A Datastructure for Iterated Powers

Ralph Matthes

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

April 14, 2006

Abstract. Bushes are considered as the first example of a truly nested datatype, i. e., a family of datatypes indexed over all types where a constructor argument not only calls this family with a changing index but even with an index that involves the family itself. For the time being, no induction principles for these datatypes are known. However, the author has introduced with Abel and Uustalu (TCS 333(1-2), pp. 3-66, 2005) iteration schemes that guarantee to define only terminating functions on those datatypes.
The article uses a generalization of Bushes to n-fold self-application and shows how to define elements of these types that have a number of data entries that is obtained by iterated raising to the power of n. Moreover, the data entries are just all the n-branching trees up to a certain height. The real question is how to extract this list of trees from that complicated data structure and to prove this extraction correct. Here, we use the "refined conventional iteration" from the cited article for the extraction and describe a verification that has been formally verified inside Coq with its predicative notion of set.

1 Introduction

Assume, we had a type transformation Bsh_3 that yields for every type A an abstract datatype $Bsh_3 A$ with just two datatype constructors

$$\begin{aligned} bnil_3 &: Bsh_3 A \ , \\ bcons_3 &: A \to (Bsh_3)^3 A \to Bsh_3 A \ . \end{aligned}$$

As is usual, we write $F^n A := \underbrace{F(\ldots(F}_{n\times} A)\ldots)$ for F any type transformation.

With these constructors, Bsh_3 becomes a "nested datatype" in the sense of Bird and Meertens [7]: These are families of datatypes indexed over types (hence type transformations) where arguments of constructors of the family member with index A may refer to some other members of the family. A classical example are the powerlists that precisely represent perfect binary leaf trees: a powerlist over A is either an element of A or a powerlist over $A \times A$. In $bcons_3$, we even refer to the member with index $(Bsh_3)^2 A$ that itself refers to Bsh_3. We would like to call such families truly nested datatypes.

Definition 1. *A* truly nested datatype *is a nested datatype with a call to the family name within a type argument of an argument of one of the datatype constructors.*

The treatment to come is not confined to truly nested datatypes but shows that even those complicated nested datatypes can be treated in a framework that guarantees termination of all programs.

The paper [7] considers an example, namely the analogous bushes with exponent 2 instead of our 3. There is not yet a well-established tradition in truly nested datatypes, and Bird and Paterson [8] study a family $Host$ just for demonstration purposes although they discuss an extended de Bruijn representation of lambda terms via true nesting elsewhere [9]. They also appear naturally in Hinze's account of generalized *trie* data structures [13]. But these works do not consider termination guarantees for the programs that traverse these data structures.

Type theory is well aware of termination questions. However, truly nested datatypes are not accepted as inductive definitions in the theorem prover Coq [23] since they are considered to violate the condition of strict positivity [6, Section 14.1.2.1]. This is certainly accepted due to the fact that there are not yet type-theoretical formulations of the reasoning principles for the proposed means of structured programming for these datatypes [8, 12, 17].

At least, there are already systems that guarantee termination of all functions that follow some type discipline in their recursive calls: With Abel and Uustalu [2–4], the author has proposed a number of iteration principles that can be used with truly nested datatypes and guarantee termination (strong normalization of the respective extension of Girard's system F^ω of higher-order polymorphism [10]). These iteration schemes can all be simulated already within F^ω and hence are available through encodings in Coq – with the universe Set taken to be impredicative, as it used to be in versions prior to 8.0. But impredicativity is still an option for the Coq system, and so the essential parts of the journal article [4] were implemented and verified in Coq in form of a student project with the author [22]. Here, we aim at something different. Although there is still no direct support for reasoning on truly nested datatypes in Coq, we may construct some of their more interesting elements and reason about what they contain.

Let us construct elements of $Bsh_3\,Tri$ with Tri the datatype of ternary unlabelled finite trees, hence just with the constructors

$$\begin{aligned} L &: Tri \ , \\ N &: Tri \to Tri \to Tri \to Tri \ . \end{aligned}$$

The function $mkTriBsh_3 : nat \to Bsh_3\,Tri$ shall take a natural number m and yield a bush that contains all the elements of Tri of height less than m. For this, we need a more general function $mkTriBsh'_3$ that takes a natural number m and a function argument f of type $Tri \to A$ for some type A and yields an element of $Bsh_3\,A$, in other words:

$$mkTriBsh'_3 : nat \to \forall A.\,(Tri \to A) \to Bsh_3\,A \ .$$

The expression $\forall A. (Tri \to A) \to Bsh_3\, A$ is again a type, i. e., a member of Coq's *Set*, only if *Set* is impredicative. Otherwise, it is a construction of the framework (in Coq, an element of *Type*). We define $mkTriBsh'_3$ as the fixpoint F of the following structural recursion on *nat* (this is even plain iteration):

$$\begin{aligned} F 0 f &:= bnil_3 \ , \\ F(Sm) f &:= bcons_3\, (f\, L)\, (Fm(\lambda t_1.Fm(\lambda t_2.Fm(\lambda t_3.f(N t_1 t_2 t_3))))) \ . \end{aligned}$$

Finally, set (implicitly taking $A := Tri$)

$$mkTriBsh_3\, m := mkTriBsh'_3\, m (\lambda x.x) \ .$$

Type-correctness can be seen as follows: If $t_1, t_2, t_3 : Tri$, then $f(Nt_1t_2t_3) : A$, hence $\lambda t_3.f(Nt_1t_2t_3) : Tri \to A$ which qualifies as functional argument for $mkTriBsh'_3\, m$. By induction hypothesis, we have $Fm(\lambda t_3.f(Nt_1t_2t_3)) : Bsh_3\, A$. The next step shows that in fact we have a polymorphic recursion here: The universally quantified type changes during the recursion. $\lambda t_2.Fm(\lambda t_3.f(Nt_1t_2t_3)) : Tri \to Bsh_3\, A$ has $Bsh_3\, A$ in place of A, hence the recursive call with Fm yields an element of $Bsh_3(Bsh_3\, A)$. Moving up the hierarchy once more, we arrive at

$$Fm(\lambda t_1.Fm(\lambda t_2.Fm(\lambda t_3.f(Nt_1t_2t_3)))) : (Bsh_3)^3 A \ ,$$

which is a good second argument to $bcons_3$.

We would now like to argue that $mkTriBsh_3\, 3$ contains precisely the *Tri*-elements of height less than 3, moreover that they do not occur several times. In essence, we would like to read off the bush the following list (drawn from the Coq development [18]):

```
L :: N L L L
    :: N L L (N L L L)
       :: N L (N L L L) L
          :: N L (N L L L) (N L L L)
             :: N (N L L L) L L
                :: N (N L L L) L (N L L L)
                   :: N (N L L L) (N L L L) L
                      :: N (N L L L) (N L L L) (N L L L) :: nil.
```

Therefore, there is the need for a function $toListBsh_3$ that transforms any element of $Bsh_3 A$ into a (finite) list over A. This cannot be done without the assumption that $Bsh_3 A$ has only elements that enter through $bnil_3$ and $bcons_3$. Unfortunately, we cannot say that $Bsh_3 A$ is the least set with some closure properties since we relate different instances of Bsh_3 – this is the problem with nested datatypes. The article [8] gives a semantics of nested datatypes within functor categories. Nested datatypes are then initial algebras for an endofunctor on a category of endofunctors. In order to have a better operational view and a termination guarantee through mere typing, Abel and the author introduced in [2] a system of iteration that depends not on an endofunctor on a functor category but just a rank-2 type transformer F with a *monotonicity witness* –

a term of a type that expresses monotonicity of F. In the journal version [4] this has been turned into the system $It^{\omega}_{=}$ (Section 6 of *loc. cit.*) of which we will show only the iterator with superscript $\kappa 1$ (we omit those superscripts altogether): Whenever F is a transformation of type transformations, then μF is a type transformation (the nested datatype associated with F). There is only one generic constructor in for the elements of μF for all those F's:

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

(If F happens to be a sum type, in represents a product of constructors, one for every summand.) In our case $Bsh_3 := \mu(BshF_3)$ with

$$BshF_3 := \lambda X \lambda A.\, 1 + A \times X(X(XA)) \ \ ,$$

and in particular, $in : (1 + A \times (Bsh_3)^3 A) \to Bsh_3 A$ whose type is isomorphic with the product of the types assigned to $bnil_3$ and $bcons_3$.

This is well-established tradition. The new contribution is the elimination rule that needs the following abbreviation for type transformations X, G (already present in [11]):

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

The more straightforward notion would have been

$$X \subseteq G := \forall A.\, XA \to GA \ \ .$$

It is important to base monotonicity of rank-2 type transformers on $\le$ instead of $\subseteq$, since otherwise truly nested datatypes will not be covered, see [4, Section 5.4].[1]

Monotonicity of a type transformation is thus expressed by $X \le X$. The notion of monotonicity for rank-2 type transformers F is defined analogously, but one level higher: The type construction $mon_2\,F$ (inhabiting *Type* in Coq) is defined to be

$$mon_2\,F := \forall X \forall G.\, X \le G \to FX \le FG \ \ .$$

System $It^{\omega}_{=}$ assumes a binary function symbol $It_{=}$ such that for every type transformation G, we have: If m has type $mon_2\,F$ and s has type $FG \subseteq G$, then $It_{=}(m,s)$ has type $\mu F \le G$. More pictorially:

$$\frac{\Gamma \vdash m : mon_2\,F \qquad \Gamma \vdash s : FG \subseteq G}{\Gamma \vdash It_{=}(m,s) : \mu F \le G}$$

There is no condition on F (except being transformation of type transformations) or G. The monotonicity witness m need not be closed, it may even be a variable

[1] One can pass from $X \subseteq G$ to $X \le G$ if X or G is monotone – certainly a standard assumption in any category-theoretic treatment when type constructors are interpreted as functors.

occurring in the typing context Γ. Certainly, the *step term* s corresponds to the morphism part of the F-algebra G – were this system based on category theory.

The computation rule is

$$It_=(m,s)f(in\,t) \longrightarrow s\ (m\,It_=(m,s)\,f\,t)\ .$$

Here $f : A \to B$ (from the definition of $\leq$) and $t : F(\mu F)A$, while X in the type of m is instantiated with μF. Hence, the terms on both sides of the rule have type GB. The major result of [2] has been that system F^ω, extended with these rules, is still strongly normalizing and hence yields total functions. This counts among the results on *type-based termination*: It is just the type of m that ensures that the recursive calls to $It_=(m,s)$ cannot go wrong: m dispatches the argument t (and the type-changing parameter f) to the iteratively defined function $It_=(m,s)$. (Note that applications are implicitly parenthesized to the left, hence there is no subterm $It_=(m,s)\,f\,t$ in the term to the right-hand side.)

The unavoidable disadvantage of this iteration scheme is that we cannot directly define a function $toListBsh_3 : Bsh_3 \subseteq list$. But we can define $toListBsh'_3 : Bsh_3 \leq list$ as $It_=(m,s)$ with the terms m and s, described as follows:

Assume $X \leq G$ and $A \to B$. We have to produce a term inhabiting

$$BshF_3XA \to BshF_3GB\ \equiv\ 1 + A \times X(X(XA)) \to 1 + B \times G(G(GB))\ .$$

Trivially, we get from 1 to 1 and from A to B. So, we concentrate on

$$X(X(XA)) \to G(G(GB))\ .$$

But this just requires three applications of the hypothesis $X \leq G$ to the hypothesis $A \to B$, thus illuminating the use of $\leq$ in place of $\subseteq$. Since this construction is the canonical one for $BshF_3$, let $bshf_3$ be a name for that monotonicity witness m.

For $s : BshF_3\,list \subseteq list$, the type nearly suggests the program: Assume a type A and an argument $t : 1 + A \times list(list(list\,A))$. We have to produce an element of $list\,A$. We do case analysis on the sum argument. In the left case, just return nil, in the right case, decompose the argument into $a : A$ and $b : list^3A$. The first element of the list we output is a, and then $flatten(flatten\,b)$, with the usual operation $flatten : list^2A \to listA$ that concatenates the argument lists.

With the computation rule, we get the following operational equations (we abbreviate $toListBsh'_3 := It_=(bshf_3, s)$ by toL):

$$\begin{aligned} toL\,f\,bnil_3 &= nil\ ,\\ toL\,f\,(bcons_3\,a\,b) &= f\,a :: flatten(flatten(toL(toL(toL\,f))b))\ . \end{aligned}$$

The function parameter f is replaced by $toListBsh'_3(toListBsh'_3 f)$ in the outermost recursive call. The essential difficulty with truly nested datatypes is that the recursive function is used as a whole (for unspecified bush arguments in our case since the second argument is missing!) in the changing parameter f.

Setting that parameter to the identity, we arrive at the desired function $toListBsh_3$. Using a fuller specification that already uniquely determines the result, we will verify in general that $toListBsh_3(mkTriBsh_3\, m)$ always yields the list of all ternary trees with height less than m. And we will generalize this from 3-bushes to n-bushes.

Outline of the paper

After this lengthy introduction that gently introduced many of the needed concepts – in particular the iteration principle that we use throughout to read off lists from our bushes – we informally prove the statement on $mkTriBsh_3$. In section 3, the bushes are generalized to n-fold nesting, and such n-bushes are constructed in detail for the n-ary trees of a given maximum height, thus with an iterated power of entries. In section 4, a mathematical description of the structure of the verification of these n-bushes is given. There is sufficient detail so that everybody could do the proof without having any further ideas. But there is also a full development in the theorem prover Coq, commented in Section 5. It discusses the fine points while the development itself is available on the author's homepage [18]. Future work and some further questions are addressed in Section 6.

Acknowledgements

The mentioned student project [22] brought up the questions that led to this paper. I am especially thankful to Dulma Rodriguez for her program that computes a powerlist of height n with elements $1, \ldots, 2^n$, the linear 2-bush and the 2-bushes with a number of entries that iterates squares. And many thanks to Jan Johannsen who suggested to me to study the relation with the n-branching unlabelled trees of height less than m.

2 Intuitive Verification

One can program the list of all ternary trees with height less than m by plain iteration on m without making reference to Bsh_3. Given $a : A$ and $s : A \to A$, we use the notation $[a; s]$ for the function $f : nat \to A$ that is defined by iteration as $f\,0 := a$ and $f\,(m+1) := s\,(f\,m)$. For example, $mkTriBsh'_3 = [a_B; s_B]$ with $a_B\, f := bnil_3$ and, for $v : \forall A.(Tri \to A) \to Bsh_3 A$ and $f : Tri \to A$,

$$s_B\, v\, f := bcons_3\,(f\, L)\,(v_{Bsh_3^2 A}(\lambda t_1.v_{Bsh_3 A}(\lambda t_2.v_A\,(\lambda t_3.f(N t_1 t_2 t_3))))) \ .$$

(The indices to v are here just for information about the instantiation of the parameter A.)

The question arises how one could find this term s_B. Unfortunately, this cannot be answered fully here, but a motivation can be given.

Define $\mathit{mkTriList} : \mathit{nat} \to \mathit{list}\,\mathit{Tri}$ as $\mathit{mkTriList} := [a_L; s_L]$ with $a_L := \mathit{nil}$ and for $l : \mathit{list}\,\mathit{Tri}$

$$s_L\,l := L :: \mathit{flatten}\Big(\mathit{flatten}\Big(\mathit{map}\Big(\lambda t_1.\mathit{map}(\lambda t_2.\mathit{map}(\lambda t_3.\, N t_1 t_2 t_3)l)l\Big)l\Big)\Big) \ .$$

Then $\mathit{mkTriList}\,0 = \mathit{nil}$, and $\mathit{mkTriList}(m+1) = s_L(\mathit{mkTriList}\,m)$.

The construction of s_L is explained as follows: If l contains all the ternary trees of height less than m, then all ternary trees of height less than $m+1$ are obtained by taking just a leaf L or by taking a node N with three trees from l. With t_1, t_2 from l fixed, $\mathit{map}(\lambda t_3.\, N t_1 t_2 t_3)l$ builds the list of all $N t_1 t_2 t_3$ with t_3 in l. The next step yields the list of these lists where t_2 runs through l, and, finally, the list of lists of lists when t_1 runs through l is obtained. Flattening twice and adding just L yields the result.

Hence, the problem is fully specified by $\mathit{mkTriList}$.

Theorem 1. *For all $m : \mathit{nat}$, $\mathit{toListBsh}_3(\mathit{mkTriBsh}_3\,m) = \mathit{mkTriList}\,m$.*

We cannot hope for a direct verification of this theorem by induction on m since $\mathit{mkTriBsh}'_3(m+1)(\lambda x.x)$ refers to instances of $\mathit{mkTriBsh}'_3\,m\,f$ with f not the identity.

Hence, the theorem has to be generalized for those arguments.

Lemma 1. *For all $m : \mathit{nat}, A, B, f : \mathit{Tri} \to A, g : A \to B$ holds*

$$\mathit{toListBsh}'_3\,g\,(\mathit{mkTriBsh}'_3\,m\,f) = \mathit{map}(g \circ f)(\mathit{mkTriList}\,m) \ .$$

The first idea here is to use the *fusion law* for natural numbers in order to express the right-hand side as a single iteration. To recall, fusion says that $f \circ [a; s] = [a'; s']$ if $a' = f\,a$ and $s' \circ f = f \circ s$ (to be proven by a simple induction over the natural numbers argument).

Let us define $\mathit{mkTriListMap} : \mathit{nat} \to \forall A.(\mathit{Tri} \to A) \to \mathit{list}\,A$ by

$$\mathit{mkTriListMap}\,m\,f := \mathit{map}\,f\,(\mathit{mkTriList}\,m) \ .$$

We will not be able to fix the parameter f in our intended application of the fusion law. Instead, we consider

$$\mathit{map}' : \mathit{list}\,\mathit{Tri} \to \forall A.\,(\mathit{Tri} \to A) \to \mathit{list}\,A \ ,$$

defined by $\mathit{map}'\,l\,f := \mathit{map}\,f\,l$. Thus, $\mathit{mkTriListMap} = \mathit{map}' \circ \mathit{mkTriList}$. Fusion will tell us that $\mathit{mkTriListMap} = [a_M; s_M]$ if $a_M = \mathit{map}'\,a_L$ and

$$s_M \circ \mathit{map}' = \mathit{map}' \circ s_L \ .$$

For the first equation, just set $a_M\,f := \mathit{map}'\,a_L\,f = \mathit{map}\,f\,\mathit{nil} = \mathit{nil}$. The second equation with arguments $l : \mathit{list}\,\mathit{Tri}, f : \mathit{Tri} \to A$ requires

$$s_M(\lambda f.\,\mathit{map}\,f\,l)f = \mathit{map}\,f\,(s_L\,l) \ .$$

We want to calculate the right-hand side of this equation. Here, we need categorical laws for map and $flatten$, namely the second functor law for map, i. e., that map commutes with composition, and naturality of $flatten$ as a transformation from $list^2$ to $list$: For all appropriate f, l,

$$flatten(map(map\, f)l) = map\, f(flatten\, l) \ .$$

With these, the right-hand side $map\, f\, (s_L\, l)$ above becomes

$$fL :: flatten\Big(flatten\Big(map\Big(\lambda t_1.map(\lambda t_2.map(\lambda t_3.\, f(Nt_1t_2t_3))l)l\Big)l\Big)\Big) \ .$$

Now, it is easy to find the definition of $s_M\, v\, f$ with $v : \forall A.\, (Tri \to A) \to list\, A$ that satisfies the equation:

$$s_M\, v\, f := fL :: flatten(flatten(v(\lambda t_1.v(\lambda t_2.v(\lambda t_3.\, f(Nt_1t_2t_3)))))) \ .$$

Note that these arguments will need a good deal of extensionality: If two functions are pointwise equal, they are equal. This principle does not hold in intensional type theory. But this does not mean that our reasoning becomes wrong. There is only much more work to do, namely to show that the contexts in which we want to replace one function by “another” that is extensionally equal, only depend on the extension, i. e., on the values of the function. Details are in the Coq development [18], further discussion is to be found in Section 5.

Proving Lemma 1 now means showing

$$toListBsh'_3\, g\, (mkTriBsh'_3\, m\, f) = [a_M; s_M]\, m(g \circ f) \ .$$

With this specialized argument $g \circ f$ to $[a_M; s_M]\, m$, the author sees no hope for further program transformation that might have suggested the recursive definition of $mkTriBsh'_3$. However, the verification is now a plain induction on m that, in intensional type theory, also needs that $[a_M; s_M]\, m\, f$ only depends on the extension of f.

3 Programming the n-Bushes

The number $\sharp_3 m$ of ternary trees of height strictly less than m is given by

$$\begin{array}{lcl} \sharp_3 0 & := & 0 \ , \\ \sharp_3(m+1) & := & (\sharp_3 m)^3 + 1 \ . \end{array}$$

We get the sequence $(\sharp_3 m)_m$ of iterated third powers (plus 1): 0, 1, 2, 9, 730, 389017001, ... Hence, we now know that we can define elements of $Bsh_3\, Tri$ with $\sharp_3 m$ entries by unfolding m steps of the recursive definition of $mkTriBsh'_3$. We could now sort of generalize this by defining elements of $Bsh_3 A$ with $\sharp_3 m$ elements, described as the values of a function $f : nat \to A$ at the arguments $0, 1, \ldots, \sharp_3 m - 1$. However, for space reasons, this will only be part of the implementation [18] and not pursued further in this article.

But we do generalize in a second direction: The exponent 3 is quite immaterial to the ideas and is replaced by an arbitrary n. Defining and reasoning with arbitrary n is then a challenge for metaprogramming.

Definition 2 (power of type transformation). *Let X be a type transformation. Then $X^0 := \lambda A.A$ and $X^{k+1} := \lambda A.X(X^k A)$.*

Note that we put the "new" X on the outside. For the mathematician, this is immaterial. For our intended verification with Coq, this seems to be an important decision, to be discussed after Definition 5.

Definition 3 (n-bushes). *The rank-2 type transformation is*

$$BshF_n := \lambda X \lambda A.\, 1 + A \times X^n A$$

and its fixed point is $Bsh_n := \mu(BshF_n)$ (using the μ of the introduction that does not require any property of its argument).

Definition 4 (datatype constructors). *$bnil_n := in(inl\langle\rangle)$ with inl the left injection into the sum and $bcons_n := \lambda a \lambda b.\, in(inr\langle a, b\rangle)$ with inr the right injection and pairing operation $\langle\cdot,\cdot\rangle$.*

The parameter n will be fixed throughout (we do not relate Bsh_n and Bsh_m).

This time, we start with the bush decomposition function. Since we cannot argue with dots and braces that are given numbers of repetitions, we have to define the building blocks by iteration on k and then use them with $k := n$.

Definition 5 (iterated flattening). *Define $flat_k : \forall A.\, list^k A \to list A$ by iteration on k as follows:*

$$\begin{aligned} flat_0\, a &:= a :: nil \;\;, \\ flat_{k+1}\, l &:= flatten(map\, flat_k\, l) \;\;. \end{aligned}$$

Remark: The much easier right-hand side $flatten(flat_k\, l)$ would require in the verification that $list(list^k A) = list^k(list A)$ is known to the type-checker which it is not due to our decision in Definition 2. With some effort due to extensionality problems (also see Section 5), it can be shown nevertheless for every l, that $flat_2\, l = flatten\, l$ and $flat_3\, l = flatten(flatten\, l)$.[2] We used this last term up to now, but $toListBsh'_3$ will now only formally be different but not extensionally.

Definition 6 (lifting $\leq$ to the k-th power). *For type transformations X, G and $i : X \leq G$ define $POW_k\, i : X^k \leq G^k$ by iteration on k:*

$$\begin{aligned} POW_0\, i &:= \lambda f.\, f \;\;, \\ POW_{k+1}\, i &:= \lambda f.\, i(POW_k\, i\, f) \;\;. \end{aligned}$$

Definition 7 (monotonicity witness). *Define $bshf_n : mon_2(BshF_n)$ in analogy with $bshf_3$ in the introduction: Assume $i : X \leq G$, $f : A \to B$ and $t : BshF_n X A$. Produce a term of type $BshF_n G\, B$. For this, do case analysis on t. If it is a left injection from 1, then inject this into the left. If it is the right injection of a pair of a term $a : A$ and a term $b : X^n A$, then inject the following pair into the right: It consists of $f a$ and $POW_n\, i\, f\, b$.*

[2] The general statement $flat_{k+1}\, l = flatten^k\, l$ does not type-check. Using heterogeneous equality alluded to in Section 5, one can work around this problem so that a single proof is obtained which can be instantiated to every concrete number k.

Definition 8 (listifying). *Define* $toListBsh'_n : Bsh_n \leq list$ *analogously to* $toListBsh'_3$ *in the introduction:* $toListBsh'_n := It_=(bshf_n, s)$ *with step term* $s : BshF_n\, list \subseteq list$ *just as before, but with* $flat_n\, b$ *instead of flattening* b *twice.*

We see that the earlier definition for $n = 3$ is only a special case modulo the remark after Definition 5. As usual, set $toListBsh_n := toListBsh'_n(\lambda x.x)$.

Lemma 2 (properties of $toListBsh'_n$**).** *The operational equations are*

$$toListBsh'_n\, f\, bnil_n = nil \quad and$$

$$toListBsh'_n\, f(bcons_n\, a\, b) = f\, a :: flat_n(POW_n\, toListBsh'_n\, f\, b) \ .$$

Proof. Unfold the definitions and use the operational rule for the iterator. □

The last three definitions only concerned the *analysis* of bushes. Now, we prepare for their definition.

Evidently, the number of n-ary unlabelled trees with height less than m is $\sharp_n m$, defined as follows:

Definition 9 (generalized $\sharp_n m$**).**

$$\begin{array}{lcl} \sharp_n 0 & := & 0 \ , \\ \sharp_n(m+1) & := & (\sharp_n m)^n + 1 \ . \end{array}$$

The datatype of the n-ary unlabelled trees will be called $Tree_n$, with datatype constructors $L_n : Tree_n$ and $N_n : vec_n \to Tree_n$. Here (recall that we fix n throughout), vec_k shall represent the k-tuples of $Tree_n$, with datatype constructors

$$\begin{array}{c} vnil : vec_0 \ , \\ vcons : Tree_n \to \forall k.\, vec_k \to vec_{k+1} \ . \end{array}$$

First, we generalize $mkTriList := [a_L; s_L] : nat \to list\, Tri$ to

$$mkTreeList_n := [a_{Ln}; s_{Ln}] : nat \to list\, Tree_n$$

with

$$\begin{array}{l} a_{Ln} := nil : list\, Tree_n \ , \\ s_{Ln} := \lambda l.\, L_n :: flat_n(\mathcal{L}\, l\, n\, N_n). : list\, Tree_n \to list\, Tree_n \ . \end{array}$$

Here, we use the functional

$$\mathcal{L} : list\, Tree_n \to \forall k.\, (vec_k \to Tree_n) \to list^k\, Tree_n \ ,$$

defined by plain iteration on k as

$$\begin{array}{l} \mathcal{L}\, l\, 0\, f := f\, vnil \ , \\ \mathcal{L}\, l\, (k+1)\, f := map\Big(\lambda t.\, \mathcal{L}\, l\, k\, (f \circ (vcons\, t\, k))\Big) l \ . \end{array}$$

Define for $t_1, t_2, t_3 : Tree_3$ the abbreviation

$$N'_3 t_1 t_2 t_3 := N'(vcons\, t_1\, 2\, (vcons\, t_2\, 1\, (vcons\, t_3\, 0\, vnil))) : Tree_3 \ .$$

Definition unfolding yields

$$s_{L3}\, l = L_3 :: flat_3\Big(map\Big(\lambda t_1.map(\lambda t_2.map(\lambda t_3.\, N_3' t_1 t_2 t_3)l)l\Big)l\Big) \enspace .$$

Consequently, up to the remark after Definition 5 and to the isomorphism between Tri and $Tree_3$, we get back our previous s_L.

Again, the function $mkTreeList_n$ fully specifies what we expect from our bush-making function $mkTreeBsh_n : nat \to Bsh_n\, Tree_n$:

$$\forall m.\, toListBsh_n(mkTreeBsh_n\, m) = mkTreeList_n\, m \enspace .$$

Certainly, one could hope for a proof that this specification really meets our intuition, in the sense that it implies certain properties (a looser specification) such as the correct number of elements and that the elements are of the required maximum height. We shall be satisfied if our definition of $mkTreeBsh_n$validates the specification above.

As usual, we define a more general function

$$mkTreeBsh_n' : nat \to \forall A.\, (Tree_n \to A) \to Bsh_n A$$

as $mkTreeBsh_n' := [a_{Bn}; s_{Bn}]$ and set

$$mkTreeBsh_n\, m := mkTreeBsh_n'\, m\, (\lambda x.x) \enspace .$$

For $v : \forall A.\, (Tree_n \to A) \to Bsh_n A$ and $f : Tree_n \to A$, define:

$$\begin{aligned} a_{Bn}\, f &:= bnil_3 \enspace , \\ s_{Bn}\, v\, f &:= bcons_3(f L_n)(\mathcal{B}\, v\, n\, (f \circ N_n)) \enspace , \end{aligned}$$

where the functional

$$\mathcal{B} : (\forall A.\, (Tree_n \to A) \to Bsh_n A) \to \forall k \forall A.\, (vec_k \to A) \to Bsh_n^k A$$

is defined by plain iteration on k:

$$\begin{aligned} \mathcal{B}\, v\, 0\, f &:= f\, vnil \enspace , \\ \mathcal{B}\, v\, (k+1)\, f &:= v_{Bsh_n^k A}(\lambda t.\, \mathcal{B}\, v\, k\, (f \circ (vcons\, t\, k))) \enspace . \end{aligned}$$

With these definitions, the case $n = 3$ is as follows:

$$s_{B3}\, v\, f = bcons_3\, (f\, L_n)\, (v_{Bsh_3^2 A}(\lambda t_1.v_{Bsh_3 A}(\lambda t_2.v_A\, (\lambda t_3.f(N_n' t_1 t_2 t_3))))) \enspace ,$$

which is s_B up to the isomorphism between $Tree_3$ and Tri.

As is usual for nested datatypes, a statement for $mkTreeBsh_n'$ instead of $mkTreeBsh_n$ has to be proved:

Theorem 2 (soundness for general n**).** *For all numbers m, types A, B and functions $f : Tree_n \to A$, $g : A \to B$,*

$$toListBsh_n'\, g\, (mkTreeBsh_n'\, m\, f) = map(g \circ f)(mkTreeList_n\, m) \enspace .$$

The proof of the theorem occupies the next section.

4 Verification

Remember that we fixed some n. Theorem 2 will be shown by induction on m. So let $T\,m$ stand for its statement with m fixed. Due to the repeated (n times) use of our function $mkTreeBsh'_n\,m$ in the recursive call for $m+1$ (replacing the formal parameter v of s_{Bn}, there will also be n intermediate steps of a proposition $P\,m\,k$, given in Definition 10. From $T\,m$, we will successively prove $P\,m\,0, \ldots, P\,m\,n$ and then be able to deduce $T(m+1)$, hence achieve the induction step of the theorem.

Definition 10 (Proposition). *Let $P\,m\,k$ be the following statement: For all types A, B, functions $f_0 : vec_k \to Tree_n$, $f : Tree_n \to A$, $g : A \to B$,*

$$flat_k(POW_k\,toListBsh'_n\,g\,(\mathcal{B}\,(mkTreeBsh'_n\,m)\,k\,(f \circ f_0))) = \\ map(g \circ f)(flat_k(\mathcal{L}\,(mkTriList_n\,m)\,k\,f_0)).$$

Lemma 3 (Theorem implies Proposition). *For all m, if $T\,m$ holds, then also for all k, $P\,m\,k$ holds.*

Proof. We fix m and assume $T\,m$ holds. Prove by induction on k that $P\,m\,k$ holds. The case $k = 0$ goes by unfolding of the definitions. For the inductive step we assume that $P\,m\,k$ holds and want to show $P\,m\,(k+1)$. This starts with simple unfolding of definitions. Then $T\,m$ is used with

$$\lambda t.\,\mathcal{B}\,(mkTreeBsh'_n\,m)\,k\,(f \circ f_0 \circ (vcons\,t\,k))$$

in place of f and $POW_k\,toListBsh'_n\,g$ in place of g. Then, the second functor law for *map* applies. Now, we may apply the induction hypothesis, namely that $P\,m\,k$ holds. Interestingly, this happens in the function argument of *map* (and the function f_0 for which the induction hypothesis is used will vary with that argument). In extensional mathematics, this is innocuous, but for the actual verification this requires further thought, see the next section. Once again, we have to use the second functor law for *map* and then naturality of $flatten$ (also used in Section 2; it can be proven by induction on lists and also needs that $map\,f$ is a homomorphism for list concatenation "++" which is in turn an easy induction). With this exception, we did not need any "free theorems" [24] in this verification. The free theorems say that every function of type $\forall A.\,X A \to G A$ with X, G functors, is natural in the sense of category theory. This is a very extensional view that would not be well supported by Coq.

Back to the proof: A further application of the second functor law for *map* suffices. Having the precise proof here does not seem necessary with this detailed description of the kind of reasoning that is needed. □

The proof of Theorem 2 is then by induction on m: For $m = 0$ just calculate. The induction step is nearly an immediate application of the previous lemma with $k := n$, as announced earlier.

5 Implementation

The complete development in Coq can be found on the author's webpage [18].

How does the presentation in this paper match with the implementation in Coq? Firstly, the current presentation lives in F^ω with type dependency on indices n and k added. If we regard n as a fixed constant, then this type dependency already disappears. Then, the induction over k can be reduced just to a step-by-step consideration of the cases $k := 0, \ldots, n$. Moreover, we adopted the raw syntax of the Curry-style typing. Hence, we might see ourselves precisely in the situation of the paper [4]. Hence, through the translations into F^ω given there, the programs can be found even in plain F^ω which is a subsystem of Coq's calculus of inductive constructions as long as *Set* is kept impredicative. Therefore, we know that these functions are all terminating despite their highly unconventional call structure. So good for the programs. Secondly, we never use in any way the universally quantified types in places where we speak about ordinary types. In other words, we do not use the full comprehension of impredicative *Set*. Thus, we may use (and have used) the new Coq 8.0 with predicative *Set* but cannot rely on the built-in equality any longer. The nested datatypes we used in this paper come just from a module parameter of a module type that has typed parameters for:

- the fixed point μ of rank-2 type constructors,
- the datatype constructor *in* and
- the iterator $It_=(\cdot,\cdot)$.

In addition, there is the hypothesis that $It_=(m,s)f(in\,t) = s\ (m\,It_=(m,s)\,f\,t)$, with $=$ not the built-in decidable convertibility relation but Leibniz equality that is animated by a rewriting mechanism in Coq. There is no implementation whatsoever. We just provide a name for a hypothetical implementation of this module type.

And there is no built-in termination guarantee. So, all the calculations have to be triggered by tactics that rewrite with equations. This will certainly have bad consequences if one wants to study types that depend on results computed from these iterators: The type-checker will not invoke the reductions, and rewritten arguments will no longer be feasible for these dependencies since equality can only be *expressed* between terms of types that are convertible (with respect to that built-in decidable convertibility relation). Fortunately, Coq also supports a heterogeneous equality that will not prove any equality between terms of types that are not provably equal, but where those equalities can at least be expressed. And as soon as they have been expressed, one may work on them by rewriting types with user-defined rewrite rules such as our computation rule for $It_=$. If this succeeds, one may finally also establish Leibniz equality. This heterogeneous equality has been introduced by McBride under the name "John Major Equality" [19] (see also [6, Section 8.2.7]), and there are even extensions of that idea that try to integrate extensional reasoning into intensional type theory [21].

Namely, rewriting has another defect: rewriting cannot be done under a λ-abstraction. Otherwise, the system would be extensional, and currently known extensional type theories have undecidable type-checking. Coq is definitely not based on extensional type theory, but many functionals do not distinguish between only extensionally equal argument functions. For example, $map\,f\,l$ depends only on the extension of f. This is to say that whenever for all x, $fx = gx$, then $map\,f\,l = map\,g\,l$ for all l. This principle is easily proven by induction on lists l and used in the proof of Lemma 3. Much harder would be to show a similar extensionality of $toListBsh'_n\,f\,b$ for the argument f. One would need an induction principle for the n-bushes b that does not seem to exist yet. In a sense, it is not surprising that we do not need such extensionality for the proof of Lemma 3: the expression on the right-hand side of proposition $P\,m\,k$ only depends on the extension of f and g (due to the property of map just mentioned). So, if $P\,m\,k$ is true, then there should be no concern with extensionality of this kind because we can always pass to the other side in course of the induction proof.

To sum up this last discussion: We cannot assume that extensionally equal functions are equal, but our proofs only would have liked to apply extensionality in situations where those functions were arguments to a functional that is indifferent to intensional differences, as long as the extensions coincide. I would like to call those functionals *extensional* as well.

It should also be mentioned that we used the vectors from the Coq standard library `Bvector` that has a family of vectors for any type parameter. This explains why $Tree_n$ can be defined in Coq. Other attempts did not work, in particular a definition of the type of N_n as an iterated implication or the type of the argument of N_n as an iterated product.

6 Future Work and Conclusions

Is there a fusion law that would help in establishing Lemma 1 or even Theorem 2? The crucial problem is the splitting into f and g.

Generic programming [5] aims at descriptions of algorithms that exist for every datatype which follows a certain grammar of datatype functors. For nested datatypes, there are the "hofunctors" [8, 17]. Generic Haskell [14, 16] goes further up the hierarchy and allows all finite kinds. Since it generates a Haskell program after having inspected which datatypes are really used, it could perhaps be extended to support the shallow metaprogramming we undertook in this paper. Then Generic Haskell would be a generic extension even of DependentML [25]. Can the function $toListBsh'_n$ be obtained by generic programming?

In general, container types are now quite well understood [15]. Here, we speak about higher containers with true nesting that could and should be seen as strictly positive.[3] The authors of [1] speculate that their framework of Martin-Löf categories should also be useful in this more general setting (they excluded nested datatypes in that article). Will this shed some light on programming with truly nested datatypes?

[3] Non-strict positivity is only concerned with appearences of the function space constructor in the type. Our framework does not exclude that at all, but we do not yet have interesting examples of nested datatypes which crucially contain $\to$.

It would be desirable to see a closer relationship with continuations since our examples worked on function arguments that first only produced elements of type A but climbed up the powers of $Bsh_n A$ in course of the recursion. One should be able to learn considerably from the approaches to program with these first-class higher-order functions.

What exactly is gained if one knows that a certain object is of type $Bsh_n A$ for some specific n and A? The powerlists mentioned in the introduction ensure that one has 2^i elements for some i. With n-bushes, this is by no means the case: One can construct easily elements of $Bsh_n\, nat$ that contain exactly the first i numbers, for any i (this is also contained in the Coq development [18]). What we have seen is that only a recursion of height m is needed to get $\sharp_n m$ elements and that one can precisely control what they are. Is this the maximum number? A theorem would require a precise computational model for its formulation.

For the moment, our contribution could perhaps more be understood as an adventure in types [20] - more precisely in truly nested container types.

And this adventure is only a beginning in the following sense: Although generic programs sometimes generate truly nested datatypes [13], they are seen as not yet proven useful. This might well be the case, but also the support by type-theoretic systems is still missing for them. Especially with these intricate iteration schemes, one would like to know – at least – that the programs terminate on finite input. Haskell does provide a programming environment, but no guarantees of that sort. In this article, we could reason about elements of truly nested datatypes since they were parameterized by datatypes as simple as the natural numbers. If the only purpose for generating such an n-bush were to flatten it immediately afterwards into a list, this would not justify our efforts. Once arrived in the realm of n-bushes, we would like to transform them – again with termination guarantees for those transformations. And we would like to reason about these transformations in a theorem prover that has decidable type-checking. The principles underlying such a reasoning have yet to be found. Then, truly nested datatypes like the n-bushes may start to prove their usefulness even if an adventure in types is not intended.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: Construction strictly positive types. Theoretical Computer Science **342** (2005) 3–27
2. 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
3. Abel, A., Matthes, R., Uustalu, T.: Generalized iteration and coiteration for higher-order nested datatypes. In: Gordon, A., ed., Foundations of Software Science and Computational Structures, 6th International Conference, volume 2620 of Lecture Notes in Computer Science. Springer Verlag (2003), 54–68
4. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333** (2005) 3–66

5. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming—an introduction. In: Swierstra, S. D., Henriques, P. R., Oliveira, J. N., eds., Advanced Functional Programming, volume 1608 of Lecture Notes in Computer Science (1999), 28–115
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer Verlag (2004)
7. 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
8. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11** (1999) 200–222
9. Bird, R. S., Paterson, R.: De Bruijn notation as a nested datatype. Journal of Functional Programming **9** (1999) 77–91
10. Girard, J.-Y.: Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Doctorat d'État, Université de Paris VII (1972)
11. Hinze, R.: Polytypic values possess polykinded types. Technical Report IAI-TR-99-15, Institut für Informatik III, Universität Bonn (1999)
12. Hinze, R.: Efficient generalized folds. In: Jeuring, J., ed., Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal (2000)
13. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10** (2000) 327–351
14. Hinze, R., Jeuring, J.: Generic Haskell: practice and theory. In: Backhouse, R., Gibbons, J., eds., Generic Programming, volume 2793 of Lecture Notes in Computer Science. Springer (2003), 1–56
15. Hoogendijk, P. F., de Moor, O.: Container types categorically. Journal of Functional Programming **10** (2000) 91–225
16. Löh, A.: Exploring Generic Haskell. Proefschrift (PhD thesis), Universiteit Utrecht, Institute for Programming Research and Algorithmics (2004). 331 pages.
17. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Aspects of Computing **16** (2004) 19–35
18. Matthes, R.: Coq development for "A Datastructure for Iterated Powers". http://www.irit.fr/~Ralph.Matthes/Coq/MPC06/ (2006)
19. McBride, C.: Elimination with a motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R., eds., Types for Proofs and Programs, Selected Papers, volume 2277 of Lecture Notes in Computer Science. Springer Verlag (2002), 197–216
20. Okasaki, C.: From fast exponentiation to square matrices: An adventure in types. In: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP '99), volume 34 of SIGPLAN Notices. ACM (1999), 28–35
21. Oury, N.: Extensionality in the calculus of constructions. In: Hurd, J., Melham, T. F., eds., Theorem Proving in Higher Order Logics. Proceedings, volume 3603 of Lecture Notes in Computer Science. Springer Verlag (2005), 278–293
22. Rodriguez, D.: Verification of (co)iteration schemes for nested datatypes in Coq. Student project at Dept. Comp. Sci., LMU Munich (2006). Available on http://www.tcs.ifi.lmu.de/~rodrigue/project.html
23. The Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.0. Project LogiCal, INRIA (2005). System available at coq.inria.fr.
24. 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
25. Xi, H.: Dependent Types in Practical Programming. Ph.D. thesis, Carnegie Mellon University (1998). 187 pages.