Verification of programs on truly nested datatypes in intensional type theory

Ralph Matthes
I.R.I.T., Université Paul Sabatier & C.N.R.S.
118 route de Narbonne, F-31062 Toulouse Cedex 9, France
matthes@irit.fr

Abstract

Nested datatypes are families of datatypes that are indexed over all types such that the constructors may relate different family members (unlike the homogeneous lists). Moreover, even the family name may be involved in the expression that gives the index the argument type of the constructor refers to. Especially in this case of true nesting, termination of functions that traverse these data structures is far from being obvious. A joint article with A. Abel and T. Uustalu (TCS 333(1-2), pp. 3-66, 2005) proposes iteration schemes that guarantee termination not by structural requirements but just by polymorphic typing. And they are generic in the sense that no specific syntactic form of the underlying datatype "functor" is required. However, there have not been induction principles for the verification of the programs thus obtained although they are well-known in the usual model of initial algebras on endofunctor categories.
The new contribution is a representation of nested datatypes in intensional type theory (more specifically, in the Calculus of Inductive Constructions) that is still generic, guarantees termination of all expressible programs and has induction principles that allow to prove functoriality of monotonicity witnesses (maps for nested datatypes) and naturality properties of iteratively defined polymorphic functions.

Keywords: nested datatype, induction principle, terminating recursion scheme, proof irrelevance, Coq

1. INTRODUCTION

The algebra of programming [BdM97] 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 [BM98], already intuitively introduced in the abstract, laws have been important from the beginning.

In previous work, the author concentrated on polymorphic lambda-calculi with nested datatypes that guarantee termination of all functions that follow the proposed iteration schemes. See, in particular, the comprehensive article with Abel and Uustalu [AMU05]. Laws were not given. But the schemes were more general than previous work in that they impose minimal conditions on the datatype "functor" F of rank 2 whose least fixed point μF is still a type transformation and not just a type. There is no need to require continuity properties or that F belongs to some given set of higher-order functors that is generated from some closure properties. Since the ambient calculus is not a category, and the "functors" are no functors since no functoriality laws are required, the laws of program transformation and verification were not considered. The present article proposes a combination of both worlds: the world of terminating programs known from type theory and the categorical laws used in advanced functional programming.

The advantages of Mendler's style [Men87] are once more demonstrated (for inductive *types*, i. e., not inductive families, this is amply demonstrated in [Uus98]): The approach is very flexible since no syntactic criterion is applied for termination checking. It is *type-based termination*: the types of the recursive calls ensure that there is no infinite reduction sequence starting with a well-typed term. But there has not been any contribution on the verification of programs in Mendler's style for nested datatypes. For truly nested datatypes, this is even more important since there, termination is very unintuitive while plain heterogeneous families can be used well in the conventional style of iteration that directly follows the concept of initial algebras and that is available in Coq (and other systems).

We want to do verification in the same system in which we write our programs, and we want a termination guarantee. Moreover, we insist on decidable type-checking. Thus, as our ambient calculus, we have chosen the Calculus of Inductive Constructions (CIC), in the current form in which it is implemented in the Coq theorem proving environment [Coq05]. We will only need concepts and features of Coq that are explained in the Coq textbook [BC04].

It will turn out that, after having introduced non-canonical elements into Mendler's style (following [UV02]), Coq supports reasoning very well. Unfortunately, this does not hold for nested datatypes since one programs polymorphic functions on them for which naturality laws are needed if more serious verification is aimed at. In order to enrich Mendler's style, one has to leave the realm of inductive families towards simultaneous inductive-recursive definitions that have been proposed by Dybjer since 1991. The final journal paper is [Dyb00] while Dybjer and Setzer found a finite axiomatization [DS03]. The single inductive-recursive definition we will use will not directly be an instance of these proposals due to the use of impredicativity. It will neither be an instance of Capretta's unpublished note [Cap04] since the simultaneously defined map function involves the inductive family not only in the source type constructor but also in the target type constructor. The idea for our construction is nevertheless taken from Capretta, but the induction principle for the inductive family is genuinely new work.

The next section introduces the important concepts for this article and discusses how Mendler's style for nested datatypes can also be used in the Coq theorem prover. The problems will be discussed and partial solutions sketched. Section 3 contains the precise description of the extension of the CIC we propose under the name LNMIt for "logic of natural Mendler-style iteration". It will be proven in LNMIt that the iterator only produces natural transformations—under well-motivated assumptions, and that the computation rule for the Mendler-style iterator uniquely determines that iterator (again under reasonable assumptions). Section 4 proves that LNMIt can be *defined* within CIC plus proof-irrelevance, while in Section 5, we look back at the canonical elements with which we started in Section 2. Some conclusions are drawn and further work indicated. Coq vernacular files for the results are provided on the author's web page [Mat06a].

2. TOWARDS THE SYSTEM

In this article, the only nested datatypes we study are fixed points of endofunctions on type transformations. More precisely, this will mean the following. Let $\kappa 0$ stand for the universe of (mono-)types that will be interpreted as sets of computationally relevant objects. In impredicative CIC, this will be the sort Set. Hence, $\kappa 0 := Set$. Then, let $\kappa 1$ be the kind of type transformations, hence $\kappa 1 := \kappa 0 \to \kappa 0$. Finally, the endofunctions on type transformations shall be the type constructors of kind $\kappa 2 := \kappa 1 \to \kappa 1$.

In this section, we fix a type constructor F of kind $k2$. It need not be closed and might even just be a variable.[1] We are interested in its least fixed point μF of kind $\kappa 1$. This type transformation μF is to be seen as the inductive family $(\mu F\, A)_{A:\kappa 0}$ where the index runs through all types.

There are different possibilities to specify μF. We follow the Mendler-style formulation for higher kinds that has been embodied in system MIt^ω [AMU05] (that system contains nested datatypes of arbitrary ranks, here we concentrate on the case $\kappa = \kappa 1$ and therefore omit the superscripts altogether). First, we need an abbreviation for polymorphic function space: For $X, Y : \kappa 1$, define the type

$$X \subseteq Y := \forall A : \kappa 0.\, X A \to Y A \ \ .$$

The expression $X \subseteq Y$ is of kind $\kappa 0$ since we work in *impredicative* CIC, i. e., with impredicative $\kappa 0$.[2]

[1] Later on, certain categorically motivated properties and some extensionality will be required.
[2] For practical uses, it will be immaterial that $X \subseteq Y : \kappa 0$ instead of belonging to the sort $Type$ of CIC. However, our construction of μF will be impredicative, whence we must not climb up the $Type$ hierarchy.

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

$$\begin{array}{rcl} in & : & F(\mu F) \subseteq \mu F \ , \\ \mathit{MIt} & : & \forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G \ . \end{array}$$

The reduction rule is

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

Here, $t : F(\mu F)A$ and $s : \forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G$. The latter is called the *step term* of the iteration since it provides the inductive step that extends the function from the type transformation X that is to be viewed as approximation to μF, to a function from FX to G. Here, function means an inhabitant of the universally quantified implication, hence a polymorphic function.

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

$$\mathit{MIt}\, s\, (in\, t) \longrightarrow s\, (\mathit{MIt}\, s)\, t \ .$$

However, it should be kept in mind that the formal parameter X in the type of s is instantiated with μF. In [AMU05], a direct definition within F^ω [Gir72] of a slight reformulation, using a function symbol of arity 1 instead of the constant MIt, is shown. That translation also simulates the reduction rule, in the sense that a reduction step with our new rule is transformed into at least one rewrite step of F^ω. Thus, since this mentioned transformation behaves well with respect to both type and term substitution, strong normalization follows from that of F^ω.

The aim of this work would have been to find a dependently-typed analogue of the elimination rule: a rule in the format of an induction principle that, given some predicate $P : \forall A : \kappa 0.\, \mu F A \to Prop$ with $Prop$ the sort of propositions in CIC, would have allowed to conclude that P holds universally, i. e., proved the proposition $\forall A : \kappa 0 \forall r : \mu F A.\, P_A\, r$ from a suitable inductive step (the argument A is written as an index to P for enhanced clarity; for this purpose, indexing will often be done in the sequel). The author does not have a proposal for such an induction principle that would be justifiable in CIC or a consistent extension thereof.

The way out will be a more liberal datatype constructor than in. The straightforward generalization of the rule μI in [UV97] (later used as the introduction rule of system $UVIT$ in [Mat98] and called $mapwrap$ in [UV02]) to nested datatypes would have a datatype constructor in' of type

$$\forall X : \kappa 1.\, X \subseteq \mu F \to FX \subseteq \mu F \ .$$

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

This single datatype constructor in' specifies the inductive family μF in CIC since the reference to μF in the antecedent is strictly positive. Then, CIC will have canonical elimination rules associated with μF, and Coq will generate them. The minimality scheme for sort Set will be typed by

$$\forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq \mu F \to X \subseteq G \to FX \subseteq G) \to \mu F \subseteq G \ .$$

This is the lifting of the type of Mendler's recursor [Men87] to nested datatypes. And the generated induction principle has a type that supports the following reasoning: Given a predicate P as above, i. e., $P : \forall A : \kappa 0.\, \mu F A \to Prop$, we may deduce P holds universally, i. e.,$\forall A : \kappa 0 \forall r : \mu F A.\, P_A\, r$, if, for every $X : \kappa 1$ and every $j : X \subseteq \mu F$, from the *inductive hypothesis*

$$\forall A : \kappa 0 \, \forall x : XA.\, P_A (j_A\, x)$$

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

$$\forall A : \kappa 0\, \forall t : FXA.\, P_A(in'\, j\, t) \ \ .$$

In other words, the principle is as follows:

$$\forall P : \forall A : \kappa 0.\, \mu F A \to Prop.\, \Big(\forall X : \kappa 1 \forall j : X \subseteq \mu F.\, \big(\forall A : \kappa 0\, \forall x : XA.\, P_A(j_A\, x)\big) \to \\ \forall A : \kappa 0\, \forall t : FXA.\, P_A(in'\, j\, t)\Big) \to \forall A : \kappa 0\, \forall r : \mu F A.\, P_A\, r \ \ .$$

What does the inductive step require in the case of canonical elements, i. e., for $X := \mu F$ and j the polymorphic identity on μF? The inductive hypothesis in this case is—after normalizing away the β-redex that is implicitly done in the CIC—the proposition $\forall A : \kappa 0\, \forall r : \mu F A.\, P_A\, r$, which amounts to the conclusion of the whole induction principle. The induction step in this case is thus a triviality. Therefore, the case that is dedicated to deal with the canonical elements of the family μF does in no way contribute to the induction step. Hence, the conclusion of the induction principle can only be justified from the induction step in the cases that produce non-canonical elements. We conclude that our reasoning is entirely based on non-canonical elements.

By ignoring the additional hypothesis $X \subseteq \mu F$ in the step term of the above-mentioned minimality scheme, we can get back MIt with the original type, and the following equation holds even with respect to convertibility:[3]

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

where $s : \forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G$, $j : X \subseteq \mu F$, $t : FXA$, and $g \circ f$ denotes function composition $\lambda x.\, g(f\, x)$ (for types of f and g that fit together). So, here X is instantiated with X itself and the shown first argument to s gets type $X \subseteq G$, hence both sides of the equation get type GA. Note also that we no longer indicate the type Set of bound variables named A, B or C.

With these iteration and induction principles, one may try to program and verify functions on nested datatypes. This will be especially interesting in the case of truly nested datatypes since they are not directly supported by the CIC. Here, truly nested datatype shall mean: The inductive family has at least one datatype constructor for which one of the argument types has a nested call to the family name, i. e., the family name appears somewhere inside the type argument of the family name occurrence in the argument type of that datatype constructor. A well-known example of true nesting is given by $Bush$, specified as follows [BM98]:

$$\begin{array}{rcl} bnil & : & \forall A.\, Bush\, A \ \ , \\ bcons & : & \forall A.\, A \to Bush(Bush\, A) \to Bush\, A \ \ . \end{array}$$

The second argument to $bcons$ has type $Bush(Bush\, A)$. Here $Bush$ occurs with argument $Bush\, A$ that makes a reference to $Bush$. Another canonical example is a higher-order representation of deBruijn terms with an explicit notion of flattening, see [AMU05]. Another extension of deBruijn terms that yields a truly nested datatype $TermE$ is described in [BP99].

[AMU05] shows some programs on nested datatypes (including on truly nested datatypes) but does not at aim at verifying them. It turns out, however, that the induction principle of the system just described would be too weak for that purpose. Here is an intuitive argument why this is so: One of the first programs for a nested datatype is a map function that applies some function $f : A \to B$ to all elements of type A contained in the data structure of type $\mu F\, A$:

$$map : \forall A \forall B.\, (A \to B) \to \mu F\, A \to \mu F\, B \ \ .$$

And, certainly, one would like to establish the categorical laws for map, namely that map behaves as the morphism part of a functor whose object part is just μF. With our induction principle, this will not be possible since we do not know anything about the parameter X itself.

[3]This rule also appeared in Peter Aczel's presentation at TYPES 2003.

The first main idea here is to require that X is accompanied by some map term $m : mon\,X$ where

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

and require that m is functorial: It satisfies

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

The equality sign $=$ means propositional equality (Leibniz equality) that is just an inductive definition in the CIC. It is the basis of the rewriting mechanism of Coq that goes beyond the built-in and automatic convertibility relation which is implemented as one fixed strongly normalizing and confluent rewrite system. Universally quantified equations such as $fct_1\,m$ and $fct_2\,m$ have the impredicative kind $Prop$ of computationally irrelevant types. Since Set is sufficiently impredicative, we may place such equations as further premisses into our datatype declarations. This will not be sufficient, though.

Rewriting in Coq with Leibniz equality cannot take place under binders such as λ-abstraction, unlike convertibility that can be applied to arbitrary subterms: Propositional equality is not extensional since the CIC is an intensional type theory. Since the additional functoriality conditions are just equations and therefore do not affect convertibility, truly nested datatypes with their tendency to favour programming with functional arguments (see [AMU05]) call for a special attention to means to ensure that rewriting can nevertheless take place. The second main idea here is that most of the functionals that occur during programming only depend of the extension of their function arguments. This is typically so for map terms, hence we define

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

With these definitions, we might now *define* μF by

$$in'' : \forall X : \kappa 1\,\forall m : mon\,X.\, ext\,m \to fct_1\,m \to fct_2\,m \to X \subseteq \mu F \to FX \subseteq \mu F \enspace .$$

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

In order to avoid overly long formulae in the sequel, the map term m and the three proofs e, f_1, f_2 of $ext\,m$, $fct_1\,m$ and $fct_2\,m$ are organized as a dependently typed record $efct\,X$ (expressing that X is an extensional functor) where the type of the fields e, f_1, f_2 depends on the field m. Given a record ef, i. e., an element of type $efct\,X$, 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 $ef.m$.

Canonical elements can now the obtained from the following preservation property for F: There has to be a term

$$Fpefct : \forall X : \kappa 1.\, efct\,X \to efct(FX).$$

Note that it would be too demanding to require that monotone X's were transported to monotone FX's and that extensionality and the functoriality properties were each preserved separately. In particular, advanced examples require extensionality in order to establish functoriality.

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

However, this approach will not allow to prove that, for a monotone target constructor G—with map term $mG : mon\,G$—and a step term $s : \forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G$ with "reasonable" properties (see below), $MIt\,s : \mu F \subseteq G$ behaves like a natural transformation from $(\mu F, map_{\mu F})$

to (G, mG). In general, for $X, Y : \kappa 1$, $mX : mon\, X$, $mY : mon\, Y$ and $j : X \subseteq Y$, we define the proposition

$$j \in NAT(mX, mY) := \forall A \forall B \forall f : A \to B \forall t : X A.\, j_B\, (mX\, A\, B\, f\, t) = mY\, A\, B\, f\, (j_A\, t) \ \ ,$$

which just says that j is a natural transformation from (X, mX) to (Y, mY). Note that functoriality of either side is not required.

The "reasonable" property above would have been

$$\forall X : \kappa 1 \forall ef : efct\, X \forall j : X \subseteq G.\, j \in NAT(m\, ef, mG) \to s\, j \in NAT(m(Fpefct\, ef), mG) \ \ .$$

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

$$\forall X : \kappa 1\, \forall ef : efct\, X \forall j : X \subseteq \mu F.\, j \in NAT(m\, ef, map_{\mu F}) \to F X \subseteq \mu F \ \ .$$

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

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

So, we may say that μF is the source type constructor of $map_{\mu F}$, and that the recursion is over μF. Unfortunately, the target type constructor G involves μF again, which excludes this situation from being covered by [Cap04]. Nevertheless, it is a simultaneous inductive-recursive definition in a broad sense, and Capretta's idea is applicable.

3. THE SYSTEM

We will call $LNMIt$ ("logic for natural Mendler-style iteration of rank 2")[4] the extension of CIC by the following ingredients and later prove that they can already be defined in CIC plus proof-irrelevance (in sort $Prop$, every proposition has at most one proof):

Assume $F : \kappa 2$ and $Fpefct : \forall X : \kappa 1.\, efct\, X \to efct(F X)$.

Then there are $\mu F : \kappa 1$ and $map_{\mu F} : mon(\mu F)$ with datatype constructor In of the type (already shown above)

$$\forall X : \kappa 1\, \forall ef : efct\, X \forall j : X \subseteq \mu F.\, j \in NAT(m\, ef, map_{\mu F}) \to F X \subseteq \mu F,$$

with iterator MIt of type (the one shown before)

$$\forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq G \to F X \subseteq G) \to \mu F \subseteq G \ \ ,$$

with the induction principle $\mu FInd$ of type

$$\begin{aligned}\forall P : \forall A.\, \mu F A \to Prop.\, \Big(\forall X : \kappa 1 \forall ef : efct\, X\, \forall j : X \subseteq \mu F \forall n : j \in NAT(m\, ef, map_{\mu F}).\\ \big(\forall A \forall x : X A.\, P_A(j_A\, x)\big) \to \forall A \forall t : F X A.\, P_A(In\, ef\, j\, n\, t)\Big)\\ \to \forall A \forall r : \mu F A.\, P_A\, r\end{aligned}$$

and the following equations that are contained in the convertibility relation:

$$\begin{aligned} map_{\mu F}\, f\, (In\, ef\, j\, n\, t) &= In\, ef\, j\, n\, (m(Fpefct\, ef)\, f\, t) \ \ ,\\ MIt\, s\, (In\, ef\, j\, n\, t) &= s\, (\lambda A.\, (MIt\, s)_A \circ j_A)\, t \ \ .\end{aligned}$$

In $LNMIt$, we can now prove the following theorem that is not provable in the systems of Section 2:

[4]For rank 1, naturality does not make any sense because, for inductive *types*, only a function from a monotype μF to a monotype B is defined.

Theorem 1 (Naturality of $\mathit{MIt}\,s$) *Assume $G : \kappa 1$, $mG : mon\, G$, $s : \forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G$ and that the following holds:*

$$\forall X : \kappa 1 \forall ef : efct\, X \forall j : X \subseteq G.\, j \in \mathit{NAT}(m\, ef, mG) \to s\, j \in \mathit{NAT}(m(Fpefct\, ef), mG) \ .$$

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

Proof. This is done by induction with

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

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

$$\forall A \forall x : XA \forall B \forall f : A \to B.\, MIt\, s(map_{\mu F}\, f\, (j_A\, x)) = mG\, f\, (MIt\, s\, (j_A\, x)) \ .$$

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

$$\mathit{MIt}\, s\, (map_{\mu F}\, f(\mathit{In}\, ef\, j\, n\, t)) = mG\, f\, (\mathit{MIt}\, s\, (\mathit{In}\, ef\, j\, n\, t)) \ .$$

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

$$s\, aux\, (m\, (\mathit{Fpefct}\, \mathit{ef})\, f\, t) = mG\, f\, (s\ aux\ t) \ .$$

We want to apply the assumption of the theorem. It suffices to show $aux \in \mathit{NAT}(m\, ef, mG)$. Assume A, B, f, t. Show $aux_B(m\, ef\, f\, t) = mG\, f\, (aux_A\, t)$. Its left-hand side is equivalent to

$$\mathit{MIt}\, s\, (j_B(m\, ef\, f\, t)) = \mathit{MIt}\, s\, (map_{\mu F}\, f\, (j_A\, t)) \ ,$$

where we used the assumption n of type $j \in \mathit{NAT}(m\, ef, map_{\mu F})$ for the last step. Now, the induction hypothesis is applicable. □

Moreover, under reasonable assumptions, $\mathit{MIt}\, s$ is uniquely characterized by the equation above:

Theorem 2 (Uniqueness of $\mathit{MIt}\, s$) *Assume $G : \kappa 1$, $s : \forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G$ and $h : \mu F \subseteq G$ (the candidate for being $\mathit{MIt}\, s$). Assume further the following extensionality property of s (s only depends on the extension of its function argument):*

$$\forall X : \kappa 1 \forall f, g : X \subseteq G.\, (\forall A \forall x : XA.\, f\, x = g\, x) \to \forall A \forall y : FXA.\, s\, f\, y = s\, g\, y \ .$$

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

$$\forall X : \kappa 1 \forall ef : efct\, X\, \forall j : X \subseteq \mu F \forall n : j \in \mathit{NAT}(m\, ef, map_{\mu F}) \\ \forall A \forall t : FXA.\, h_A(\mathit{In}\, ef\, j\, n\, t) = s\, (\lambda A.\, h_A \circ j_A)\, t \ .$$

Then, $\forall A \forall r : \mu F\, A.\, h_A\, r = \mathit{MIt}\, s\, r$.

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

$$h_A(\mathit{In}\, ef\, j\, n\, t) = \mathit{MIt}\, s\, (\mathit{In}\, ef\, j\, n\, t).$$

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

$$s\, (\lambda A.\, h_A \circ j_A)\, t = s\, (\lambda A.\, (\mathit{MIt}\, s)_A \circ j_A)\, t \ .$$

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

$$\forall A \forall x : XA.\, (h_A \circ j_A)\, x = ((\mathit{MIt}\, s)_A \circ j_A)\, x \ ,$$

but this is the induction hypothesis. □

Note that this uniqueness theorem would have been available also in the theory in Section 2 that did not integrate naturality into the approximations to μF.

4. JUSTIFICATION

Theorem 3 (Main Theorem) *The system LNMIt can be defined within the CIC, extended by the principle of proof irrelevance, i. e., by $\forall P : \mathit{Prop}\, \forall p_1, p_2 : P.\, p_1 = p_2$.*

The proof will occupy the remainder of this section. First, assume $F : \kappa 2$. Capretta's idea [Cap04] is to introduce first something bigger than the desired μF, i. e., a type transformation $\mu^+ F$ such that, later, there is a function of type $\mu F \subseteq \mu^+ F$. In fact, μF will be defined as the restriction of $\mu^+ F$ by some predicate, and the mentioned function will just be the first projection out of that strong sum type. While $\mu^+ F$ will not be a "real" recursive type—there is no recursive call to $\mu^+ F$, hence it is just a record—that predicate is defined inductively with induction hypotheses that are in no way a priori smaller than the conclusion.

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

$$\begin{aligned} &\mathit{In}^+ : \forall G : \kappa 1 \forall ef : \mathit{efct}\, G\, \forall G' : \kappa 1 \forall m' : \mathit{mon}\, G' \\ &\qquad\qquad \forall it : \mathit{MItPretype}\, G'\, \forall j : G \subseteq G'.\, j \in \mathit{NAT}(m\, ef, m') \to FG \subseteq \mu^+ F, \end{aligned}$$

with

$$\mathit{MItPretype}\, S := \forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq G \to FX \subseteq G) \to S \subseteq G \ .$$

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

The minimality scheme for sort Set generated from In^+ by Coq is just case analysis on this record-like $\mu^+ F$. By its help, we can immediately define $\mathit{map}_{\mu^+ F} : \mathit{mon}\, \mu^+ F$ with

$$\mathit{map}_{\mu^+ F}\, f\, (\mathit{In}^+\, ef\, m'\, it\, j\, n\, t) = \mathit{In}^+\, ef\, m'\, it\, j\, n\, (m(\mathit{Fpefct}\, ef)\, f\, t)$$

that holds even w. r. t. convertibility. Similarly, one defines $\mathit{MIt}^+ : \mathit{MItPretype}(\mu^+ F)$ such that

$$\mathit{MIt}^+\, s\, (\mathit{In}^+\, ef\, m'\, it\, j\, n\, t) = s\, (\lambda A.\, (it\, s)_A \circ j_A)\, t$$

again also holds within the convertibility relation. Evidently, this has nothing to do with iteration since there is no recursive call whatsoever.

With $\mathit{map}_{\mu^+ F}$ and MIt^+ in place, it can now be defined what is a "good" element of $\mu^+ F$. Following Capretta [Cap04], this is done by way of an inductive predicate $\mathit{chk}_{\mu^+ F} : \forall A.\, \mu^+ F\, A \to \mathit{Prop}$ for which there is a single inductive clause Inchk of type

$$\begin{aligned} &\forall G : \kappa 1 \forall ef : \mathit{efct}\, G\, \forall j : G \subseteq \mu^+ F\, \forall n : j \in \mathit{NAT}(m\, ef, \mathit{map}_{\mu^+ F}).\, \big(\forall A \forall t : GA.\, \mathit{chk}_{\mu^+ F}(j_A\, t)\big) \to \\ &\qquad\qquad \forall A \forall t : FGA.\, \mathit{chk}_{\mu^+ F}\Big(\mathit{In}^+\, ef\, \mathit{map}_{\mu^+ F}\, \mathit{MIt}^+\, \big(\lambda A \lambda t : GA.\, j_A\, t\big)\, n\, t\Big) \ . \end{aligned}$$

This is a strictly-positive inductive definition, hence Coq generates an induction principle as follows. Given a predicate $P : \forall A.\, \mu^+ F\, A \to \mathit{Prop}$, P holds universally, if the following induction step is provided:

$$\begin{aligned} &\forall G : \kappa 1 \forall ef : \mathit{efct}\, G\, \forall j : G \subseteq \mu^+ F\, \forall n : j \in \mathit{NAT}(m\, ef, \mathit{map}_{\mu^+ F}). \\ &\qquad \big(\forall A \forall t : GA.\, \mathit{chk}_{\mu^+ F}(j_A\, t)\big) \to \big(\forall A \forall t : GA.\, P_A(j_A\, t)\big) \to \forall A \forall t : FGA. \\ &\qquad\qquad P_A\Big(\mathit{In}^+\, ef\, \mathit{map}_{\mu^+ F}\, \mathit{MIt}^+\, \big(\lambda A \lambda t : GA.\, j_A\, t\big)\, n\, t\Big) \ . \end{aligned}$$

The premise $\forall A \forall t : GA.\, \mathit{chk}_{\mu^+ F}(j_A\, t)$ yields the inversion principle for $\mathit{chk}_{\mu^+ F}$, the premise $\forall A \forall t : GA.\, P_A(j_A\, t)$ is the induction hypothesis.

Note that the η-expansion $\lambda A \lambda t : GA.\, j_A\, t$ of j is needed for the later proof. Note also that no *recursion* principle is possible for $\mathit{chk}_{\mu^+ F}$ since it would inject elements of Prop into the realm of Set.

Slightly sloppily, we can say that an element of $\mu^+F\,A$ is "good" if it is of the form $In^+\ldots$ where the argument m' is replaced by map_{μ^+F} and it is replaced by MIt^+ and all the j-images are already "good". Finally, set

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

This notation stands for the inductively defined sig of Coq which is a strong sum in the sense that the first projection yields the element r and the second projection the proof that $chk_{\mu^+F}\,r$.

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

Since map_{μ^+F} is not recursive at all, $map_{\mu F}$ is recursive neither.The required recursive behaviour will only come from a definition of In that involves map_{μ^+F}. In order to define In of the required type, assume $X : \kappa1$, $ef : efct\,X$, $j : X \subseteq \mu F$, $n : j \in NAT(m\,ef, map_{\mu F})$, A and $t : FXA$. We have to define $In\,ef\,j\,n\,t : \mu F\,A$. Its first component of type $\mu^+F\,A$ is given by $In^+\,ef\,map_{\mu^+F}\,MIt^+\,j'\,n'\,t$ with $j' : X \subseteq \mu^+F$ defined by typewise composing the first projection out of μF with j, and n' its canonical naturality proof that depends on n and the fact that the first projection of $map_{\mu F}\,f\,r$ is defined to be map_{μ^+F}, applied to f and the first projection of r. The second component, i. e., the proof part, again direct follows from the introduction rule for chk_{μ^+F} since, by the very definition of μF, we have $\forall A \forall t : XA.\, chk_{\mu^+F}(j'_A\,t)$.[5]

Even w. r. t. convertibility, this construction fulfills the required equation for $map_{\mu F}$. To understand better why this equation can be recursive now, note that it is not just an equation about $map_{\mu F}$ but a relation between $map_{\mu F}$ and In. Since map_{μ^+F} entered both parts, the possible "miracle" happens.

The same peculiarity occurs with the definition of MIt, but this is easier than for $map_{\mu F}$. Just define, given $s : \forall X : \kappa1.\, X \subseteq G \to FX \subseteq G$ and $r : \mu F\,A$, the term $MIt\,s\,r : GA$ as MIt^+, applied to s and the first projection of r. The desired equality for MIt holds even as convertibility since composition is associative also in this sense.

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

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

However, it is well-known that there are models of CIC that are proof-irrelevant, and this does by no means degenerate the computational world inside sort Set.

The following proof has no counterpart in Capretta's work [Cap04]. It seems that it profits from our very special situation while Capretta intended to give a general method for simultaneous inductive-recursive definitions.

In order to prove $\mu FInd$, assume the predicate P, the term s representing the inductive step and $A : Set$, $r : \mu F\,A$. We have to show $P_A\,r$. The term r decomposes into a term $: \mu^+F\,A$

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

and a proof $p : chk_{\mu^+F}\, r'$. We do induction on p. We write $cons$ for the opposite operation of this decomposition, hence $cons : \forall A \forall r' : \mu^+ F\, A.\, chk_{\mu^+F}\, r' \to \mu F\, A$. Thus, we want to show $P_A\,(cons\, r'\, p)$ by induction on p. As such, this is not covered by the given induction principle for chk_{μ^+F}. But Coq may also generate a more dependent version (induction scheme for sort $Prop$): Given a predicate $P' : \forall A \forall r' : \mu^+ F\, A.\, chk_{\mu^+F}\, r' \to Prop$, P' holds universally, i. e., $\forall A \forall r' : \mu^+ F\, A \forall p : chk_{\mu^+F}\, r'.\, P'_A\, r'\, p$, if

$$\begin{aligned}\forall G : \kappa 1 \forall ef : efct\, G \forall j : G \subseteq \mu^+ F \forall n : j \in NAT(m\, ef, map_{\mu^+F}) \forall k : \big(\forall A \forall t : GA.\, chk_{\mu^+F}(j_A\, t)\big).\\ \big(\forall A \forall t : GA.\, P'_A(j_A\, t)(k_A\, t)\big) \to \forall A \forall t : FGA.\\ P'_A\Big(In^+\, ef\, map_{\mu^+F}\, MIt^+\, \big(\lambda A \lambda t : GA.\, j_A\, t\big)\, n\, t\Big)(Inchk\, ef\, j\, n\, k\, t)\ .\end{aligned}$$

Here, $P'_A\, r'\, p := P_A\,(cons\, r'\, p)$. Assume G, ef, j, n, k according to this induction principle. Define $j' := \lambda A \lambda t : GA.\, cons(j_A\, t)(k_A\, t)$ of type $G \subseteq \mu F$. Assume $H : \big(\forall A \forall t : GA.\, P_A(j'_A\, t)\big)$, $A : Set$ and $t : FGA$. We have to prove

$$P_A\Big(cons\Big(In^+\, ef\, map_{\mu^+F}\, MIt^+\, \big(\lambda A \lambda t : GA.\, j_A\, t\big)\, n\, t\Big)(Inchk\, ef\, j\, n\, k\, t)\Big)\ .$$

First show $j' \in NAT(m\, ef, map_{\mu F})$. For this, one has to remove the outer quantifiers and then use proof-irrelevance in showing the equation only for the first projections. But this follows by a short calculation from our naturality proof n. Let n_1 be this proof of $j' \in NAT(m\, ef, map_{\mu F})$. We deduce $P_A(In\, ef\, j'\, n_1\, t)$ from the general assumption s and our assumption H.

It also holds that

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

is Leibniz-equal to the first projection of $In\, ef\, j'\, n_1\, t$. This is a simple calculation for the "j argument", and we identify all naturality proofs in our system. Since we identify elements of $\mu F\, A$ with the same first component, the two arguments of P_A in this proof development are equal, hence we may pass from the validity of the second such statement to that of the first one. Again, all the details can be found in the Coq development [Mat06a]. □

5. BACK TO CANONICAL ELEMENTS

Why did we require for $LNMIt$ not only a map-preserving function that takes $m : mon\, X$ to some term of type $mon\,(F\, X)$? Up to now, only the m components of the extensional functors ef have been used. Since we want to construct canonical elements, we also need the properties—just as in the preliminary system in section 2.

Theorem 4 (Canonical Elements in $LNMIt$) *There is a term $InCan : F(\mu F) \subseteq \mu F$ (the* canonical *datatype constructor that constructs canonical elements) such that the following equations are even contained in the convertibility relation:*

$$\begin{aligned} map_{\mu f}\, f\,(InCan\, t) &= InCan(m\,(Fpefct\, ef_{\mu F})\, f\, t)\ ,\\ MIt\, s\,(InCan\, t) &= s\,(MIt\, s)\, t\ ,\end{aligned}$$

for some term $ef_{\mu F} : efct\, \mu F$. Thus, for canonical elements, we get back the ordinary behaviour.

Proof. We want to take μF as its own approximation, with $map_{\mu F}$ as the map term, hence with a term $ef_{\mu F} : efct\, \mu F$ s. t. $m\, ef_{\mu F} = map_{\mu F}$. Therefore, we need to establish ext, fct_1 and fct_2 for $map_{\mu F}$. Then, trivially, the polymorphic identity on μF serves as argument j to In, and (lambda-abstracted) reflexivity of equality yields the respective proof of naturality. Then, the claimed equations follow from those of $LNMIt$ and the fact that $MIt\, s$ is a λ-abstraction, hence composition with the identity can be β-reduced away.

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

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

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

$$map_{\mu F}\, f\, (In\, ef\, j\, n\, t) = map_{\mu F}\, g\, (In\, ef\, j\, n\, t) \enspace .$$

By convertibility, this amounts to

$$In\, ef\, j\, n\, (m\, (Fpefct\, ef)\, f\, t) = In\, ef\, j\, n\, (m\, (Fpefct\, ef)\, g\, t) \enspace ,$$

which follows from $e\, (Fpefct\, ef) : ext(m\, (Fpefct\, ef))$. □

It should be noted that the term $m\, (Fpefct\, ef_{\mu F})$ in the behaviour of $map_{\mu F}$ can usually be simplified to $Fpmon\, map_{\mu F}$, namely when there is a map-transforming function $Fpmon$ of type $\forall X : \kappa 1.\, mon\, X \to mon\, (F\, X)$ such that for all X and $ef : efct\, X$, $m\, (Fpefct\, ef) = Fpmon\, (m\, ef)$, i. e., when the map term for $F\, X$ does not depend on the properties of the map term for X. This is usually the case and leads to the standard behaviour of $map_{\mu F}$ that is programmed in functional programming languages that do not guarantee termination, unlike our approach.

6. CONCLUSIONS

It is now possible to combine the following benefits:

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

In fact, the whole development has been formalized in Coq, and it has been tested with the example of 3-bushes [Mat06b]. However, the final version of that article does no longer use Mendler-style iteration but confines itself to conventional style (motivated from initial algebras). A fortiori, these little algorithms can be presented in Mendler's style, and the “reasonable” application conditions of our theorems are met.

In practice, one often uses refined forms of iteration that are also known under the names of efficient folds [Hin00, MGB04]. In [AMU05], some more general forms in the spirit of Mendler's style are studied, e. g., the system $MIt^{\omega}_{=}$. Its iterator can be expressed by MIt of this article, but only at the expense of some right Kan extension as target type constructor G. Although our Theorem 1 would apply, its application condition would speak about equality of functions, and that is usually not provable. With some more refined notions of extensionality and more careful use of quantification, it is nevertheless possible to prove a theorem for MIt that will yield a naturality property for $MIt_{=}$ that precisely captures the map fusion law. This can even be extended to treat $GMIt$, a more liberal form of $MIt_{=}$ introduced in [AMU05].

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

Finally, it should be mentioned that the present work does not restrict the system to one single nested datatype or only the introduction of one such datatype after the other. All the constructions are fully parametric so that arbitrary interleaving of such families (in the style of generalized rose trees) is admissible.

Acknowledgements: Several suggestions by the referees for the improvement of the presentation are gratefully acknowledged. Without financial support by the FP6 coordination action 510996 “TYPES”, this work would not have been published.

REFERENCES

[[AMU05]] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.

[[BC04]] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[[BdM97]] Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.

[[BM98]] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer Verlag, 1998.

[[BP99]] Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[[Cap04]] Venanzio 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.

[[Coq05]] Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. Project LogiCal, INRIA, January 2005. System available at `coq.inria.fr`.

[[DS03]] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1–47, 2003.

[[Dyb00]] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(2):525–549, 2000.

[[Gir72]] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.

[[Hin00]] Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal*, 2000.

[[Mat98]] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. Doktorarbeit (PhD thesis), University of Munich, 1998. Available via the homepage `http://www.irit.fr/~Ralph.Matthes/`.

[[Mat06a]] Ralph Matthes. Coq development for "Verification of Programs on Truly Nested Datatypes in Intensional Type Theory". `http://www.irit.fr/~Ralph.Matthes/Coq/MSFP06/`, June 2006.

[[Mat06b]] Ralph Matthes. A datastructure for iterated powers. In Tarmo Uustalu, editor, *Mathematics of Program construction. Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 299–315. Springer Verlag, 2006. To appear.

[[Men87]] Nax 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.*, pages 30–36. IEEE Computer Society Press, 1987.

[[MGB04]] Clare Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004.

[[Uus98]] Tarmo Uustalu. *Natural Deduction for Intuitionistic Least and Greatest Fixedpoint Logics, with an Application to Program Construction*. PhD thesis, Royal Institute of Technology, Kista, Sweden, 1998.

[[UV97]] Tarmo Uustalu and Varmo Vene. A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In Magne Haveraaen and Olaf Owe, editors, *Selected Papers of the 8th Nordic Workshop on Programming Theory (NWPT '96), Oslo, Norway, December 1996*, volume 248 of *Research Reports, Department of Informatics, University of Oslo*, pages 237–246, May 1997.

[[UV02]] Tarmo Uustalu and Varmo Vene. Least and greatest fixed points in intuitionistic natural deduction. *Theoretical Computer Science*, 272:315–339, 2002.