGMIt* are *formulated* as extensions of pCIC with its predicative *Set* as κ_0.

The functions that are defined by a direct application of *GMIt* are uniquely determined (up to pointwise propositional equality) by their recursive equation (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

$$\forall A^{\kappa_0} \forall B^{\kappa_0}. (A \to HB) \to XA \to GB$$

for type transformations X, H, G. For the usual polymorphic function spaces described by $\forall A^{\kappa_0}. XA \to GA$, 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 κ_0 which stands for the universe of computationally relevant types (in Coq, this is *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 κ_0 are denoted by A, B and C. Type transformations in κ_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 κ_2. With this naming convention, we will never have to give kinding annotations in the sequel. The universe κ_0 has the following constants and operations: the singleton type 1, the unary operation *opt* for the option type (with term constants *None* and *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 *inl* and the right injection by *inr*, the pair of t_1 and t_2 in a product by (t_1, t_2) and the projections π_1 and π_2. Moreover, there is universal quantification over variables of kind κ_0. However, unless one assumes that κ_0 is impredicative, this leads out of κ_0. In pCIC, $\forall A.B$ has kind *Type*, which is a predicative universe (in fact, a cumulative hierarchy of universes). *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 κ_0 or κ_1 does not lead out of *Type*.

We assume the type transformation *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 ℓ). The type transformation *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 κ_0). We also have $\lambda X.G$ as a rank-2 type transformation (with X a variable in κ_1 and F an expression in κ_1). Application is written as juxtaposition, and XA is a type in κ_0, and FX is a type transformation in κ_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 *Type*.

On the term level, we assume λ-abstraction over typed term variables and over variables of kind κ_0 and κ_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 α-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 *Prop*. The universe *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 κ_0 or variables A, B, C of kind κ_0. On propositions, $\rightarrow$ means implication.

2.2. Plain Mendler-style Iteration MIt

In order to fit the informal definition of *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 κ_2 qualifies as rank-2 functor for the moment, and $\mu F : \kappa_1$ denotes the generated family of datatypes. For our example, set

$$LamF := \lambda X \lambda A.\, A + X\,A \times X\,A + X(opt\,A) + X(X\,A),$$

which has kind κ_2, and $Lam := \mu\, LamF$. In general, there is just one datatype constructor for μF, namely $in : F(\mu F) \subseteq \mu F$. For *Lam*, more clarity comes from the four derived datatype constructors

$$\begin{array}{lcl} var & : & \forall A.\, A \rightarrow Lam\,A \ , \\ app & : & \forall A.\, Lam\,A \rightarrow Lam\,A \rightarrow Lam\,A \ , \\ abs & : & \forall A.\, Lam(opt\,A) \rightarrow Lam\,A \ , \\ \mathit{flat} & : & \forall A.\, Lam(Lam\,A) \rightarrow Lam\,A \ , \end{array}$$

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

$$abs(app\,(var\,None)\,(var\,(Some\,None))) \ ,$$

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

[3] In Haskell 98, one would define *Lam* through the types of its datatype constructors whose names would be fixed already in the definition of *Lam*. In Coq, one can do this for $\underline{Lam}$, but for *Lam*, this will be rejected as being "non-strictly positive".

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

$$t_2 := var\,(Some\, None) : Lam\,(opt(optA)) \ .$$

Now, t shall be represented as the term

$$flat(abs\, t_3) : Lam\,(opt(optA)) \ ,$$

hence with $t_3 : Lam\,(opt\,(Lam\,(opt(optA))))$, defined as

$$t_3 := app\,\Big(app\,(var\, None)\,(var\,(Some\, t_1))\Big)\,(var\,(Some\, t_2)) \ ,$$

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

$$MIt : \forall G.\,(\forall X.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G$$

and the iteration rule

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

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

$$\forall X.\, X \subseteq G \to FX \subseteq G \ .$$

The term s is called the *step term* of the iteration since it provides the inductive step that extends the function from the type transformation X that is to be viewed as approximation to μF, to a function from FX to G.

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

[4] For a recursive removal of all explicit flattenings, in order to obtain results in the family $\underline{Lam}$, 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):

$$\begin{array}{lcl} FV_A\,(var_A\,a) & \simeq & [a]\ , \\ FV_A\,(app_A\,t_1\,t_2) & \simeq & FV_A\,t_1 +\!\!+ FV_A\,t_2\ , \\ FV_A\,(abs_A\,r) & \simeq & filterSome_A\,(FV_{opt\,A}\,r)\ , \\ FV_A\,(flat_A\,e) & \simeq & flat\_map\,FV_A\,(FV_{Lam\,A}\,e)\ . \end{array}$$

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

$$\begin{array}{lcl} filterSome_A\,[] & \simeq & [] \\ filterSome_A\,(None :: \ell) & \simeq & filterSome_A\,\ell \\ filterSome_A\,(Some\,a :: \ell) & \simeq & a :: filterSome_A\,\ell \end{array}$$

The *abs* clause is interpreted as follows: the extra element *None* of $opt\,A$ is the variable name that is considered bound in $abs_A\,r$, and that therefore all its occurrences have to be removed from the list of free variables.

The set of free variables of $flat_A\,e$ is the union of the sets of free variables of the free variables of e, which are still elements of $Lam\,A$.[5] 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 $FV_{Lam\,A}\,e$. This is achieved by the function

$$flat\_map : \forall A \forall B.\, (A \to List\,B) \to List\,A \to List\,B,$$

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

$$\begin{array}{lcl} flat\_map_{A,B}\,f\,[] & \simeq & [] \\ flat\_map_{A,B}\,f\,(a :: \ell) & \simeq & f\,a +\!\!+ flat\_map_{A,B}\,f\,\ell \end{array}$$

In our example, we calculate

$$\begin{array}{lcl} FV_{opt\,(Lam\,(opt(optA)))}\,t_3 & \simeq & None :: Some\,t_1 :: Some\,t_2 :: [] \\ FV_{Lam\,(opt(optA))}\,(abs\,t_3) & \simeq & t_1 :: t_2 :: [] \\ FV_{opt(optA)}\,(flat(abs\,t_3)) & \simeq & FV_{opt(optA)}\,t_1 +\!\!+ FV_{opt(optA)}\,t_2 \simeq None :: SomeNone :: [] \end{array}$$

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 $MIt\,List\,s_{FV}$ for some closed term

$$s_{FV} : \forall X.\, X \subseteq List \to LamFX \subseteq List\ ,$$

and therefore, we have the termination guarantee (in [12], a definition of *MIt* within F^ω 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 *var*, *app*, *abs* and *flat*:

$$\begin{array}{lcllcl} var^-\, a & := & inl(inl(inl\, a)) & abs^-\, r & := & inl(inr\, r) \\ app^-\, t_1\, t_2 & := & inl(inl(inr(t_1, t_2)) & flat^-\, e & := & inr\, e \end{array}$$

We define $s_{FV} := \lambda X \lambda it^{X \subseteq List} \lambda A \lambda t^{LamF\, X\, A}.\, \mathrm{match}\, t\, \mathrm{with}$

$$\begin{array}{lcl} \mid var^-\, a^A & \mapsto & [a] \\ \mid app^-\, t_1^{XA}\, t_2^{XA} & \mapsto & it_A\, t_1 +\!\!+ it_A\, t_2 \\ \mid abs^-\, r^{X(opt\, A)} & \mapsto & filterSome(it_{opt\, A}\, r) \\ \mid flat^-\, e^{X(XA)} & \mapsto & flat\_map\, it_A\, (it_{XA}\, e)\ . \end{array}$$

For $FV := MIt\ List\ s_{FV}$, the required equational specification is obviously satisfied (since the pattern-matching mechanism behaves properly with respect to definitional equality $\simeq$).[6]

The visible reason why Mendler's style can guarantee termination without any syntactic descent (in which way can the flat-mapping over FV_A be seen as "smaller"?[7]) is the following: the recursive calls come in the form of uses of *it*, which does not have type $LamF \subseteq List$ but just $X \subseteq List$, and the type arguments of the datatype constructors are replaced by variants that only mention X instead of *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 *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 *map term* as

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

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 $X := List$, with function $map : mon\, List$ that maps its function argument over all the elements of its list argument. However, also the *renaming* operation *lam* will have a type of this form, more precisely, $lam : mon\, Lam$, and $lam\, f\, t$ has to represent t after renaming every free variable occurrence of name a in t by the variable of name fa. It would be possible to define *lam* by help of *GMIt* introduced in the next section, but it will automatically be available in the systems *LNMIt* and *LNGMIt* that will be described in Section 4. Therefore, we content ourselves in displaying its recursive behaviour (we omit the type arguments to *lam*):

$$\begin{array}{lcl} lam\, f\, (var_A\, a) & \simeq & var_B(fa)\ , \\ lam\, f\, (app_A\, t_1\, t_2) & \simeq & app_B\, (lam\, f\, t_1)\, (lam\, f\, t_2)\ , \\ lam\, f\, (abs_A\, r) & \simeq & abs_B(lam\, (opt\_map\, f)\, r)\ , \\ lam\, f\, (flat_A\, e) & \simeq & flat_B(lam\, (lam\, f)\, e)\ . \end{array}$$

[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 $opt\_map : mon\, opt$ that lifts any function of type $A \to B$ to one of type $opt\, A \to opt\, B$. Notice that *lam* is called with type arguments $opt\, A$ and $opt\, B$. In the final clause, the outer call to *lam* is with type arguments $Lam\, A$ and $Lam\, B$, while the inner one stays with A and B. Thus, the free variables of $e : Lam(Lam\, A)$, that have names in $Lam\, A$, are renamed according to $lam\, f : Lam\, A \to Lam\, B$, 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

$$\forall A \forall B \forall f^{A \to B} \forall t^{Lam\, A}.\, FV_B(lam\, f\, t) = map\, f\, (FV_A\, t) \ ?$$

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 $m_X : mon\, X$ and $m_G : mon\, G$. 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:

$$\forall A \forall B \forall f^{A \to B} \forall t^{X\, A}.\, h_B\, (m_X\, A\, B\, f\, t) = m_G\, A\, B\, f\, (h_A\, t) \ .$$

The system *LNMIt*, described in Section 4.1, allows to answer the above question by showing that FV is a natural transformation from (Lam, lam) to $(List, map)$, i. e., where $X := Lam$, $m_X := lam$, $G := List$, $m_G := map$ and $h := FV$. 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 *LNMIt* and *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 *MIt*, one can also define a function $eval : Lam \subseteq \underline{Lam}$ that evaluates all the explicit flattenings and thus yields the representation of a usual lambda term [16]. In [16], also *eval* is seen in *LNMIt* to be a natural transformation, w. r. t. renaming for *Lam* and $\underline{Lam}$, respectively.

2.3. Generalized Mendler-style Iteration GMIt

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

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

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

[8] We say "would" since one of the monad laws cannot be established in *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:

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

for any H, G. Here, we λ-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 κ_0 in the proof of Proposition 1. In pCIC, $Ran_H\ G$ has kind $\kappa_0 \to Prop$.

For our example of substitution, we could use $Ran_{Lam}\ Lam$, based on the following evident logical equivalence:

$$\big(\forall A.\, Lam\ A \to (Ran_{Lam}\ Lam)\ A\big) \Leftrightarrow \big(\forall A \forall B.\, (A \to Lam\ B) \to Lam\ A \to Lam\ B\big)\ ,$$

where only the universal quantification over B has to be moved across the implication $Lam\ A \to \cdot$ and the two premisses $Lam\ A$ and $A \to Lam\ B$ 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 κ_0 because, only with impredicative κ_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

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

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

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

and the generalized iteration rule

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

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

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

$$s_{subst} : \forall X.\, X \leq_{Lam} Lam \to LamF\ X \leq_{Lam} Lam\ ,$$

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

$$\begin{array}{l}
\lambda X \lambda it^{X \leq_{Lam} Lam} \lambda A \lambda B \lambda f^{A \to Lam\ B} \lambda t^{LamF\ X\ A}.\, \mathrm{match}\ t\ \mathrm{with} \\
\quad \begin{array}{lcl}
| \, var^-\, a^A & \mapsto & f\, a \\
| \, app^-\, t_1^{X\,A}\, t_2^{X\,A} & \mapsto & app\ (it_{A,B}\ f\ t_1)\ (it_{A,B}\ f\ t_2) \\
| \, abs^-\, r^{X(opt\ A)} & \mapsto & abs(it_{opt\ A, opt\ B}\ (lift\ f)\ r) \\
| \, flat^-\, e^{X(XA)} & \mapsto & flat\big(it_{XA, Lam\ B}(var_{Lam\ B} \circ (it_{A,B}\ f))\ e\big)\ .
\end{array}
\end{array}$$

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

$$\mathit{lift} : \forall A \forall B.\, (A \to \mathit{Lam}\, B) \to \mathit{opt}\, A \to \mathit{Lam}(\mathit{opt}\, B) \ ,$$

definable by pattern-matching, with properties

$$\begin{array}{lcl} \mathit{lift}_{A,B}\, f\, \mathit{None} & \simeq & \mathit{var}_{\mathit{opt}\, B}\, \mathit{None} \ , \\ \mathit{lift}_{A,B}\, f\, (\mathit{Some}\, a) & \simeq & \mathit{lam}\, \mathit{Some}\, (f a) \ , \end{array}$$

where renaming *lam* is essential.

Note that $\mathit{var}_{\mathit{Lam}\, B} \circ (\mathit{it}_{A,B}\, f)$ has type $XA \to \mathit{Lam}(\mathit{Lam}\, B)$ (the infix operator $\circ$ denotes composition of functions). From the point of view of clarity of the definition, we would have much preferred $\mathit{flat}\, (\mathit{lam}\, (\mathit{it}_{A,B}\, f)\, e)$ to the term in the last clause of the definition of s_{subst}. It would express that substitution is only carried out on the terms-as-variables in the argument $e : \mathit{Lam}(\mathit{Lam}\, A)$ of $\mathit{flat}\, e$, without touching the outer term structure. But that right-hand side would only type-check *after* instantiating X with *Lam*, hence generalized Mendler-style iteration cannot accept this alternative. However, one could rectify this by applying an extra hypothetical transformation $j : X \subseteq \mathit{Lam}$ to e, i. e., by taking $\mathit{flat}\, (\mathit{lam}\, (\mathit{it}_{A,B}\, f)\, (j_{X\, A}\, e))$ as right-hand side in the last case of the pattern-matching definition of s_{subst}, assuming j would be instantiated by the polymorphic identity on *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 *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 *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 *subst* above that would be executable in pure higher-order polymorphic λ-calculus, as shown in [12].

Our definition satisfies

$$\mathit{subst}\, f\, (\mathit{flat}\, e) \simeq \mathit{flat}(\mathit{subst}(\mathit{var} \circ (\mathit{subst}\, f))\, e) \ ,$$

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 $\mathit{Lam}\, A$ are not only renamed into the resulting terms of type $\mathit{Lam}\, B$, but they are substituted by the terms of type $\mathit{Lam}(\mathit{Lam}\, B)$ that happen to be variables with those resulting terms as names.

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

$$\begin{array}{lcl} \mathit{subst}\, f\, (\mathit{app}\, t_1\, t_2) & \simeq & \mathit{app}\, (\mathit{subst}\, f\, t_1)\, (\mathit{subst}\, f\, t_2) \ , \\ \mathit{subst}\, f\, (\mathit{abs}\, r) & \simeq & \mathit{abs}\, (\mathit{subst}\, (\mathit{lift}\, f)\, r) \ . \end{array}$$

The results beyond convertibility about *subst* are collected in the following theorem.

Theorem 1. *In system LNGMIt, to be defined later in this article, one can prove the following about the representation subst of substitution for lambda terms with explicit flattening, where we mean the universal (and well-typed) closure of all statements:*

1. $(\forall a.\, f\,a = g\,a) \to \mathit{subst}\, f\, t = \mathit{subst}\, g\, t$
2. $(\forall a.\, a \in \mathit{FV}\, t \to f\,a = g\,a) \to \mathit{subst}\, f\, t = \mathit{subst}\, g\, t$
3. $\mathit{lam}\, g\, (\mathit{subst}\, f\, t) = \mathit{subst}\, (\,(\mathit{lam}\, g) \circ f)\, t$
4. $\mathit{subst}\, g\, (\mathit{lam}\, f\, t) = \mathit{subst}\, (g \circ f)\, t$
5. $\mathit{subst}\, g\, (\mathit{subst}\, f\, t) = \mathit{subst}\, ((\mathit{subst}\, g) \circ f)\, t$
6. $\mathit{FV}(\mathit{subst}\, f\, t) = \mathit{flat\_map}\, (\mathit{FV} \circ f)\, (\mathit{FV}\, t)$

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 $a \in \ell$ 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., $\mathit{subst}\ \mathit{var}_A\, t^{\mathit{Lam}\, A} = t$ is not provable, see Section 5) to *List*. An easy consequence from it is $b \in \mathit{FV}(\mathit{subst}\, f\, t) \to \exists a.\, a \in \mathit{FV}\, t \wedge b \in \mathit{FV}(f\,a)$, 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 $\simeq$ is directly available, without any need for equational reasoning. For example, the term $\mathit{GMIt}\, H\, G\, s\, A\, B\, f\, (\mathit{in}\, A\, t)$ is definitionally equal to $s\, \mathit{Lam}\, (\mathit{GMIt}\, H\, G\, s)\, A\, B\, f\, t$ 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 $\simeq$ 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_{κ_0}. 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 $(\mathit{mon}\, X) \simeq (X \leq X)$.

Assume a function $h : X \subseteq G$ and map terms $m_X : \mathit{mon}\, X$ and $m_G : \mathit{mon}\, G$. 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 $h\, f$ 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

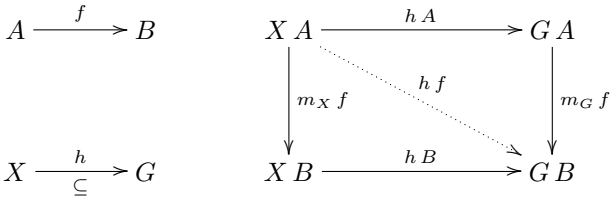

Figure 1: Naturality of $h : X \subseteq G$

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

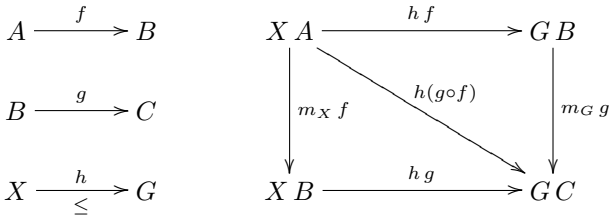

Figure 2: Naturality of $h : X \leq G$

functional h for which the second diagram commutes, one obtains in a unique way a natural transformation h from X to G with h_A being $h\, id_A$. In category theory, this is a simple exercise, but in our intensional setting, this allows to define naturality for any $X, G : \kappa_1$, $m_X : mon\, X$, $m_G : mon\, G$ and $h : X \leq G$.

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

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

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

Definition2 (Naturality of $h : X \leq_H G$). Given $X, H, G : \kappa_1$ and $h : X \leq_H G$, define the two parts of naturality of h as follows: If $m_H : mon\, H$ and $m_G : mon\, G$, define the first part $gnat_1\ m_H\, m_G\, h$ by

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

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

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

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

In Theorem 1, the third item, $lam\, g\, (subst\, f\, t) = subst\, (\, (lam\, g) \circ f)\, t$, is nothing but $gnat_1$ *lam lam subst* without the quantifiers, and the fourth item, $subst\, g\, (lam\, f\, t) = subst\, (g \circ f)\, t$, is the unquantified $gnat_2$ *lam subst*.

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

Lemma 2. *Given $X, H, G : \kappa_1$, $m_X : mon\, X$, $m_H : mon\, H$, $m_G : mon\, G$ and $h : X \leq_H G$ such that $gnat_1\ m_H\, m_G\, h$ and $gnat_2\, m_X\, h$ hold, the function*

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

is natural 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 $f : A \to B$ and $x : X(HA)$. We have to show

$$m_G\, f\, (h^{\subseteq}_A\, x) = h^{\subseteq}_B (m_X\, (m_H\, f)\, x) \ .$$

The l. h. s. is $m_G\, f\, (h_{HA,A}\, (\lambda y.y)\, x) = h_{HA,B}\, ((m_H\, f) \circ (\lambda y.y))\, x$ by $gnat_1$. The r. h. s. is $h_{HB,B}\, (\lambda y.y)\, (m_X\, (m_H\, f)\, x) = h_{HA,B}\, ((\lambda y.y) \circ (m_H\, f))\, x$ by $gnat_2$. 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 $(GMIt\, s)^{\subseteq}$. For example, $(subst)^{\subseteq} : Lam \circ Lam \subseteq Lam$ would be the multiplication operation of the monad $(Lam, var, subst)$ (but, as mentioned before, we will not be able to establish all the monad laws). Unlike *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 *LNMIt* from [16], then we extend it by *GMIt* and its definitional rules in order to obtain its extension *LNGMIt*.

The following definitions should be readable on the basis of the notations of Section 2.1, in particular the definitions of κ_1, κ_2, *Type*, $\subseteq$ and $\simeq$ and the conventions on the kinds of A, G, X. We frequently used the symbol $\circ$ for function composition, and also the definition $mon\, X := \forall A \forall B.\, (A \to B) \to XA \to XB$, which is a type of kind *Type*.

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

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

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

Parameters:

F : κ_2

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

Constants:

μF : κ_1

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

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

MIt : $\forall G.\, (\forall X.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G$

$\mu FInd$: $\forall P : (\forall A.\, \mu F A \to Prop).\, \Big(\forall X \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(m\, ef,\, map_{\mu F})}.$
$\big(\forall A \forall x^{XA}.\, P_A(j_A\, x)\big) \to \forall A \forall t^{FXA}.\, P_A(In\; ef\; j\; n\; t)\Big)$
$\to \forall A \forall r^{\mu F A}.\, P_A\, r$

Rules:

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

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

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

Figure 3: Specification of *LNMIt* as extension of pCIC.

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 $m_X : mon\, X$ and $m_G : mon\, G$ and a term $h : X \subseteq G$, the proposition expressing h's naturality w. r. t. m_X, m_G is

$$h \in \mathcal{N}(m_X,\, m_G) := \forall A \forall B \forall f^{A \to B} \forall t^{XA}.\, h_B\, (m_X\, A\, B\, f\, t) = m_G\, A\, B\, f\, (h_A\, t)\ .$$

4.1. LNMIt

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

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

Given a record ef of type $\mathcal{E}X$, Coq's notation for its field m is $m\, ef$, and likewise for the other fields. We adopt this notation instead of the more common $ef.m$. Preservation of extensional[9] functors by F is required in the form of a term of type $\forall X.\, \mathcal{E}\, X \to \mathcal{E}(FX)$, and *LNMIt* is defined to be pCIC with $\kappa_0 := Set$, extended by the constants and rules of Figure 3, adopted from [16]. It has to be admitted that this specification goes beyond the explained notations from Section 2.1: the type of *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 *Type* and under adding a proposition as premise since *Prop* is included in *Type*. The kind of $\mu FInd$'s type is *Prop* since *Prop* is also closed under universal quantification over variables with a complex type of kind *Type* such as $\forall A.\, \mu F A \to Prop$ ($\mu F A$ also has kind *Type* since *Set* is included in *Type* and *Prop* is of kind *Type*). The application of P of that type to A and $r : \mu F A$ is denoted by $P_A r$, which is a proposition.

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

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

The proof of this theorem needs the induction rule $\mu FInd$ in order to show that $map_{\mu F}$ is extensional and satisfies the functor laws. These proofs enter $ef_{\mu F}$, and *In* can then be instantiated with $X := \mu F$, $ef := ef_{\mu F}$ and j the identity on μF with its trivial proof of naturality, to yield the desired *InCan*.[10]

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

$$\forall A \forall B \forall f^{A \to B} \forall x^{XA}.\, j_B\, (m\, ef\, A\, B\, f\, x) = map_{\mu F}\, A\, B\, f\, (j_A\, x)\ ,$$

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

Notice also that the definitional rule for $map_{\mu F}$ is not even recursive, but only does pattern-matching on the single datatype constructor *In* of μF. So, *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 *MIt* has not been touched at all; there is just a more general iteration rule that also covers non-canonical elements, but for the canonical elements, we get the same behaviour, i. e., the same equation with respect to $\simeq$. The crucial part is the induction principle $\mu FInd$. Without access to the argument n that assumes naturality of j as a transformation from $(X, m\, ef)$ to $(\mu F, map_{\mu F})$, one would not be able to prove naturality of $MIt\, s$, i. e., of iteratively defined functions on the nested datatype μF. The author is not aware of ways how to avoid non-canonical elements and nevertheless have

[10] By taking the identity for j in the second definitional rule of *LNMIt*, one obtains after β-reduction an η-expansion of $(MIt\, s)_A$, and this has to be remedied by the somewhat undesirable third definitional rule of *LNMIt*. Since it is an η-like rule, one would rather not like to require it. In the Coq development, this is avoided by defining *MIt* by appropriate η-expansions of a preliminary constant of the same type.

Constant:

$GMIt \quad : \quad \forall H \forall G.\, (\forall X.\, X \leq_H G \to FX \leq_H G) \to \mu F \leq_H G$

Rules:

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

$\lambda A \lambda B \lambda f^{A \to HB} \lambda x^{\mu F A}.\, GMIt_{H,G}\, s\, A\, B\, f\, x \simeq GMIt_{H,G}\, s$

Figure 4: Specification of *LNGMIt* as extension of *LNMIt*.

an induction principle that allows to establish naturality of $MIt\, s$ [16, Theorem 1] (but see the final discussion in the conclusions).

The system *LNMIt* can be defined within CIC with impredicative *Set*, extended by the principle of proof irrelevance, i. e., by $\forall P : Prop\, \forall p_1^P \forall p_2^P.\, p_1 = p_2$. This is the main result of [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 *LNMIt* implies convertibility in that implementation.

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

$$\begin{array}{l} \lambda X \lambda m^{mon\, X} \lambda A \lambda B \lambda f^{A \to B} \lambda t^{LamF\, X\, A}.\, \mathrm{match}\, t\, \mathrm{with} \\ \quad \begin{array}{lcl} \mid var^-\, a^A & \mapsto & var^-(fa) \\ \mid app^-\, t_1^{XA}\, t_2^{XA} & \mapsto & app^-\, (m_{A,B}\, f\, t_1)\, (m_{A,B}\, f\, t_2) \\ \mid abs^-\, r^{X(opt\, A)} & \mapsto & abs^-\, (m_{opt\, A, opt\, B}\, (opt\_map\, f)\, r) \\ \mid flat^-\, e^{X(XA)} & \mapsto & flat^-\, ((m \star m)_{A,B}\, f\, e)\ , \end{array} \end{array}$$

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

4.2. LNGMIt

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

Theorem [16, Theorem 3] about $ef_{\mu F}$ and *InCan* for *LNMIt* immediately extends to *LNGMIt* and yields the following additional convertibility:

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

which has this concise form only because of the η-like rule for *GMIt* that was made part of *LNGMIt* (the second rule in Figure 4). Thus, we get back the original behaviour of *GMIt* described in Section 2.3, but with the derived datatype constructor *InCan* instead

of the *defining* datatype constructor *in*. For our case study, we are guaranteed that *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 κ_0 of computationally relevant types is impredicative.*

Proof. The proof is nothing but the observation that the embedding of GMIt^ω into MIt^ω of [12, Section 4.3] extends for our situation of a rank-2 inductive constructor μF to non-canonical elements, i. e., the full datatype constructor *In* instead of only *in*, considered in that work: define for $H, G : \kappa_1$ the terms

$$\begin{array}{lcl} \mathit{LeqRan} & := & \lambda X \lambda h^{X \leq_H G} \lambda A \lambda x^{XA} \lambda B \lambda f^{A \to HB}.\, h\, A\, B\, f\, x \ , \\ \mathit{RanLeq} & := & \lambda X \lambda h^{X \subseteq \mathit{Ran}_H\, G} \lambda A \lambda B \lambda f^{A \to HB} \lambda x^{XA}.\, h\, A\, x\, B\, f \ . \end{array}$$

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

$$\begin{array}{lcl} \mathit{LeqRan} & : & \forall X.\, X \leq_H G \to X \subseteq \mathit{Ran}_H\, G \ , \\ \mathit{RanLeq} & : & \forall X.\, X \subseteq \mathit{Ran}_H\, G \to X \leq_H G \ . \end{array}$$

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

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

Then, we can define

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

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*:

$$\begin{array}{lcl} \mathit{GMIt}_{H,G}\, s\, f\, (\mathit{In}\, \mathit{ef}\, j\, n\, t) & \simeq & \mathit{RanLeq}_{\mu F}\, (\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')\, f\, (\mathit{In}\, \mathit{ef}\, j\, n\, t) \\ & \simeq & \mathit{MIt}_{\mathit{Ran}_H\, G}\, s'\, (\mathit{In}\, \mathit{ef}\, j\, n\, t)\, f \\ & \simeq & s' \big(\lambda A.\, (\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')_A \circ j_A\big)\, t\, f \\ & \simeq & \mathit{LeqRan}_{FX}\, (s_X\, (\mathit{RanLeq}_X\, (\lambda A.\, (\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')_A \circ j_A)))\, t\, f \\ & \simeq & s_X\, (\mathit{RanLeq}_X\, (\lambda A.\, (\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')_A \circ j_A))\, f\, t \ , \end{array}$$

and the first term argument to s is then convertible with

$$\lambda A \lambda B \lambda f^{A \to HB} \lambda x^{XA}.\, \mathit{MIt}_{\mathit{Ran}_H\, G}\, s'\, A\, (j_A\, x)\, B\, f$$

and further with $\lambda A \lambda B \lambda f^{A \to HB}.\, (\mathit{RanLeq}_{\mu F}\, (\mathit{MIt}_{\mathit{Ran}_H\, G}\, s')\, A\, B\, f) \circ j_A$.

The η-like rule for *GMIt* is immediate from the definition since $\mathit{RanLeq}_{\mu F}\, h$ β-reduces for any term h to a term that has the following prefix $\lambda A \lambda B \lambda f \lambda x$. Therefore, η-expansion with those four variables can be eliminated just by β-reduction.[11] Impredicativity of κ_0 is needed to have $\mathit{Ran}_H\, G : \kappa_1$, 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 LNGMIt can be defined within CIC with impredicative Set, extended by the principle of propositional proof irrelevance, i. e., by admitting the axiom* $\forall P : Prop\, \forall p_1^P \forall p_2^P . p_1 = p_2$.

Proof. Use the previous proposition and the main theorem of [16] that states the same property of *LNMIt*. □

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

Theorem 4 (Uniqueness of *GMIt s***).** *Assume type transformations* $H, G : \kappa_1$ *and terms* $s : \forall X. X \leq_H G \to FX \leq_H G$ *and* $h : \mu F \leq_H G$ *(the candidate for being GMIt s). Assume further the following extensionality property of s (s only depends on the extension of its first function argument, but in a way adapted to the parameter f):*

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

Assume finally that h satisfies the equation for GMIt s:

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

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

Proof. By the induction principle $\mu FInd$, as for [16, Theorem 2]. It seems nevertheless appropriate to show the argument. The induction principle is used with the property

$$P := \lambda A \lambda r^{\mu F\, A} \forall B \forall f^{A \to HB}. h_{A,B}\, f\, r = (GMIt\, s)_{A,B}\, f\, r \ ,$$

where the quantification of the parameters B and f is compulsory already for typing purposes. Then assume the appropriate X, ef, j, n. The inductive hypothesis is

$$\forall A \forall x^{XA} \forall B \forall f^{A \to HB}. h_{A,B}\, f\, (j_A\, x) = (GMIt\, s)_{A,B}\, f\, (j_A\, x) \ .$$

We assume further $A, B, f^{A \to HB}, t^{FXA}$ and have to show

$$h_{A,B}\, f\, (In\, ef\, j\, n\, t) = (GMIt\, s)_{A,B}\, f\, (In\, ef\, j\, n\, t).$$

Applying the equational hypothesis on h and the computation rule for *GMIt* yields the following equivalent equation:

$$s\, (\lambda A \lambda B \lambda f^{A \to HB}. (h_{A,B}\, f) \circ j_A)\, f\, t = s(\lambda A \lambda B \lambda f^{A \to HB}. ((GMIt\, s)_{A,B}\, f) \circ j_A)\, f\, t \ .$$

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

$$\forall A \forall B \forall f^{A \to HB} \forall x^{XA}. ((h_{A,B}\, f) \circ j_A)\, x = (((GMIt\, s)_{A,B}\, f) \circ j_A)\, x \ ,$$

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

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

(The name *gext* stands for *generalized extensionality.*) The earlier definition of *ext* is the special instance where X and G coincide and where H is the identity type transformation $Id_{\kappa_0} := \lambda A.\, A$.

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

The following statement has no precursor in *LNMIt*.

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

Proof. An easy application of $\mu FInd$. The predicate we need is

$$P := \lambda A \lambda r^{\mu F\, A} \forall B \forall f, g^{A \to HB}.\, (\forall a^A.\, fa = ga) \to (GMIt\, s)_{A,B}\, f\, r = (GMIt\, s)_{A,B}\, g\, r \ .$$

Then assume the appropriate X, ef, j, n and the inductive hypothesis $\forall A \forall x^{XA}.\, P_A(j_A\, x)$. Given A, t^{FXA}, we want to show $P_A(In\, ef\, j\, n\, t)$. So, assume B and $f, g^{A \to HB}$ such that $\forall a^A.\, fa = ga$. Our aim is to show $(GMIt\, s)_{A,B}\, f\, (In\, ef\, j\, n\, t) = (GMIt\, s)_{A,B}\, g\, (In\, ef\, j\, n\, t)$. Both sides are convertible by help of the first rule for *GMIt*. Since s preserves extensionality, it suffices to show generalized extensionality for the common first term argument of s in both reducts, i. e., $gext(\lambda A \lambda B \lambda f^{A \to HB}.\, ((GMIt\, s)_{A,B}\, f) \circ j_A)$. Up to order of quantifiers and convertibility, this is just the induction hypothesis. □

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

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

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

Theorem 6 (First part of naturality of *GMIt s*). *Given $H, G : \kappa_1$, map terms $m_H : mon\, H$, $m_G : mon\, G$ and a term $s : \forall X.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality. Assume further*

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

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

Proof. Induction with $\mu FInd$ for the predicate

$$\lambda A \lambda r^{\mu\, F\, A} \forall B \forall C \forall f^{A \to HB} \forall g^{B \to C}.\, m_G\, g\, ((GMIt\, s)_{A,B}\, f\, r) = (GMIt\, s)_{A,C}\, ((m_H\, g) \circ f)\, r \ .$$

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

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

We want to use Theorem 5 for the function $h : X \leq_H G$, defined by

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

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 $\mathit{gext}(\mathit{GMIt}\, s)$, 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

$$m_G\, g\, (s\, h\, f\, t) = s\, h\, ((m_H\, g) \circ f)\, t \ ,$$

hence we only have to show $\mathit{gnat}_1\, m_H\, m_G\, (s\, h)$. The main assumption of the theorem is made for that: *ef* has type $\mathcal{E}\, X$, $\mathit{gext}\, h$ has been shown above, and the induction hypothesis provides $\mathit{gnat}_1\, m_H\, m_G\, h$. □

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.

Theorem 7 (Second part of naturality of *GMIt s*—map fusion). *Given $H, G : \kappa_1$ and a term $s : \forall X.\, X \leq_H G \to FX \leq_H G$ that preserves extensionality. Assume further*

$$\forall X \forall h^{X \leq_H G} \forall ef^{\mathcal{E}\, X}.\, \mathit{gext}\, h \to \mathit{gnat}_2\, (m\, ef)\, h \to \mathit{gnat}_2\, (m\, (Fp\mathcal{E}\ ef))\, (s\, h) \ .$$

Then, GMIt s satisfies the second part of naturality, i. e., $\mathit{gnat}_2\ \mathit{map}_{\mu F}\, (\mathit{GMIt}\, s)$.

Proof. Induction with $\mu\mathit{FInd}$, quite analogously to the previous proof. The induction predicate is

$$\lambda A \lambda r^{\mu\, F\, A} \forall B \forall C \forall f^{A \to B} \forall g^{B \to HC}.\, (\mathit{GMIt}\, s)_{B,C}\, g\, (\mathit{map}_{\mu F}\, f\, r) = (\mathit{GMIt}\, s)_{A,C}\, (g \circ f)\, r \ .$$

As before, assume the appropriate $X, ef, j, n, A, t, B, C, f, g$. We have to show

$$(\mathit{GMIt}\, s)_{B,C}\, g\, (\mathit{map}_{\mu F}\, f\, (\mathit{In}\ ef\ j\, n\, t)) = (\mathit{GMIt}\, s)_{A,C}\, (g \circ f)\, (\mathit{In}\ ef\ j\, n\, t) \ .$$

The left-hand side enjoys a rule application for $\mathit{map}_{\mu F}$ and is thus convertible with

$$(\mathit{GMIt}\, s)_{B,C}\, g\, (\mathit{In}\ ef\ j\, n\, (m\, (Fp\mathcal{E}\ ef)\, f\, t)) \ .$$

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 $\mathit{gext}\, h$. 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 $\mathit{gnat}_2\, (m\, (Fp\mathcal{E}\ ef))\, (s\, h)$, but here, the main assumption of the theorem applies if we can show $\mathit{gnat}_2\, (m\, ef)\, h$. We now crucially need naturality of j that comes as assumption n of $j \in \mathcal{N}(m\, ef,\ \mathit{map}_{\mu F})$ with the induction principle. Assume A, B, C, f, g, x and calculate

$$\begin{array}{rcl} h_{B,C}\, g(m\ ef\ f\, x) & \simeq & (\mathit{GMIt}\, s)_{B,C}\, g\, (j_B(m\ ef\ f\, x)) \\ & = & (\mathit{GMIt}\, s)_{B,C}\, g\, (\mathit{map}_{\mu F}\, f\, (j_A\, x)) \\ & = & (\mathit{GMIt}\, s)_{A,C}\, (g \circ f)\, (j_A\, x) \\ & \simeq & h_{A,C}\, (g \circ f)\, x \end{array}$$

The first = uses naturality of j, the second applies the induction hypothesis. □

Although the proof is rather simple, this is the main point of the complicated system *LNGMIt* with its inductive-recursive nature: ensure naturality to be available for j inside the inductive step of reasoning on μF. One might wonder whether this theorem could be an instance of [16, Theorem 1], using the definition of *GMIt* in Proposition 1 for impredicative κ_0. This is not true, due to problems with extensionality: Proving propositional equality between functions rarely works in intensional type theory such as CIC, and the use of $Ran_H\ G$ in the construction of 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 $(subst)^{\subseteq} : Lam \circ Lam \subseteq Lam$ is natural with respect to $lam \star lam$ and lam. The fifth item (the interchange law for substitution that is one of the monad laws) can now be proven by the induction principle $\mu FInd$, 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 *lift* is defined by help of *lam*).

5. Completion of the Case Study on Substitution

The last item of Theorem 1 on properties of *subst* can be proven by the induction principle $\mu FInd$ without any results about *Lam*, just with several preparations about lists, also using naturality of 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

$$\forall A \forall t^{Lam\ A}.\ subst\ var_A\ t = t\ ,$$

which has an η-like flavour since $subst\ var_A$ must then have any term representation t as image. We called every term of the form $InCan\ t$ with $t : F(\mu F)A$ a canonical term in $\mu F A$; this makes the terms of the form $app\ t_1\ t_2$ or $abs\ r$ or $flat\ e$ canonical terms that govern the results of substitution in all but the variables case, as can be seen from the definition of s_{subst}. Because of the presence of non-canonical terms in *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 *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 *Bush* of "bushes" with $Bush\ A = 1 + A \times Bush(Bush\ A)$, 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.

Definition3 (Hereditarily canonical term). Define the notion of hereditarily canonical elements of the nested datatype *Lam*, the predicate $can : \forall A.\ Lam\ A \to Prop$, inductively by the following four closure rules:

- $\forall A \forall a^A.\ can\ (var\ a)$
- $\forall A \forall t_1^{Lam\ A} \forall t_2^{Lam\ A}.\ can\ t_1 \to can\ t_2 \to can(app\ t_1\ t_2)$
- $\forall A \forall r^{Lam(opt\ A)}.\ can\ r \to can\ (abs\ r)$

- $\forall A \forall e^{Lam(Lam\,A)}.\, can\, e \to (\forall t^{Lam\,A}.\, t \in FV\, e \to can\, t) \to can\, (\mathit{flat}\, e)$

Hence, *can* is closed under all the term formation rules of *Lam*, except for *flat*, where the extra assumption for hereditary canonicity of $\mathit{flat}\, e$ 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 $t \in FV\, e$. 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 $t \in FV_{Lam\,A}\, e$, thus ensuring well-definedness of $\mathit{flat\_map}\, FV_A\, (FV_{Lam\,A}\, e)$.

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:

$$can_2 : \forall A.\, (A \to Prop) \to Lam\, A \to Prop$$

is inductively defined by the following closure rules:

- $\forall A \forall P^{A \to Prop} \forall a^A.\, Pa \to can_2\, P\, (var\, a)$
- $\forall A \forall P^{A \to Prop} \forall t_1^{Lam\,A} \forall t_2^{Lam\,A}.\, can_2\, P\, t_1 \to can_2\, P\, t_2 \to can_2\, P\, (app\, t_1\, t_2)$
- $\forall A \forall P^{A \to Prop} \forall r^{Lam(opt\,A)}.\, can_2\, (optpred\, P)\, r \to can_2\, P\, (abs\, r)$
- $\forall A \forall P^{A \to Prop} \forall e^{Lam(Lam\,A)}.\, can_2\, (can_2\, P)\, e \to can_2\, P\, (\mathit{flat}\, e)$

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 can_2 itself in the last closure rule. Therefore, can_2 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 can_2 axiomatically: just define constants corresponding to the closure rules. An induction principle is also easy to assume as an extra constant: it expresses that can_2 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 can_2' of the

same type as can_2 can be defined in CIC. It has the same first three closure rules (just with can_2' instead of can_2), but the last rule is as follows:

$$\forall A \forall P^{A \to Prop} \forall e^{Lam(Lam\, A)}.\, can_2'\, U\, e \to (\forall t^{Lam\, A}.\, t \in FV\, e \to can_2'\, P\, t) \to can_2'\, P\, (flat\, e)$$

Here, U denotes the "universal" predicate that is always true. This variant is very close to the original *can*, and it is again not generic. It is easy to see that $can_2'\, P\, t$ implies $can\, t$ and that $can\, t$ implies $can_2'\, U\, t$, again with U the universal predicate.

By the help of some extra lemmas, one can show that $can_2\, P\, t$ and $can_2'\, P\, t$ are logically equivalent propositions for all P, t. However, for both directions, the induction principle for 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. $\forall A \forall t^{Lam\, A}.\, can\, t \to subst\, var_A\, t = t.$

Proof. Induction on $can\, t$. The variable case even holds with convertibility:

$$subst\, var_A\, (var_A\, a) \simeq var_A\, a \ .$$

Case $app_A\, t_1\, t_2$: by induction hypothesis, $subst\, var_A\, t_i = t_i$ for $i = 1, 2$. One concludes by observing

$$subst\, var_A\, (app_A\, t_1\, t_2) \simeq app_A\, (subst\, var_A\, t_1)\, (subst\, var_A\, t_2) \ .$$

Case $abs_A\, r$: by induction hypothesis, $subst\, var_{opt\, A}\, r = r$. We calculate

$$subst\, var_A\, (abs_A\, r) \simeq abs_A(subst(lift\, var_A)\, r) \ .$$

We conclude from extensionality of *subst* (item 1 in Theorem 1) since case analysis on $opt\, A$ shows: $\forall a^{opt\, A}.\, lift\, var_A\, a = var_{opt\, A}\, a$.

Case $flat_A\, e$: by induction hypothesis, $subst\, var_{Lam\, A}\, e = e$. We know

$$subst\, var_A\, (flat_A\, e) \simeq flat_A(subst(var_{Lam\, A} \circ (subst\, var_A))\, e) \ ,$$

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

$$\forall t^{Lam\, A}.\, t \in FV_{Lam\, A}\, e \to (var_{Lam\, A} \circ (subst\, var_A))\, t = var_{Lam\, A}\, t \ ,$$

but for those t, that have to be hereditarily canonical in order to make $flat_A\, e$ hereditarily canonical, the induction hypothesis provides $subst\, var_A\, t = t$. □

Another result that cannot be proven in general is a refinement of extensionality of $lam\, f\, t$ 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 *can* and elementary reasoning about free variable names:

$$\forall A \forall B \forall f, g^{A \to B} \forall t^{Lam\, A}.\, can\, t \to (\forall a.\, a \in FV\, t \to f\, a = g\, a) \to lam\, f\, t = lam\, g\, t \ .$$

It seems that the relativization to *can* is necessary since *lam*, i. e., $map_{\mu F}$ for our fixed $F := LamF$, is too deeply integrated into the system *LNGMIt* to be amenable to further reasoning that would go beyond the intended and established fact that $map_{\mu F}$ is an extensional functor.

We now address extra closure properties of *can*. Renaming *lam* preserves hereditary canonicity:

$$\forall A \forall B \forall f^{A \to B} \forall t^{Lam\, A}.\, can\, t \to can(lam\, f\, t) \ .$$

This is proven by induction on *can*, and the crucial *flat* case needs the following identification of free variables of $lam\, f\, t$:

$$\forall A \forall B \forall f^{A \to B} \forall t^{Lam\, A} \forall b^{B}.\, b \in FV(lam\, f\, t) \to \exists a^{A}.\, a \in FV\, t \wedge b = fa \ ,$$

which is nearly an immediate consequence of naturality of FV.

Analogously, *subst* preserves hereditary canonicity:

$$\forall A \forall B \forall f^{A \to Lam\, B} \forall t^{Lam\, A}.\, (\forall a^{A}.\, a \in FV\, t \to can(fa)) \to can\, t \to can(subst\, f\, t) \ .$$

Again, this is proven by induction on *can*, and again, the crucial case is with $flat\, e$, for which free variables of $subst\, f\, t$ 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 *lam* and the second part of naturality of *subst* (item 4 of the list, proven by map fusion), one can see *lam* as a special instance of *subst* for hereditarily canonical elements:

$$\forall A \forall B \forall f^{A \to B} \forall t^{Lam\, A}.\, can\, t \to lam\, f\, t = subst\, (var_B \circ f)\, t \ .$$

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

$$\forall A \forall B \forall f^{A \to Lam\, B} \forall e^{Lam(Lam\, A)}.\, can\, e \to subst\, f\, (flat\, e) = flat(lam\, (subst\, f)\, e) \ .$$

5.2. Hereditarily Canonical Terms as a Nested Datatype

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

$$\forall A \forall t^{Lam\, A}.\, can\, t \to Lam'\, A \ ,$$

and π_1 can be defined by pattern-matching on this construction.

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

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

$$\begin{array}{lcl} var' & : & \forall A.\, A \to Lam'\, A \ , \\ app' & : & \forall A.\, Lam'\, A \to Lam'\, A \to Lam'\, A \ , \\ abs' & : & \forall A.\, Lam'(opt\, A) \to Lam'\, A \end{array}$$

from their analogues in Lam. The construction of

$$\mathit{flat}' : \forall A.\, Lam'(Lam'\, A) \to Lam'\, A$$

is as follows: Assume $e : Lam'(Lam'\, A)$. Then, its first projection, $\pi_1\, e$, is hereditarily canonical and of type $Lam(Lam'\, A)$. Therefore, the first projection of $\mathit{flat}'_A\, e$ is taken to be

$$\hat{e} := \mathit{flat}_A(lam\ (\pi_1)_A\ (\pi_1 e)) : Lam\, A\ ,$$

with the renaming with $(\pi_1)_A : Lam'\, A \to Lam\, A$ inside. Thanks to the preservation of hereditary canonicity by lam, the argument $lam\ (\pi_1)_A\ (\pi_1 e)$ to 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 $=$) as $\pi_1\, t$ for a $t \in FV_{Lam'\, A}(\pi_1 e)$, hence is in turn hereditarily canonical. In conclusion, we get $can\, \hat{e}$, which allows to finish the definition of $\mathit{flat}'_A e$ by pairing the term $\hat{e}$ with the proof of $can\, \hat{e}$. By construction, $\pi_1(\mathit{flat}'\, e) \simeq \mathit{flat}\, (lam\, \pi_1\, (\pi_1\, e))$.

Since 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 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 $\simeq$.

From preservation of hereditary canonicity by lam and $subst$, one can easily define $lam' : mon\, Lam'$ and $subst' : \forall A \forall B.\, (A \to Lam'\, B) \to Lam'\, A \to Lam'\, B$, such that $\pi_1(lam'\, f\, t) \simeq lam\, f\, (\pi_1\, t)$ and $\pi_1(subst'\, f\, t) \simeq subst\, (\pi_1 \circ f)\, (\pi_1\, t)$.

The list of free variables is obtained through $FV' : Lam' \subseteq List$, defined by pre-composing FV with π_1, and FV' is then natural with respect to lam' and map (which immediately follows from naturality of FV). One readily verifies the analogues to the recursive description (w. r. t. $\simeq$) of FV:

$$\begin{array}{lcl} FV'\, (var'\, a) & \simeq & [a]\ , \\ FV'\, (app'\, t_1\, t_2) & = & FV'\, t_1 +\!\!+ FV'\, t_2\ , \\ FV'\, (abs'_A\, r) & = & \mathit{filterSome}_A\, (FV'_{opt\, A}\, r)\ , \\ FV'\, (\mathit{flat}'_A\, e) & = & \mathit{flat\_map}\, FV'_A\, (FV'_{Lam'\, A}\, e)\ . \end{array}$$

It is even trivial to lift item 6 of Theorem 1 to Lam', yielding

$$FV'(subst'\, f\, t) = \mathit{flat\_map}\, (FV' \circ f)\, (FV'\, t)\ .$$

One can reprove from naturality of FV' and the preceding equation or simply transfer (using the two equations two paragraphs above) the identification of free variables of $lam\, f\, t$ and $subst\, f\, t$ to lam' and $subst'$, yielding

$$\forall A \forall B \forall f^{A \to B} \forall t^{Lam'\, A} \forall b^B.\, b \in FV'(lam'\, f\, t) \to \exists a^A.\, a \in FV'\, t \wedge b = fa\ ,$$

$$\forall A \forall B \forall f^{A \to Lam'\, B} \forall t^{Lam'\, A} \forall b^B.\, b \in FV'(subst'\, f\, t) \to \exists a^A.\, a \in FV'\, t \wedge b \in FV'(fa)\ .$$

These first results about 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 $Lam'\, A$ 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 $t : \mathit{Lam}\, A$, which amounts to *proof irrelevance*, or work with a "user-defined" equivalence $\equiv_A$ on $\mathit{Lam}\, A$, which we will do: define for $t, u : \mathit{Lam}\, A$

$$t \equiv_A u :\Leftrightarrow \pi_1\, t = \pi_1\, u \ .$$

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, neither in general nor just in adding the "only if" direction above (i. e., injectivity of π_1) through an axiom.

With this equivalence in place, Theorem 1 can be transferred to subst', and also the results of Section 5.1 that were relativized to hereditarily canonical terms now hold unconditionally for Lam', as seen in the following theorem (note that we omitted the non-refined version of extensionality of subst' since it is just a weaker result, and the free variables of $\mathit{subst}'\, f\, t$ have already been determined before in this section).

Theorem 9. *In system LNGMIt, the universal (and well-typed) closure of the following statements can be proven to hold:*

1. $(\forall a.\, a \in FV'\, t \to f a \equiv g a) \to \mathit{subst}'\, f\, t \equiv \mathit{subst}'\, g\, t$
2. $\mathit{lam}'\, g\, (\mathit{subst}'\, f\, t) \equiv \mathit{subst}'\, (\, (\mathit{lam}'\, g) \circ f)\, t$
3. $\mathit{subst}'\, g\, (\mathit{lam}'\, f\, t) \equiv \mathit{subst}'\, (g \circ f)\, t$
4. $\mathit{subst}'\, g\, (\mathit{subst}'\, f\, t) \equiv \mathit{subst}'\, ((\mathit{subst}'\, g) \circ f)\, t$
5. $\mathit{subst}'\, \mathit{var}'_A\, t \equiv t.$
6. $\mathit{lam}'\, f\, t \equiv \mathit{subst}'\, (\mathit{var}' \circ f)\, t.$
7. $\mathit{subst}'\, f\, (\mathit{flat}'\, e) \equiv \mathit{flat}'(\mathit{lam}'\, (\mathit{subst}'\, f)\, e).$

Finally, a monad structure has been obtained.

For completeness, we also mention the recursive description of lam' and subst' paralleling that of lam and subst, and some more properties of lam' that come from lam.

The following recursive decription is the strict analogue of that for lam, but with $\equiv$ in place of $\simeq$.

$$\begin{array}{lcl}
\mathit{lam}'\, f\, (\mathit{var}'_A\, a) & \equiv & \mathit{var}'_B(f a)\ , \\
\mathit{lam}'\, f\, (\mathit{app}'_A\, t_1\, t_2) & \equiv & \mathit{app}'_B\, (\mathit{lam}'\, f\, t_1)\, (\mathit{lam}'\, f\, t_2)\ , \\
\mathit{lam}'\, f\, (\mathit{abs}'_A\, r) & \equiv & \mathit{abs}'_B(\mathit{lam}'\, (\mathit{opt_map}\, f)\, r)\ , \\
\mathit{lam}'\, f\, (\mathit{flat}'_A\, e) & \equiv & \mathit{flat}'_B(\mathit{lam}'\, (\mathit{lam}'\, f)\, e)\ .
\end{array}$$

The proof of the last equation needs extensionality and the second functor law for *lam*. lam' is extensional (even in the refined format that needs the restriction to *can* for *lam*) and satisfies the two functor laws, but everything w. r. t. $\equiv$:

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

We close this collection of results with the strict analogue of the recursive description for *subst*, for which we have to provide a dedicated version of lifting, namely

$$lift' : \forall A \forall B.\, (A \to Lam'\, B) \to opt\, A \to Lam'(opt\, B)$$

that we define such that

$$\begin{array}{lcl} lift'_{A,B}\, f\, None & \simeq & var'_{opt\, B}\, None \ , \\ lift'_{A,B}\, f\,(Some\, a) & \simeq & lam'\, Some\,(f a) \ . \end{array}$$

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

$$\begin{array}{lcl} subst'\, f\,(var'\, a) & \equiv & f a \ , \\ subst'\, f\,(app'\, t_1\, t_2) & \equiv & app'\,(subst'\, f\, t_1)\,(subst'\, f\, t_2) \ , \\ subst'\, f\,(abs'\, r) & \equiv & abs'\,(subst'\,(lift'\, f)\, r) \ , \\ subst'\, f\,(flat'\, e) & \equiv & flat'(subst'(var' \circ (subst'\, f))\, e) \ , \end{array}$$

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

Theorem 10. *The constructors of Lam' are exhaustive in the following sense:*

$$\forall A \forall t^{Lam'\, A}.\, (\exists a.\, t = var'\, a) \vee (\exists t_1, t_2.\, t = app'\, t_1\, t_2) \vee (\exists r.\, t = abs'\, r) \vee (\exists e.\, t \equiv flat'\, e) \ .$$

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 $t : Lam'\, A$ into $t' := \pi_1\, t$ and the proof p of $can\, t'$. 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 *can*. This is why we can even have $=$ in these cases.

Case *flat′*: We are given $e_0 : Lam(Lam\, A)$ and $p : can\, e_0$ and know that all free variable names of e_0 are hereditarily canonical. We have to find a term $e : Lam'(Lam'\, A)$ 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., $flat\, e_0 = \pi_1(flat'\, e)$. We need to come up with a term $e_1 : Lam(Lam'\, A)$ 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 $CAN : Lam \subseteq Lam'$, i. e., $e_1 := lam\, CAN\, e_0$. Since *lam* preserves hereditary canonicity, this is admissible. We must show $flat\, e_0 = \pi_1(flat'\, e)$, but the right-hand side is $flat\, (lam\, \pi_1\, (\pi_1\, e))$ by construction of *flat′*. With $\pi_1\, e \simeq e_1 \simeq lam\, CAN\, e_0$, it suffices to show $e_0 = lam\, \pi_1\, (lam\, CAN\, e_0)$. By the first functor law for *lam*, the left-hand side is equal to $lam\, (\lambda x.\, x)\, e_0$, and by the second functor law for *lam*, the right-hand side is equal to $lam\, (\pi_1 \circ CAN)\, e_0$. We can use the refined extensionality for *lam* since e_0 is hereditarily canonical. It remains to show for every $t \in FV\, e_0$, that $t = \pi_1(CAN\, t)$. We know that those t are in *can*, thus, in order to complete the proof, it suffices to construct a function $CAN : Lam \subseteq Lam'$ such that $\pi_1(CAN\, t) = t$ 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

$$\forall X \forall Y.\, mon\, Y \to X \subseteq Y \to LamF\, X \subseteq LamF\, Y \ .$$

It can be defined as follows by pattern-matching:

$$\begin{array}{rcl}
\mathcal{M} := \lambda X \lambda Y \lambda m^{mon\, Y} \lambda f^{X \subseteq Y} \lambda A \lambda t^{LamF\, X\, A}.\, \mathrm{match}\, t\, \mathrm{with} & & \\
\quad \mid var^-\, a^A & \mapsto & var^-\, a \\
\quad \mid app^-\, t_1^{X\, A}\, t_2^{X\, A} & \mapsto & app^-\, (f_A\, t_1)\, (f_A\, t_2) \\
\quad \mid abs^-\, r^{X\, (opt\, A)} & \mapsto & abs^-\, (f_{opt\, A}\, r) \\
\quad \mid flat^-\, e^{X\, (X\, A)} & \mapsto & flat^-\, (m\, f_A\, (f_{X\, A}\, e)) \ .
\end{array}$$

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

$$Ltc := MIt\, Lam\, \left(\lambda X \lambda it^{X \subseteq Lam} \lambda A \lambda t^{LamF\, X\, A}.\, InCan(\mathcal{M}_{X,Lam}\, lam\, it\, t)\right) \ .$$

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:

$$\begin{array}{lcl}
Ltc\, (var\, a) & \simeq & var\, a \ , \\
Ltc\, (app\, t_1\, t_2) & \simeq & app\, (Ltc\, t_1)\, (Ltc\, t_2) \ , \\
Ltc\, (abs\, r) & \simeq & abs\, (Ltc\, r) \ , \\
Ltc\, (flat_A\, e) & \simeq & flat_A\, (lam\, Ltc_A\, (Ltc_{Lam\, A}\, e)) \ .
\end{array}$$

From these rules, induction on *can* can show that

$$\forall A \forall t^{Lam\, A}.\, can\, t \to Ltc\, t = t \ .$$

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):

$$\forall A \forall t^{Lam\, A}.\, can(Ltc\, t) \ .$$

The proof goes by the induction principle $\mu FInd$.

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

This finishes the proof of Theorem 10. □

6. Conclusions and Future Work

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

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

Certainly, the other laws in, e. g., [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 $\mu FInd$, 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

[1] R. Bird, L. Meertens, Nested datatypes, in: J. Jeuring (Ed.), Mathematics of Program Construction, MPC'98, Proceedings, Vol. 1422 of Lecture Notes in Computer Science, Springer Verlag, 1998, pp. 52–67.

[2] R. Bird, J. Gibbons, G. Jones, Program optimisation, naturally, in: J. Davies, B. Roscoe, J. Woodcock (Eds.), Millenial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symp. in Honour of Professor Sir Anthony Hoare, Palgrave, 2000.

[3] R. Hinze, Efficient generalized folds, in: J. Jeuring (Ed.), Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal, 2000.

[4] F. Bellegarde, J. Hook, Substitution: A formal methods case study using monads and transformations, Science of Computer Programming 23 (1994) 287–311.

[5] R. S. Bird, R. Paterson, De Bruijn notation as a nested datatype, Journal of Functional Programming 9 (1) (1999) 77–91.

[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, CSL '99, Proceedings, Vol. 1683 of Lecture Notes in Computer Science, Springer Verlag, 1999, pp. 453–468.

[7] P. Johann, N. Ghani, A principled approach to programming with nested types in Haskell, Higher-Order and Symbolic Computation 22 (2) (2009) 155–189.

[8] The Coq Development Team, The Coq Proof Assistant Reference Manual Version 8.2, Project TypiCal, INRIA, system available at `coq.inria.fr`. (2009).

[9] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer Verlag, 2004.

[10] C. Paulin-Mohring, Définitions inductives en théorie des types d'ordre supérieur, Habilitation à diriger les recherches, Université Claude Bernard Lyon I (1996).

[11] P. Letouzey, A new extraction for Coq, in: H. Geuvers, F. Wiedijk (Eds.), TYPES 2002 Post-Conference Proceedings, Vol. 2646 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 200–219.
[12] A. Abel, R. Matthes, T. Uustalu, Iteration and coiteration schemes for higher-order and nested datatypes, Theoretical Comput. Sci. 333 (1–2) (2005) 3–66.
[13] R. Bird, R. Paterson, Generalised folds for nested datatypes, Formal Aspects of Computing 11 (2) (1999) 200–222.
[14] A. Abel, R. Matthes, (Co-)iteration for higher-order nested datatypes, in: H. Geuvers, F. Wiedijk (Eds.), TYPES 2002 Post-Conference Proceedings, Vol. 2646 of Lecture Notes in Computer Science, Springer Verlag, 2003, pp. 1–20.
[15] R. Bird, O. de Moor, Algebra of Programming, Vol. 100 of International Series in Computer Science, Prentice Hall, 1997.
[16] R. Matthes, An induction principle for nested datatypes in intensional type theory, Journal of Functional Programming 19 (3&4) (2009) 439–468.
[17] C. Martin, J. Gibbons, I. Bayley, Disciplined, efficient, generalised folds for nested datatypes, Formal Aspects of Computing 16 (1) (2004) 19–35.
[18] R. Matthes, Coq development for "Map fusion for nested datatypes in intensional type theory", `http://www.irit.fr/~Ralph.Matthes/Coq/MapFusion/` (March 2009).
[19] R. Matthes, Coq development for "An induction principle for nested datatypes in intensional type theory", `http://www.irit.fr/~Ralph.Matthes/Coq/InductionNested/` (January 2008).
[20] R. Matthes, Nested datatypes with generalized Mendler iteration: map fusion and the example of the representation of untyped lambda calculus with explicit flattening, in: P. Audebaud, C. Paulin-Mohring (Eds.), Mathematics of Program Construction, Proceedings, Vol. 5133 of Lecture Notes in Computer Science, Springer Verlag, 2008, pp. 220–242.
[21] N. P. Mendler, Recursive types and type constraints in second-order lambda calculus, in: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y., IEEE Computer Society Press, 1987, pp. 30–36.
[22] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, T. Uustalu, Type-based termination of recursive definitions, Mathematical Structures in Computer Science 14 (2004) 97–141.
[23] T. Uustalu, V. Vene, A cube of proof systems for the intuitionistic predicate μ-, ν-logic, in: M. Haveraaen, O. Owe (Eds.), Selected Papers of the 8th Nordic Workshop on Programming Theory (NWPT '96), Vol. 248 of Research Reports, Department of Informatics, University of Oslo, 1997, pp. 237–246.
[24] R. Matthes, Naive reduktionsfreie Normalisierung (translated to English: naive reduction-free normalization), slides of talk on December 19, 1996, given at the Bern Munich meeting on proof theory and computer science in Munich, available at the author's homepage (December 1996).
[25] P. Wadler, Theorems for free!, in: Proceedings of the fourth international conference on functional programming languages and computer architecture, Imperial College, London, England, September 1989, ACM Press, 1989, pp. 347–359.
[26] A. Abel, R. Matthes, Fixed points of type constructors and primitive recursion, in: J. Marcinkowski, A. Tarlecki (Eds.), Computer Science Logic: 18th International Workshop, CSL 2004. Proceedings, Vol. 3210 of Lecture Notes in Computer Science, Springer Verlag, 2004, pp. 190–204.
[27] R. Matthes, Recursion on nested datatypes in dependent type theory, in: A. Beckmann, C. Dimitracopoulos, B. Löwe (Eds.), Logic and Theory of Algorithms, Vol. 5028 of Lecture Notes in Computer Science, Springer Verlag, 2008, pp. 431–446.
[28] S. Mac Lane, Categories for the Working Mathematician, 2nd Edition, Vol. 5 of Graduate Texts in Mathematics, Springer Verlag, 1998.
[29] P. Dybjer, A general formulation of simultaneous inductive-recursive definitions in type theory, The Journal of Symbolic Logic 65 (2) (2000) 525–549.
[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).
[31] A. Abel, A polymorphic lambda-calculus with sized higher-order types, Doktorarbeit (PhD thesis), LMU München (2006).