Recursion on Nested Datatypes in Dependent 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. Nested datatypes are families of datatypes that are indexed over all types and where the datatype constructors relate *different* members of the family. This may be used to represent variable binding or to maintain certain invariants through typing.
In dependent type theory, a major concern is the termination of all expressible programs, so that types that depend on object terms can still be type-checked mechanically. Therefore, we study iteration and recursion schemes that have this termination guarantee throughout. This is not based on syntactic criteria (recursive calls with "smaller" arguments) but just on types ("type-based termination"). An important concern are reasoning principles that are compatible with the ambient type theory, in our case induction principles.
In previous work, the author has proposed an abstract description of nested datatypes together with a mapping operation (like map for lists) and an iterator on the term side and an induction principle on the logical side that could all be implemented within the Coq system (with impredicative Set that is just needed for the justification, not for the definition and the examples). For verification purposes, it is important to have naturality theorems for the obtained iterative functions. Although intensional type theory does not provide naturality in general, criteria for naturality could be established that are met in case studies on "bushes" and representations of lambda terms (also with explicit flattening).
The new contribution is an extension of this abstract description to full primitive recursion and its illustration by way of examples that have been carried out in Coq. Unlike the iterative system, we do not yet have a justification within Coq.

1 Introduction

Nested datatypes [1] are families of datatypes that are indexed over all types and where the datatype constructors relate different members of the family (i. e., at least one datatype constructor constructs a family member from data of a type that refers to a different member of the family). Let κ_0 stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. Then, let κ_1 be the kind of type transformations, hence $\kappa_1 := \kappa_0 \to \kappa_0$. A typical example would be *List* of kind κ_1, where *List* A is the type of finite lists with elements from type A. But *List* is not a nested datatype since the recursive equation for *List*, i. e., $\mathit{List}\,A = 1 + A \times \mathit{List}\,A$, does not relate lists with different indices. A simple example of a nested datatype where an invariant is guaranteed through its definition are the powerlists [2], with recursive equation $\mathit{PList}\,A = A + \mathit{PList}(A \times A)$, where the type $\mathit{PList}\,A$ represents trees of 2^n elements of A with some $n \geq 0$ (that is not fixed) since, throughout this article, we will only consider the least solutions to these equations. The basic example where variable binding is represented through a nested datatype is a higher-order de Bruijn representation of untyped lambda calculus, following ideas of [3–5]. The lambda terms with free variables taken from A are given by $\mathit{Lam}\,A$, with recursive equation

$$\mathit{Lam}\,A = A + \mathit{Lam}\,A \times \mathit{Lam}\,A + \mathit{Lam}(\mathit{option}\,A) \enspace .$$

The first summand gives the variables, the second represents application of lambda terms and the interesting third summand stands for lambda abstraction: An element of $\mathit{Lam}(\mathit{option}\,A)$ (where $\mathit{option}\,A$ is the type that has exactly one more element than A) is seen as an element of $\mathit{Lam}\,A$ through lambda abstraction of that designated extra variable that need not occur freely in the body of the abstraction.

In dependent type theory, a major concern is the termination of all expressible programs. This may be seen as a heritage of polymorphic lambda calculus (system F^ω) that, by the way, is able to express nested datatypes and many algorithms on them [6]. But termination is also of practical concern with dependent types, namely that type-checking should be decidable: If types depend on object terms, object terms have to be evaluated in order to verify types, as expressed in the convertibility rule. Note, however, that this only concerns evaluation within the definitional equality (i. e., convertibility), henceforth denoted by $\simeq$. Except from the above intuitive recursive equations, $=$ will denote propositional equality throughout: this is the equality type that requires proof and that satisfies the Leibniz principle, i. e., that validity of propositions is not affected by replacing terms by equal (w. r. t. $=$) terms.

Here, we study iteration and recursion schemes that have this termination guarantee throughout. Termination is not based on syntactic criteria such as strict positivity and that all recursive calls are done with “smaller” arguments, but just on types (called “type-based termination” in [7]). The article with Abel and Uustalu [6] presents a variety of iteration principles on nested datatypes in this spirit, all within the framework of system F^ω. However, no reasoning principles, in particular no induction principles, were studied there. Newer work by the author [8] integrates rank-2 Mendler iteration into the Calculus of Inductive Constructions [9–11] that underlies the Coq theorem prover [12] and also justifies an induction principle for them. This is embodied in the system *LNMIt*, the “logic for natural Mendler-style iteration”, defined in Section 3.1.

The articles [6, 8] only concern plain iteration. While an extension of primitive Mendler-style recursion [13] to nested datatypes has been described earlier [14], we will present here an extension *LNMRec* of system *LNMIt* by an enriched

Mendler-style recursor where the step term additionally has access to a map term for the unknown type transformation X that occurs there. By way of examples, its merits will be studied. An overview of extensions to *LNMRec* of results established for *LNMIt* in [8] is given. However, the main theorem of [8] is not carried over to the present setting, i. e., we do not yet have a justification within the Calculus of Inductive Constructions. Nevertheless, all the concepts and results have been formalised in the Coq system, using module functors with parameters of a module type that specifies our extension of *LNMRec*. The Coq code is available [15] and is based on [16].

The next section describes two examples of truly nested datatypes. The first with "bushes", treated in Section 2.1, motivates primitive recursion instead of plain iteration and the second about lambda calculus with explicit flattening, treated in Section 2.2, motivates the access to a map term in the defining clauses of an iterative function. Section 3.1 completes the precise definition of *LNMIt* from [8], while Section 3.2 defines the new system *LNMRec* and shows theorems about it. Section 4 describes when and how to define iterative functions with access to a map term in *LNMIt* and establishes a precise relation with the alternative within *LNMRec*.

Acknowledgement: I am grateful to the referees whose valuable feedback helped to improve the presentation and will influence future work even more.

2 Motivating Examples

A nested datatype will be called "truly nested" (non-linear [17]) if the intuitive recursive equation for the inductive family has at least one summand with a nested call to the family name, i. e., the family name appears somewhere inside the type argument of a family name occurrence of that summand. Our two examples will be the bushes [1] and the lambda terms with explicit flattening [18], described as follows:

$$\begin{aligned} \mathit{Bush}\,A &= 1 + A \times \mathit{Bush}(\mathit{Bush}\,A) \enspace , \\ \mathit{LamE}\,A &= A + \mathit{LamE}\,A \times \mathit{LamE}\,A + \mathit{LamE}(\mathit{option}\,A) + \mathit{LamE}(\mathit{LamE}\,A) \enspace . \end{aligned}$$

The last summand in both examples qualifies them as truly nested datatypes; it is even the same nested call pattern. Truly nested datatypes cannot be directly represented in the current version of the Calculus of Inductive Constructions (CIC), as it is implemented in Coq, while the examples of *PList* and *Lam*, mentioned in the first paragraph of the introduction, are now (since version 8.1) fully supported with recursion and induction principles. In these cases, our proposal is more generic but offers less comfort since it has neither advanced pattern matching nor guardedness checking. *PList* and *Lam* are strictly positive, but *Bush* and *LamE* are not even considered to be positive [19] (see [14] for a notion based on polarity that covers these examples). Since there was no system that combined the termination guarantee for recursion schemes on truly nested datatypes with a logic to reason about the defined functions, it seems only natural that examples like *Bush* and *LamE* did not receive more attention. They are studied in detail in [8]. Here, they are recapitulated and developed so as to motivate our new extension *LNMRec* of *LNMIt* that will be defined in Section 3.2.

2.1 Bushes

In order to fit the above intuitive definition of *Bush* into the setting of Mendler-style recursion, the notion of rank-2 functor is needed. Let $\kappa_2 := \kappa_1 \to \kappa_1$. Any constructor F of kind κ_2 qualifies as rank-2 functor for the moment, and $\mu F : \kappa_1$ denotes the generated nested datatype.[1] For bushes, set

$$\mathit{BushF} := \lambda X^{\kappa_1} \lambda A^{\kappa_0}.\, 1 + A \times X(X\,A)$$

and $\mathit{Bush} := \mu\, \mathit{BushF}$. In general, there is just one datatype constructor for μF, namely $\mathit{in} : F(\mu F) \subseteq \mu F$, with the abbreviation $X \subseteq Y := \forall A^{\kappa_0}.\, X A \to Y A$ for any $X, Y : \kappa_1$. For bushes, more clarity comes from two derived datatype constructors

$$\begin{aligned} \mathit{bnil} &: \forall A^{\kappa_0}.\, \mathit{Bush}\,A \ , \\ \mathit{bcons} &: \forall A^{\kappa_0}.\, A \to \mathit{Bush}(\mathit{Bush}\,A) \to \mathit{Bush}\,A \ , \end{aligned}$$

defined by $\mathit{bnil} := \lambda A^{\kappa_0}.\, \mathit{in}\,A\,(\mathit{inl}\,\mathit{tt})$ (with tt the inhabitant of 1 and left injection inl) and $\mathit{bcons} := \lambda A^{\kappa_0} \lambda a^A \lambda b^{\mathit{Bush}(\mathit{Bush}\,A)}.\, \mathit{in}\,A\,(\mathit{inr}(a,b))$ (with right injection inr and pairing notation $(\cdot,\cdot)$).

Our first example of an iterative function on bushes is the function $\mathit{BtL} : \mathit{Bush} \subseteq \mathit{List}$ (*BtL* is a shorthand for *BushToList*) that gives the list of all elements in the bush and that obeys to the following specification:

$$\begin{aligned} &\mathit{BtL}\,A\,(\mathit{bnil}\,A) \simeq [\,] \ , \\ &\mathit{BtL}\,A\,(\mathit{bcons}\,A\,a\,b) \simeq a :: \mathit{flat\_map}_{\mathit{Bush}\,A,A}\,(\mathit{BtL}\,A)(\mathit{BtL}\,(\mathit{Bush}\,A)\,b) \ . \end{aligned}$$

Here, we denoted by $[\,]$ the empty list and by $a :: \ell$ the cons operation on lists, and $\mathit{flat\_map}_{B,A}\,f\,\ell$ is the concatenation of all the A-lists $f\,b'$ for the elements b' of the B-list ℓ. See below why *BtL* is to be called an iterative function.

With the *length* function for lists, we get a function that calculates the size of bushes: $\mathit{size}_i := \lambda A \lambda t^{\mathit{Bush}\,A}.\, \mathit{length}(\mathit{BtL}_A\,t)$. Note that we write the type parameter to *BtL* just as an index, which we will do frequently in the sequel for type-indexed functions—if we do not omit it altogether, e. g., for size_i. The definition of size_i is not iterative[2], but an easy induction on $\mathit{BtL}_{\mathit{Bush}\,A}\,b$ reveals

$$\mathit{size}_i(\mathit{bcons}_A\,a\,b) = S\big(\mathit{fold\_right}_{\mathit{nat},\mathit{Bush}\,A}\,(\lambda x \lambda s.\, \mathit{size}_i\,x + s)\,0\,(\mathit{BtL}_{\mathit{Bush}\,A}\,b)\big) \ ,$$

with S the successor function on the type *nat* of natural numbers and

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

with $\mathit{fold\_right}_{A,B}\,f\,a\,[\,] \simeq a$ and

$$\mathit{fold\_right}_{A,B}\,f\,a\,(b :: \ell) \simeq f\,b\,(\mathit{fold\_right}_{A,B}\,f\,a\,\ell) \ .$$

[1] Strictly speaking, this includes *List* since nesting is not required.
[2] The index in the name size_i stands for *indirect*, not for iterative.

Since we used induction on bushes above, the recursive equation only holds for propositional equality and not for the definitional equality $\simeq$. But we might desire just that, i. e., we might want a recursive version $size_r$ of $size_i$ such that

$$size_r(bcons_A\, a\, b) \simeq S\big(fold\_right_{nat,Bush\,A}\,(\lambda x\lambda s.\, size_r\, x + s)\, 0\, (BtL_{Bush\,A}\, b)\big) \ ,$$

but this is no longer within the realm of iteration in Mendler's style, as we will argue right now. Mendler iteration of rank 2 [6] can be described as follows: There is a constant

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

and the iteration rule

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

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

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

The term s is called the *step term* of the iteration since it provides the inductive step that extends the function from the type transformation X that is to be viewed as approximation to μF (although this is not expressed here!), to a function from FX to G.

Given a step term s, one gets $s\, G\, (\lambda A^{\kappa_0} \lambda x^{GA}.\, x) : FG \subseteq G$ – an F-algebra. Conversely, given an F-algebra $s_0 : FG \subseteq G$, one can construct a step term s if there is a term $M : \forall X^{\kappa_1} \forall G^{\kappa_1}.\, X \subseteq G \to FX \subseteq FG$:

$$s := \lambda X^{\kappa_1} \lambda it^{X \subseteq G} \lambda A^{\kappa_0} \lambda t^{FXA}.\, s_0\, A\, (M\, X\, G\, it\, A\, t).$$

However, a typical feature of truly nested datatypes is that there is no such (closed) term M [6, Lemma 5.3] (but see the notion of relativized basic monotonicity in Section 4). Moreover, the traditional approach with F-algebras does not display the operational behaviour as much as Mendler's style does.

The function BtL is an instance of this iteration scheme with

$$\begin{aligned} BtL := MIt\, List\, \big(\lambda X^{\kappa_1} \lambda it^{X \subseteq List} \lambda A^{\kappa_0} \lambda t^{BushF\, X\, A}.\, &\mathrm{match}\, t\, \mathrm{with}\, inl\, \_ \mapsto [] \\ &\mid inr(a^A, b^{X(X\,A)}) \mapsto a :: flat\_map_{X A,\, A}\, (it\, A)(it\, (XA)\, b)\big) \ . \end{aligned}$$

Note that when the term t of type $BushF\, X\, A$ is matched with $inr(a, b)$, the variable b is of type $X(X\, A)$.[3] This is the essence of Mendler's style: the recursive calls come in the form of uses of it that does not have type $Bush \subseteq List$ but just $X \subseteq List$, and the type arguments of the datatype constructors are replaced by variants that only mention X instead of $Bush$. So, the definitions have to be uniform in that type transformation variable X, but this is already sufficient to guarantee termination (for the rank-1 case of inductive *types*, this has been

[3] The pattern matching could easily be replaced by case analysis on sums and projections for products.

discovered in [20] by syntactic means and, independently, by the author with a semantic construction [21]).

We conclude that BtL is an iterative function in the sense of Mendler but also in a more general sense since Mendler iteration can be simulated by impredicative encodings in system F^ω. In a less technical sense, BtL is iterative as opposed to primitive recursive since the recursive argument b of $bcons$ is only used as an argument of BtL itself. The recursive equation for $size_r$, however, uses b as an argument not of $size_r$, but the previously defined BtL, whose result is then fed element-wise into $size_r$. It seems very unlikely that there is a direct definition of $size_r$ by help of MIt: If, through pattern matching, b is only available with type $X(X\,A)$, the function $BtL_{Bush\,A}$ just cannot be applied to it. Neither could $BtL_{X A}$. The way out is provided already by Mendler for inductive types [13] and has been generalized to nested datatypes in [14]: Express in the step term in addition that X is an approximation of μF, in the sense of an "injection" $j : X \subseteq \mu F$ that is available in the body of the definition. So, we assume a constant (the minus sign indicates that it is a preliminary version)

$$MRec^- : \forall G^{\kappa_1}.\,(\forall X^{\kappa_1}.\,X \subseteq \mu F \to X \subseteq G \to F X \subseteq G) \to \mu F \subseteq G$$

and the recursion rule

$$MRec^-\,G\,s\,A\,(in\,A\,t) \simeq s\,\mu F\,(\lambda A^{\kappa_0}\lambda x^{\mu F A}.\,x)\,(MRec^-\,G\,s)\,A\,t\ \ .$$

(In a more traditional formulation instead of Mendler's style, one would require a recursive F-algebra of type $F(\lambda A^{\kappa_0}.\,\mu F A \times G A) \subseteq G$ instead of a step term s, see [22] for the case of inductive *types*. Again, Mendler's style does not necessitate any consideration of monotonocity of F.) $MRec^-$ allows to program $size_r$ with $G := \lambda A.\,nat$ as

$$MRec^-\,G\,\Big(\lambda X^{\kappa_1}\lambda j^{X\subseteq Bush}\lambda rec^{X\subseteq G}\lambda A^{\kappa_0}\lambda t^{BushF\ X\ A}.\,\text{match}\,t\,\text{with}\,inl\,\_ \mapsto 0$$
$$\mid inr(a^A, b^{X(X\,A)}) \mapsto S\big(fold\_right_{nat,X A}\,(\lambda x\lambda s.\,rec_A\,x + s)\,0\,(BtL_{X A}\,(j_{X A}\,b))\big)\Big).$$

The injection j is an artifact of Mendler's method to enforce termination. In the recursion rule, the term $j_{X A}\,b$ is instantiated by $(\lambda A^{\kappa_0}\lambda x^{Bush\,A}.\,x)\,(Bush\,A)\,b \simeq b$. Therefore, only b appears in the displayed right-hand side of the recursive equation.[4] Viewed from a different angle that already accepts to use Mendler's style, the variable j is there only for type-checking purposes: It allows to get a well-typed step term although it will not be visible afterwards. The advantage of this artifact is that no modification of the ambient type system is needed.

Certainly, we would now like to prove that $\forall A^{\kappa_0}\forall t^{Bush\,A}.\,size_i\,t = size_r\,t$, but this will require our new system $LNMRec$, to be defined in Section 3.2.

Looking back at this little example, we may say that the original function $size_i$ is not itself an instance of Mendler iteration. But there is a recursive equation that can even be made to hold definitionally (with respect to $\simeq$), in form of

[4] In an example that is only mentioned at the end of Section 3.2, the author encountered a natural situation where j is not directly applied to a recursive argument of a datatype constructor, showing again the flexibility of type-based termination.

the instance $size_r$ of the primitive recursor $MRec^-$. But the essential question is how to prove that both functions agree.

2.2 Untyped Lambda Calculus with Explicit Flattening

The untyped lambda terms with explicit flattening are obtained as $LamE := \mu\, LamEF$ with

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

Since the Calculus of Inductive Constructions allows a direct representation of *Lam* (described in the first paragraph of the introduction), we go without *LNMIt* for *Lam* and just assume that we already have an implementation of renaming $lam : \forall A^{\kappa_0} \forall B^{\kappa_0}.\, (A \to B) \to Lam\, A \to Lam\, B$ and parallel substitution

$$subst : \forall A^{\kappa_0} \forall B^{\kappa_0}.\, (A \to Lam\, B) \to Lam\, A \to Lam\, B \ \ ,$$

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 $a : A$ in the term representation $t : Lam\, A$ by the term $f\, a : Lam\, B$. From now on we will no longer decorate variables A, B with their kind κ_0.

The interesting new datatype constructor for *LamE*,

$$flat : \forall A.\, LamE(LamE\; A) \to LamE\; A \ \ ,$$

is obtained by composing *in* with the right injection *inr* (we assume that + in the definition of *LamEF* associates to the left). Its interpretation is an explicit (not executed) form of an integration of the lambda terms that constitute its free variable occurrences into the term itself. This is the monad multiplication form of explicit substitution, as opposed to parallel explicit substitution that would have the type of *subst*, with *Lam* replaced by *LamE*. That other approach would need a quantifier in the rank-2 functor, but also an embedded function space which is not covered by *LNMIt*/*LNMRec* in their current form due to extensionality problems. Moreover, the arising *LamE* would not be a truly nested datatype.

We will review the definition of the iterative function $eval : LamE \subseteq Lam$ that evaluates all the explicit flattenings and thus yields the representation of a usual lambda term. As for bushes, we might want to enforce recursive equations with respect to $\simeq$ that are originally only provable. We do this first with an important property of *subst* and then with naturality (having been proven for Mendler iteration). This will again lead out of the realm of Mendler iteration and motivate a second and conceptually new extension in an orthogonal direction.

Define *eval* by Mendler iteration as

$$\begin{aligned} eval := MIt\big(\lambda X^{\kappa_1} \lambda it^{X \subseteq Lam} \lambda A \lambda t^{F X A}.\, \text{match}\, t\, \text{with} \ \ldots \\ \mid inr\, e^{X(XA)} \mapsto subst_{XA,A}\, it_A\, (it_{XA}\, e)\big) \ \ , \end{aligned}$$

where the routine part has been omitted. This function, introduced in [8], evidently satisfies

$$eval_A(flat_A\, e) \simeq subst_{Lam\, A,A}\; eval_A\, (eval_{LamE\; A}\, e) \ \ .$$

In view of the equation (see [8])

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

the right-hand side is propositionally equal to

$$subst_{Lam\,A,A}\,(\lambda x^{Lam\,A}.\,x)\,(lam\,eval_A\,(eval_{LamE\,A}\,e)) \ .$$

We can easily enforce the latter term to be definitionally equal to the outcome on $flat_A\,e$ for a variant $eval1$ of $eval$, fitting better with what will come later:

$$\begin{aligned} eval1 := MIt\big(\lambda X^{\kappa_1} \lambda it^{X \subseteq Lam} \lambda A \lambda t^{FXA}.\,&\text{match}\,t\,\text{with}\,\ldots \\ &\mid inr\,e^{X(XA)} \mapsto subst\,(\lambda x.\,x)\,(lam\,it_A\,(it_{XA}\,e)\big) \ . \end{aligned}$$

It is an easy exercise to prove in *LNMIt* that *eval1* and *eval* agree propositionally on all arguments.

A natural question for polymorphic functions j of type $X \subseteq Y$ is whether they behave—propositionally—as a natural transformation from (X, mX) to (Y, mY), given map functions[5] $mX : mon\,X$ and $mY : mon\,Y$, where, for any type transformation $X : \kappa_1$, we define

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

Here, the pair (X, mX) is seen as a functor although no functor laws are required. The proposition that defines j to be such a natural transformation is

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

and *LNMIt* allows to prove $eval \in \mathcal{N}(lamE,\, lam)$, with *lamE* the canonical renaming operation for *LamE*, to be provided by *LNMIt* (see Section 3.1). Since [23], naturality is seen as free in pure functional programming because naturality with respect to parametric equality is an instance of the parametricity theorem for types of the form $X \subseteq Y$, but in intensional type theory such as our *LNMIt*, naturality with respect to propositional equality has to be proven on a case by case basis. Since naturality of j only depends propositionally on the values of $j_A\,t$, we also have $eval1 \in \mathcal{N}(lamE,\, lam)$.

It is time to address the second extension of *MIt* that we consider desirable from the point of view of computational behaviour of algorithms on nested datatypes. It does not seem to have been considered previously but is somehow implicit in the author's [8, section 2.3]. It does not even make sense for inductive types but is confined to inductive families.

An instance of naturality of *eval1* is (for $e : LamE(LamE\,A)$)

$$eval1_{Lam\,A}(lamE\,eval1_A\,e) = lam\,eval1_A\,(eval1_{LamE\,A}\,e) \ .$$

In the same spirit as for $size_r$, we might now desire to have the left-hand side as subterm of the definitional outcome of evaluation of $flat_A\,e$ instead of

[5] The name map function comes from the function *map* on lists that is of type *mon List* and that we prefer to call *list*.

the right-hand side, but $\mathit{subst}\,(\lambda x.\,x)\,(\mathit{it}_{\mathit{Lam}\,A}(\mathit{lamE}\;\mathit{it}_A\,e))$ does not type-check in place of the right branch of the definition of *eval1*. As noted in [8], we would need to replace *lamE* by some term $m : \mathit{mon}\,X$, hence a map term for X, but this did not seem available. In fact, it is not available in general (see Section 4 for a discussion under which circumstances it can be constructed). Our proposal is now simply to add such an m to the bound variables that are accessible in the body of the step term. We thus require a constant

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

and the extended iteration rule

$$\mathit{MIt}^+\,G\,s\,A\,(\mathit{in}\,A\,t) \simeq s\,\mu F\,\mathit{map}_{\mu F}\,(\mathit{MIt}^+\,G\,s)\,A\,t\ \ ,$$

with a canonical map term $\mathit{map}_{\mu F} : \mathit{mon}\,\mu F$ that is anyhow provided by *LNMIt* and which is *lamE* in our example. Now, define $\mathit{eval1}' := \mathit{MIt}^+\,s_{\mathit{eval1}'}$ with

$$\begin{aligned} s_{\mathit{eval1}'} := \lambda X^{\kappa_1}\lambda m^{\mathit{mon}\,X}\lambda \mathit{it}^{X\subseteq \mathit{Lam}}\lambda A\lambda t^{F\,X\,A}.\,&\mathrm{match}\,t\,\mathrm{with}\,\ldots\\ &\mid \mathit{inr}\,e^{X(X\,A)} \mapsto \mathit{subst}\,(\lambda x.\,x)\,(\mathit{it}_{\mathit{Lam}\,A}(m\;\mathit{it}_A\,e))\ \ , \end{aligned}$$

which enjoys the desired operational behaviour

$$\mathit{eval1}'_A(\mathit{flat}_A\,e) \simeq \mathit{subst}\,(\lambda x.\,x)\,(\mathit{eval1}'_{\mathit{Lam}\,A}(\mathit{lamE}\;\mathit{eval1}'_A\,e))\ \ .$$

The termination of such a function can be proven by sized nested datatypes [24], but there do not yet exist induction principles for reasoning on programs with sized nested datatypes. A *logic* for Mendler-style recursion that covers our examples will now be described.

Before that, let us note that this second example showed us a situation where a recursive equation with the map term of the nested datatype on the right-hand side may require to abstract over an arbitrary term of type $\mathit{mon}\,X$ in the step term of an appropriately extended notion of Mendler iteration.

3 Logic for Natural Mendler-style Recursion of Rank 2

First, we recall *LNMIt* from [8], then we describe its modifications in order to obtain its extension *LNMRec*.

3.1 *LNMIt*

In *LNMIt*, for a nested datatype μF, we require that $F : \kappa_2$ preserves *extensional functors*. In the Calculus of Inductive Constructions, we may form for $X : \kappa_1$ the dependently-typed record $\mathcal{E}X$ that contains a map term $m : \mathit{mon}\,X$, a proof e of extensionality of m, defined by

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

and proofs f_1, f_2 of the first and second functor laws for (X, m), defined by the propositions

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

Given a record ef of type $\mathcal{E}X$, Coq's notation for its field m is $m\, ef$, and likewise for the other fields. We adopt this notation instead of the more common $ef.m$. Preservation of extensional[6] functors for F is required in the form of a term of type $\forall X^{\kappa_1}.\, \mathcal{E}\, X \to \mathcal{E}(FX)$, and the full definition of *LNMIt* is given as the extension of the predicative Calculus of Inductive Constructions (= pCIC) [12] by the constants and rules in Figure 1, adopted from [8].[7] In *LNMIt*, one can

Parameters:

$F \quad : \kappa_2$

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

Constants:

$\mu F \quad : \kappa_1$

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

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

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

$$\mu FInd : \forall P : \forall A.\, \mu F A \to Prop.\, \Big(\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall n^{j \in \mathcal{N}(m\, ef,\, map_{\mu F})}.\; \big(\forall A \forall x^{XA}.\, P_A(j_A\, x)\big) \to \forall A \forall t^{FXA}.\, P_A(In\, ef\, j\, n\, t)\Big) \to \forall A \forall r^{\mu F A}.\, P_A\, r$$

Rules:

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

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

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

Fig. 1. Specification of *LNMIt*.

show the following theorem [8, 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:

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

Some explanations are in order: The datatype constructor *In* is way more complicated than our previous *in*, but we get back *in* in the form of *InCan* that

[6] While the functor laws are certainly an important ingredient of program verification, the extensionality requirement is more an artifact of our intensional type theory that does not have extensionality of functions in general.

[7] If the rules were formulated with = instead of $\simeq$, *LNMIt* would just be a signature within the pCIC and only a logic without any termination guarantees for the rules.

only constructs the "canonical elements" of the nested datatype μF. The map term $map_{\mu F}$ for μF, which does renaming in the example of *LamE*, is an integral part of the system definition since it occurs in the type of *In*. The Mendler iterator *MIt* has not been touched at all; there is just a more general iteration rule that also covers non-canonical elements, but for the canonical elements, we get the same behaviour, i. e., the same equation with respect to $\simeq$. The crucial part is the induction principle $\mu FInd$, where *Prop* denotes the universe of propositions (all our propositional equalities and their universal quantifications belong to it). Without access to the argument n that assumes naturality of j as a 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 an induction principle that allows to establish naturality of $MIt\, s$ [8, Theorem 1].

BushF and *LamEF* are easily seen to fulfill the requirement of *LNMIt* to preserve extensional functors (using [8, Lemma 1 and Lemma 2]).

3.2 *LNMRec*

Let *LNMRec* be the modification of *LNMIt*, where *MIt* and its two rules are replaced by *MRec* and its proper two rules: the recursor, the rule of primitive recursion and the extensionality rule

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

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

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

The first general consequence of the definition of *LNMRec* is the analogue of [8, Theorem 3] with the primitive recursion rule (generalizing both the rules of $MRec^-$ and MIt^+; recall $in := InCan$)

$$MRec\, G\, s\, (in\, A\, t) \simeq s\, \mu F\, map_{\mu F}\, (\lambda A^{\kappa_0} \lambda x^{\mu F A}.\, x)\, (MRec\, G\, s)\, A\, t \ ,$$

where we are more explicit about all the type parameters.

Evidently, the examples of Section 2 can be formulated in *LNMRec* by not using the argument of type $mon\, X$ in the case of bushes and by not using the injection of type $X \subseteq \mu F$ in the case of evaluation. By using neither monotonicity nor injection, we get back *MIt* and hence may view *LNMRec* as an extension of *LNMIt*.

We continue with two new results that are analogues of results in [8].

Theorem 1 (Naturality of $MRec\, s$). *Assume* $G : \kappa_1$, $mG : mon\, G$, $s : \forall X^{\kappa_1}.\, mon\, X \to X \subseteq \mu F \to X \subseteq G \to FX \subseteq G$ *and that the following holds:*

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E} X} \forall j^{X \subseteq \mu F} \forall rec^{X \subseteq G}.\, j \in \mathcal{N}(m\, ef,\, map_{\mu F}) \to rec \in \mathcal{N}(m\, ef,\, mG) \to s\, (m\, ef)\, j\, rec \in \mathcal{N}(m(Fp\mathcal{E}\, ef),\, mG) \ .$$

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

Proof. By the induction principle $\mu FInd$, as for [8, Theorem 1].

We abbreviate $bush := map_{\mu BushF}$. By using that $BtL \in \mathcal{N}(bush,\, list)$ [8], one can immediately use Theorem 1 to show that $size_r(bush\, f\, t) = size_r\, t$ because this is a naturality statement.

Trivially, the condition on s is simpler for $MIt^+\, s$, namely

$$(*)\ \forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall it^{X \subseteq G}.\, it \in \mathcal{N}(m\, ef,\, mG) \to s\,(m\, ef)\, it \in \mathcal{N}(m(Fp\mathcal{E}\, ef),\, mG)\ .$$

This condition is fulfilled for $s_{eval1'}$ in place of s, hence $eval1' \in \mathcal{N}(lamE,\, lam)$ follows.

Theorem 2 (Uniqueness of $MRec\, s$**).** *Assume* $G : \kappa_1$*,* s *of the type as in the preceding theorem and* $h : \mu F \subseteq G$ *(the candidate for being* $MRec\, s$*). Assume further the following extensionality property of* s*:*

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall j^{X \subseteq \mu F} \forall f, g : X \subseteq G.\, (\forall A \forall x^{XA}.\, f_A\, x = g_A\, x) \to \\ \forall A \forall t^{FXA}.\, s\,(m\, ef)\, j\, f\, t = s\,(m\, ef)\, j\, g\, t\ .$$

Assume finally that h *satisfies the equation for* $MRec\, s$*:*

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

Then, $\forall A \forall r^{\mu F\, A}.\, h_A\, r = MRec\, s\, r$*.*

Proof. By the induction principle $\mu FInd$, as for [8, Theorem 2].

The final question of Section 2.1 is precisely of the form of the conclusion of Theorem 2 (taking into account that $MRec^-$ is just an instance of $MRec$), and its conditions can be shown to be fulfilled, hence $\forall A \forall t^{Bush\, A}.\, size_i\, t = size_r\, t$ follows.[8] By using that $eval1 \in \mathcal{N}(lamE,\, lam)$, one can also apply Theorem 2 in order to show $\forall A \forall t^{LamE\, A}.\, eval1_A\, t = eval1'_A\, t$. Alternatively, one can combine [8, Theorem 2] and $eval1' \in \mathcal{N}(lamE,\, lam)$ for that result. For details, see [15], as before.

The author has carried out other case studies with *LNMRec*. For example, in order to show injectivity of the datatype constructors of *LamE* for application and lambda abstraction, one seems to need to go beyond *LNMIt*, but with $MRec\, s$ in the particular form where the body of s neither uses m nor *rec*, just the injection j. This might be termed Mendler-style *inversion.* There is also a natural example (more natural than the examples in Section 2) – namely a nicer representation of substitution than in [25] – that uses all the ingredients: the map term m, the injection j and the recursive call *rec*. However, this last example needs the ideas of [25] as well and is only an instance of *LNMRec* when *Set* is made impredicative.

[8] Note that, as a function that is only defined by help of *MIt* but not as an instance of *MIt*, the function $size_i$ does not have a characterization that could prove this equation.

4 Access to Map Terms within *MIt*?

Since there is not yet a justification of *LNMRec*, one might want to have the extra liberty of MIt^+ even within *LNMIt*. That is, can one have access to the map term $m : mon\, X$ in the body of s, despite doing a definition with *MIt*? There is no answer in the general situation of *LNMIt*, but under the following conditions that are met in the example of evaluation in Section 2.2.

Assume that F does not only preserve extensional functors, but is also monotone in the following sense (introduced as such in [8] and called relativized basic monotonicity of rank 2): There is a closed term

$$Fmon2br : \forall X^{\kappa_1} \forall Y^{\kappa_1}.\, mon\, Y \to X \subseteq Y \to FX \subseteq FY \ .$$

Assume an extensional functor $ef_G : \mathcal{E}G$ and set $mG := m\, ef_G$. Assume a step term for $MIt^+\, G$, hence $s : \forall X^{\kappa_1}.\, mon\, X \to X \subseteq G \to FX \subseteq G$.

Define $MMIt\, s := MIt\, G\, s' : \mu F \subseteq G$ with

$$s' := \lambda X^{\kappa_1} \lambda it^{X \subseteq G} \lambda A \lambda t^{FXA}.\, s\, G\, mG\, (\lambda A \lambda x^{GA}.\, x)\, (Fmon2br_{X,G}\, mG\, it\, A\, t) \ .$$

In continuation of the example in Section 2.2, we can define

$$eval1'' := MMIt\, s_{eval1'} : LamE \subseteq Lam \ ,$$

since *lam* is extensional and satisfies the functor laws and *LamEF* is monotone in the above sense. With this data, one could see that

$$eval1''_A(flat_A\, e) \simeq subst\, (\lambda x.\, x)\, \big(lam\, (\lambda x.\, x)(lam\, eval1''_A\, (eval1''\, e))\big) \ .$$

This is not too convincing, as far as $\simeq$ is concerned. By the first functor law for *lam*, one immediately gets a propositionally equal right-hand side that corresponds to the recursive equation for *eval1*. Recall that *eval1*$'$ has been derived from *eval1* with the idea of taking profit from naturality, which then justified that they always produce propositionally equal values. If we want to have a general insight about the relation between $MMIt\, s$ and $MIt^+\, s$, we need to ensure naturality. So, assume further condition $(*)$ on s, given after Theorem 1. Moreover, we impose that *Fmon2br* preserves naturality in the following sense: it fulfills

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall it^{X \subseteq G}.\, it \in \mathcal{N}(m\, ef,\, mG) \to \\ Fmon2br\, mG\, it \in \mathcal{N}(m(Fp\mathcal{E}\, ef),\, m(Fp\mathcal{E}\, ef_G)) \ .$$

Then, the naturality theorem for *MIt* [8, Theorem 1] proves naturality of $MMIt\, s$, i. e., $MMIt\, s \in \mathcal{N}(map_{\mu F},\, mG)$.

We are heading towards a proof of $\forall A \forall t^{\mu F A}.\, MMIt\, s\, t = MIt^+\, s\, t$ by Theorem 2. This imposes on us to require the respective extensionality assumption (for an s that does not use the injection facility):

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall f, g : X \subseteq G.\, (\forall A \forall x^{XA}.\, f_A\, x = g_A\, x) \to \\ \forall A \forall t^{FXA}.\, s\, (m\, ef)\, f\, t = s\, (m\, ef)\, g\, t \ .$$

The final condition is that

$$\forall X^{\kappa_1} \forall ef^{\mathcal{E}X} \forall it^{X \subseteq G}.\, it \in \mathcal{N}(m\, ef,\, mG) \to$$
$$\forall A \forall t^{FXA}.\, s\,(m\, ef)\, it\, t = s\, mG\, (\lambda A \lambda x^{GA}.\, x)\, (Fmon2br\, mG\, it\, t) \ ,$$

where the right-hand side is just the body of the definition of s'.

Theorem 3. *Under all the assumptions of the present section, we have that* $\forall A \forall t^{\mu F A}.\, MMIt\, s\, t = MIt^{+}\, s\, t$.

Proof. By Theorem 2.

With this long list of conditions, it is reassuring to verify that Theorem 3 is sufficient to prove that *eval1*$''$ and *eval1*$'$ yield propositionally equal values, and this is the case. It can be shown that, in presence of the other conditions, $(*)$ already follows from its special case $s\, m_G\, (\lambda A \lambda x^{GA}.x) \in \mathcal{N}(m(Fp\mathcal{E}\; ef_G),\, m_G)$.

While the extensions of *MIt* towards *MRec* were dictated by algorithmic concerns, more precisely, by behaviour with respect to definitional equality $\simeq$, this last section contributes more to "type-directed programming": If an argument m of type $mon\, X$ is additionally available in the body of the step term, one may try to use it, just being guided by the wish to produce a term of the right type. We already know that *MIt* can only define terminating functions. So, $MMIt\, s$ will be some terminating function of the right type. If all the requirements of Theorem 3 are met, we even know that $MMIt\, s$ calculates values that we can understand better through their description by $MIt^{+}\, s$. Of course, if there were already a justification of *LNMRec*, we would prefer to use $MIt^{+}\, s$, but this is not yet achieved.

As a final remark, the idea to try to define $MMIt\, s$ arose when studying the article [26] by Johann and Ghani since their type of interpreter transformers `InterpT` only quantifies over all `Functor y`, where the type class mechanism of Haskell is used to express the existence of a map term for the type transformation `y`. However, in that article, this map term is never used. Moreover, instead of monotonicity in the sense of *Fmon2br*, an unrelativized version is used that cannot treat truly nested datatypes such as *Bush* and *LamE*. Finally, the target constructor G was restricted to be a right Kan extension (in order to have the expressive power of generalized refined iteration [6]), and only an algebra—a term of type $FG \subseteq G$—was constructed and not the step term for Mendler-style iteration.

5 Conclusion

Already for the sake of perspicuous behaviour of iterative functions on nested datatypes, one is led to consider extensions of Mendler-style iteration towards Mendler-style primitive recursion with access to a map term for the nested datatype. The logical system *LNMIt* of earlier work by the author is extended to system *LNMRec*, without changing the induction principle that is the crucial element for verification. *LNMIt* could be defined within the Calculus of Inductive Constructions (CIC) with impredicative universe $\kappa_0 := Set$ and propositional proof irrelevance, i. e., with $\forall P : Prop \forall p_1^P \forall p_2^P .\, p_1 = p_2$,[9] and this definition does respect definitional equality $\simeq$. For *LNMRec*, there does not yet exist a justification like that, not even any justification, despite several attempts by the author to extend the construction of [8]. It is the built-in naturality that can not yet be treated. We may call *LMRec* the version of *LNMRec* where the references to naturality are deleted. The ideas of the first half of [8, Section 2.3] are sufficient to define *LMRec* in the CIC with impredicative *Set*. But naturality is important for verification purposes, and thus *LMRec* is not satisfying as a logical system. To conclude, a justification of the logic of *natural* Mendler-style recursion of rank 2 is strongly needed. It would be an instance of the more general problem of adding an impredicative version of simultaneous inductive-recursive definitions [27] to the CIC.

References

1. Bird, R., Meertens, L.: Nested datatypes. In Jeuring, J., ed.: Mathematics of Program Construction, MPC'98, Proceedings. Volume 1422 of Lecture Notes in Computer Science, Springer Verlag (1998) 52–67
2. Bird, R., Gibbons, J., Jones, G.: Program optimisation, naturally. In Davies, J., Roscoe, B., Woodcock, J., eds.: Millenial Perspectives in Computer Science, Proceedings (2000)
3. Bellegarde, F., Hook, J.: Substitution: A formal methods case study using monads and transformations. Science of Computer Programming **23** (1994) 287–311
4. Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming **9**(1) (1999) 77–91
5. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In Flum, J., Rodríguez-Artalejo, M., eds.: Computer Science Logic, 13th International Workshop, CSL '99, Proceedings. Volume 1683 of Lecture Notes in Computer Science, Springer Verlag (1999) 453–468
6. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333**(1–2) (2005) 3–66
7. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical Structures in Computer Science **14** (2004) 97–141
8. Matthes, R.: An induction principle for nested datatypes in intensional type theory. Journal of Functional Programming (2008) To appear.
9. Coquand, T., Paulin, C.: Inductively defined types. In Martin-Löf, P., Mints, G., eds.: COLOG-88, International Conference on Computer Logic. Volume 417 of Lecture Notes in Computer Science, Springer Verlag (1990) 50–66
10. Paulin-Mohring, C.: Inductive definitions in the system Coq – rules and properties. In Bezem, M., Groote, J.F., eds.: Typed Lambda Calculi and Applications, Proceedings. Volume 664 of Lecture Notes in Computer Science, Springer Verlag (1993) 328–345
11. Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation thesis, Université Claude Bernard Lyon I (1996)

[9] Proof irrelevance is needed for proofs of naturality statements and in order to have injectivity of the first projection out of a strong sum.

12. Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.1. Project LogiCal, INRIA. (2006) System available at `coq.inria.fr`.
13. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y., IEEE Computer Society Press (1987) 30–36
14. Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In Marcinkowski, J., Tarlecki, A., eds.: Computer Science Logic: 18th International Workshop, CSL 2004. Proceedings. Volume 3210 of Lecture Notes in Computer Science, Springer Verlag (2004) 190–204
15. Matthes, R.: Coq development for "Recursion on nested datatypes in dependent type theory". `http://www.irit.fr/~Ralph.Matthes/Coq/MendlerRecursion/` (January 2008)
16. Matthes, R.: Coq development for "An induction principle for nested datatypes in intensional type theory". `http://www.irit.fr/~Ralph.Matthes/Coq/InductionNested/` (January 2008)
17. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2) (1999) 200–222
18. Abel, A., Matthes, R.: (Co-)iteration for higher-order nested datatypes. In Geuvers, H., Wiedijk, F., eds.: TYPES 2002 Post-Conference Proceedings. Volume 2646 of Lecture Notes in Computer Science, Springer Verlag (2003) 1–20
19. Pfenning, F., Paulin-Mohring, C.: Inductively defined types in the calculus of constructions. In Main, M.G., Melton, A., Mislove, M.W., Schmidt, D.A., eds.: Mathematical Foundations of Programming Semantics, Proceedings. Volume 442 of Lecture Notes in Computer Science, Springer Verlag (1989) 209–228
20. Uustalu, T., Vene, V.: A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In Haveraaen, M., Owe, O., eds.: Selected Papers of the 8th Nordic Workshop on Programming Theory (NWPT '96). Volume 248 of Research Reports, Department of Informatics, University of Oslo. (May 1997) 237–246
21. Matthes, R.: Naive reduktionsfreie Normalisierung (translated to English: naive reduction-free normalization). Slides of talk on December 19, 1996, given at the Bern Munich meeting on proof theory and computer science in Munich, available at the author's homepage (December 1996)
22. Geuvers, H.: Inductive and coinductive types with iteration and recursion. In Nordström, B., Pettersson, K., Plotkin, G., eds.: Proceedings of the Workshop on Types for Proofs and Programs, Båstad, Sweden. (1992) 193–217 Only published via `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.dvi.Z`.
23. Wadler, P.: Theorems for free! In: Proceedings of the fourth international conference on functional programming languages and computer architecture, Imperial College, London, England, September 1989, ACM Press (1989) 347–359
24. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Doktorarbeit (PhD thesis), LMU München (2006)
25. Matthes, R.: Nested datatypes with generalized mendler iteration: map fusion and the example of the representation of untyped lambda calculus with explicit flattening. In Paulin-Mohring, C., ed.: Mathematics of Program Construction, Proceedings. Lecture Notes in Computer Science, Springer Verlag (2008) Accepted for publication.
26. Johann, P., Ghani, N.: Initial algebra semantics is enough! In Ronchi Della Rocca, S., ed.: Typed Lambda Calculi and Applications (TLCA) 2007, Proceedings. Volume 4583 of Lecture Notes in Computer Science, Springer Verlag (2007) 207–222
27. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. The Journal of Symbolic Logic **65**(2) (2000) 525–549