How Functorial Are (Deep) GADTs?

By: **Patricia Johann** and **Pierre Cagne**

Abstract

It is well-known that GADTs do not admit standard map functions of the kind supported by ADTs and nested types. In addition, standard map functions are insufficient to distribute their data-changing argument functions over all of the structure present in elements of deep GADTs, even just deep ADTs or nested types. This paper develops an algorithm for detecting exactly which functions are mappable over data whose types are (deep) GADTs. The algorithm takes as input a term t whose type is an instance of a deep GADT D and a function f to be mapped over t. It detects a minimal possible shape of t as an element of D, and returns a minimal set of constraints f must satisfy to be mappable over t. The crux of the algorithm is its ability to separate t's essential structure as an element of D -- i.e., the part of t that is essential for it to have the shape of an element of D -- from its incidental structure as an element of D -- i.e., the part of t that is simply data in the positions of this shape. The algorithm ensures that the constraints on f come only from t's essential structure. This work is part of an ongoing effort to define initial algebra semantics for GADTs that properly generalizes the usual semantics for ADTs and nested types as least fixpoints of higher-order endofunctors.

Johann P. & Cagne P. How Functorial Are (Deep) GADTs? 2022. Publisher version of record available at: https://arxiv.org/pdf/2203.14891.pdf

How Functorial Are (Deep) GADTs?

Patricia Johann Pierre Cagne
johannp@appstate.edu cagnep@appstate.edu
Appalachian State University

Abstract

It is well-known that GADTs do not admit standard map functions of the kind supported by ADTs and nested types. In addition, standard map functions are insufficient to distribute their data-changing argument functions over all of the structure present in elements of deep GADTs, even just deep ADTs or nested types. This paper develops an algorithm for detecting exactly which functions are mappable over data whose types are (deep) GADTs. The algorithm takes as input a term t whose type is an instance of a deep GADT D and a function f to be mapped over t. It detects a minimal possible shape of t as an element of D, and returns a minimal set of constraints f must satisfy to be mappable over t. The crux of the algorithm is its ability to separate t's *essential structure* as an element of D — i.e., the part of t that is essential for it to have the shape of an element of D — from its *incidental structure* as an element of D — i.e., the part of t that is simply data in the positions of this shape. The algorithm ensures that the constraints on f come only from t's essential structure. This work is part of an ongoing effort to define initial algebra semantics for GADTs that properly generalizes the usual semantics for ADTs and nested types as least fixpoints of higher-order endofunctors.

1 Introduction

Initial algebra semantics [6] is one of the cornerstones of the modern theory of data types. It has long been known to deliver practical programming tools — such as pattern matching, induction rules, and structured recursion operators — as well as principled reasoning techniques — like relational parametricity [23] — for algebraic data types (ADTs). Initial algebra semantics has also been developed for the syntactic generalization of ADTs known as nested types [7], and it has been shown to deliver analogous tools and techniques for them as well [16]. Generalized algebraic data types (GADTs) [22,24,25] generalize nested types — and thus further generalize ADTs — syntactically:

$$\boxed{\text{ADTs}} \xrightarrow[\text{generalized by}]{\text{syntactically}} \boxed{\text{nested types}} \xrightarrow[\text{generalized by}]{\text{syntactically}} \boxed{\text{GADTs}} \tag{1}$$

Given their ubiquity in modern functional programming, an important open question is whether or not an initial algebra semantics exists for GADTs.

The starting point for initial algebra semantics is to interpret types as objects in a suitably structured category $\mathcal{C}$, and to interpret open type expressions as endofunctors on this category. An ADT is interpreted as the least fixpoint of the endofunctor on $\mathcal{C}$ interpreting its underlying type expression. For example, the type expression underlying the standard data type [1]

$$\begin{aligned} &\mathsf{data\ List\ :\ Set \to Set\ where} \\ &\quad \mathsf{nil\ \ \ :\ \forall\, A \to List\ A} \\ &\quad \mathsf{cons\ :\ \forall\, A \to A \to List\ A \to List\ A} \end{aligned} \tag{2}$$

of lists of data of type A is $\mathsf{L_A\, X} = 1 + \mathsf{A} \times \mathsf{X}$. This is essentially the unfolding of the definition of a type X parameterized on A recognizing that an element of X can be constructed either from no data using the data constructor nil, or from one datum of type A and one already-constructed datum of type X using the data constructor cons. Replacing X by $\mathsf{List\, A}$ in (2) gives a recursive equation defining this type, so if A interprets A then the least fixpoint of the endofunctor $L_A X = 1 + A \times X$ on $\mathcal{C}$ interpreting $\mathsf{L_A}$ interprets $\mathsf{List\, A}$.

[1] Although our results apply to GADTs in any programming language, we will use Agda syntax for all code in this paper unless otherwise specified. But whereas Agda allows type parameters in the types of GADT data constructors to be implicit, we will always write all type parameters explicitly. We use sans serif font for code snippets and *italic* font for mathematics.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

Nested types generalize ADTs by allowing their constructors to take as arguments data whose types involve instances of the nested type other than the one being defined. The return type of each of its data constructors must still be precisely the instance being defined, though. This is illustrated by the following standard definitions of the nested types PTree of perfect trees and Bush of bushes:

$$\begin{array}{ll} \mathsf{data\ PTree\ :\ Set \to Set\ where} & \mathsf{data\ Bush\ :\ Set \to Set\ where} \\ \quad \mathsf{pleaf\ :\ \forall\, A \to A \to PTree\ A} & \quad \mathsf{bnil\ :\ \forall\, A \to Bush\ A} \\ \quad \mathsf{pnode\ :\ \forall\, A \to PTree\ (A \times A) \to PTree\ A} & \quad \mathsf{bcons\ :\ \forall\, A \to A \to Bush\ (Bush\ A) \to Bush\ A} \end{array}$$

A nested type N with at least one data constructor at least one of whose argument types involves an instance of N that itself involves an instance of N is called a *truly nested type*. The type of the data constructor bcons thus witnesses that Bush is a truly nested type. Because the recursive calls to a nested type's type constructor can be at instances of the type other than the one being defined, a nested type thus defines an entire family of types that must be constructed simultaneously. That is, a nested type defines an *inductive family of types*. By contrast, an ADT is usually understood as a family of inductive types, one for each choice of its type arguments. This is because every recursive call to an ADT's type constructor must be at the same instance as the one being defined.

Like ADTs, (truly) nested types can still be interpreted as least fixpoints of endofunctors. But because the recursive calls in a nested type's definition are not necessarily at the instance being defined, the endofunctor interpreting its underlying type expression must necessarily be a *higher-order* endofunctor on $\mathcal{C}$. For example, the endofunctor interpreting the type expression underlying PTree is $P\,F\,X = X + F\,(X \times X)$ and the endofunctor interpreting the type expression underlying Bush is $B\,F\,X = 1 + F\,(F\,X)$. The fact that fixpoints of higher-order endofunctors are themselves necessarily functors thus entails that nested types are interpreted as endofunctors on, rather than elements of, $\mathcal{C}$. This ensures that the fixpoint interpretation of a nested type has a functorial action and, moreover, that the map function for a nested type — such as is required to establish the nested type as an instance of Haskell's Functor class [2] — can be obtained as its syntactic reflection. For example, $\mathsf{map}_{\mathsf{PTree}}$ is the syntactic reflection of the functorial action of the fixpoint of P, and $\mathsf{map}_{\mathsf{Bush}}$ is the syntactic reflection of the functorial action of the fixpoint of B. Because nested types, including ADTs and truly nested types, are defined polymorphically, we can think of each element of such a type N as a "container" for data arranged at various *positions* in the underlying *shape* determined by the data constructors of N used to build it. Given a function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$, the function $\mathsf{map}_{\mathsf{N}}\,\mathsf{f}$ is then the expected shape-preserving-but-possibly-data-changing function that transforms an element of N with shape S containing data of type A into another element of N also of shape S but containing data of type B by applying f to each of its elements. The standard map functions for ADTs can be obtained in the very same way — i.e., by interpreting them as fixpoints of (now trivially) higher-order endofunctors, rather than of first-order endofunctors, on $\mathcal{C}$ and reflecting the functorial actions of those fixpoints back into syntax. For example, the usual map function $\mathsf{map}_{\mathsf{List}}$ for lists is nothing more than the syntactic reflection of the functorial action of the fixpoint of the higher-order endofunctor $L'\,F\,X = 1 + X \times F\,X$ underlying List.

Since GADTs syntactically subsume nested types, they would also require higher-order endofunctors for their interpretation. We might therefore expect GADTs to have *functorial initial algebra semantics*, and thus to support shape-preserving-but-possibly-data-changing map functions, just like nested types do. But because the shape of an element of a *proper* GADT — i.e., a GADT that is not a nested type (and thus is not an ADT) — is not independent of the data it contains, and is, in fact, *determined by* this data, not all GADTs do. For example, the GADT

$$\begin{array}{l} \mathsf{data\ Seq\ :\ Set \to Set\ where} \\ \quad \mathsf{const\ :\ \forall A \to A \to Seq\,A} \\ \quad \mathsf{pair\ :\ \forall A\,B \to Seq\,A \to Seq\,B \to Seq\,(A \times B)} \end{array}$$

of sequences does not support a standard structure-preserving-but-possibly-data-changing map function like ADTs and nested types do. If it did, then the clause of $\mathsf{map}_{\mathsf{Seq}}$ for an element of Seq of the form $\mathsf{pair\,x\,y}$ for $\mathsf{x} : \mathsf{A}$ and $\mathsf{y} : \mathsf{B}$ would be such that if $\mathsf{f} : (\mathsf{A} \times \mathsf{B}) \to \mathsf{C}$ then $\mathsf{map}_{\mathsf{Seq}}\,\mathsf{f}\,(\mathsf{pair\,x\,y}) = \mathsf{pair\,u\,v} : \mathsf{Seq\,C}$ for some appropriately typed u and v. But there is no way to achieve this unless C is of the form $\mathsf{A}' \times \mathsf{B}'$ for some A' and B', $\mathsf{u} : \mathsf{Seq\,A}'$ and $\mathsf{v} : \mathsf{Seq\,B}'$, and $\mathsf{f} = \mathsf{f}_1 \times \mathsf{f}_2$ for some $\mathsf{f}_1 : \mathsf{A} \to \mathsf{A}'$ and $\mathsf{f}_2 : \mathsf{B} \to \mathsf{B}'$. The non-uniformity in the type-indexing of proper GADTs — which is the very reason a GADT programmer is likely to use GADTs in the first place — thus turns out to be precisely what prevents them from supporting standard map functions.

Despite this, GADTs are currently known to support two different functorial initial algebra semantics, namely, the discrete semantics of [17] and the functorial completion semantics of [20]. The problem is that neither of these leads to a satisfactory uniform theory of type-indexed data types. On the one hand, the discrete semantics of [17] interprets GADTs as fixpoints of higher-order endofunctors on the *discretization* of the

[2] We write $\mathsf{map}_{\mathsf{D}}$ for the syntactic function $\mathsf{fmap} :: (\mathsf{A} \to \mathsf{B}) \to (\mathsf{D\,A} \to \mathsf{D\,B})$ witnessing that the type constructor D is an instance of Haskell's Functor class. Such functions are expected to satisfy syntactic reflections of the functor laws — i.e., preservation of identity functions and composition of functions — even though there is no compiler mechanism to enforce this.

category $\mathcal{C}$ interpreting types, rather than on $\mathcal{C}$ itself. In this semantics, the map function for every GADT is necessarily trivial. Viewing nested types as particular GADTs thus gives a functorial initial algebra semantics for them that does not coincide with the expected one. In other words, the discrete interpretation of [17] results in a semantic situation that does not reflect the syntactic one depicted in (1), and is thus inadequate. On the other hand, the functorial completion semantics of [20] interprets GADTs as endofunctors on $\mathcal{C}$ itself. Each GADT thus, like every nested type, has a non-trivial map function. This is, however, achieved at the cost of adding new "junk" elements, unreachable in syntax but interpreting elements in the "map closure" of its syntax, to the interpretation of every proper GADT. Functorial completion for Seq, e.g., adds interpretations of elements of the form map f (pair x y) even though these may not be of the form pair u v for any terms u and v. Importantly, functorial completion adds no junk to interpretations of nested types or ADTs, so unlike the semantics of [17], that of [20] does indeed properly extend the usual functorial initial algebra semantics for them. But since the interpretations of [20] are bigger than expected for proper GADTs, this semantics, too, is unacceptable. Although they are at the two extremes of the junk vs. functoriality spectrum, both known functorial initial algebra semantics for GADTs are fundamentally unsatisfactory.

In this paper we pursue a middle ground and ask: how much functoriality can we salvage for GADTs while still ensuring that their interpretations contain no junk? We already know that not every function on a proper GADT's type arguments will be mappable over it. But this paper answers this question more precisely by developing an algorithm for detecting exactly which functions are. Our algorithm takes as input a term t whose type is (an instance of) a GADT G and a function f to be mapped over t. It then detects the *minimal possible shape* of t as an element of G, and returns a *minimal set of constraints* f must satisfy in order to be mappable over t. The crux of the algorithm is its ability to separate t's *essential structure* as an element of G — i.e., the part of t that is essential for it to have the shape of an element of G — from its *incidental structure* as an element of G — i.e., the part of t that is simply data in the positions of this shape. The algorithm then ensures that the constraints ensuring that f is mappable come only from t's essential structure as an element of G.

The separation of a term into essential and incidental structure relative to a given specification is far from trivial, however. In particular, it is considerably more involved than simply inspecting the return types of G's constructors. As for ADTs and other nested types, a subterm built using one of G's data constructors can be an input term to another one (or to itself again), and this creates a kind of "feedback loop" in the well-typedness computation for the overall term. Moreover, if G is a proper GADT, then such a loop can force structure to be essential in the overall term even though it would be incidental in the subterm if the subterm were considered in isolation, and this can impose constraints on the functions mappable over it. This is illustrated in Examples 2.2 and 2.3 below, both of which involve a GADT G whose data constructor pairing can construct a term suitable as input to projpair.

Our algorithm is actually far more flexible than we have just described. Rather than simply considering t to be an element of the top-level GADT in its type, it can instead take as a third argument a specification, in the form of a perhaps deeper[3] data type D, one of whose instances it should be considered an element of. The algorithm will still return a minimal set of constraints f must satisfy in order to be mappable over t, but now these constraints are relative to the deep specification D rather than to the "shallow" specification G β. The feedback loops in and between the data types appearing in the specification D can, however, significantly complicate the separation of essential and incidental structure in terms. For example, if a term's specification is G (G β) then we will first need to compute which functions are mappable over its relevant subterms relative to G β before we can compute those mappable over the term itself relative to G (G β). Runs of our algorithm on deep specifications are given in Examples 2.5 and 4.5 below, as well as in our accompanying code [8].

This paper is organized as follows. Motivating examples highlighting the delicacies of the problem our algorithm solves are given in Section 2. Our algorithm is given in Section 3, and fully worked out sample runs of it are given in Section 4. Our conclusions, related work, and some directions for future work are discussed in Section 5. Our Agda implementation of our algorithm is available at [8], along with a collection of examples on which it has been run. This collection includes examples involving deep specifications and mutually recursively defined GADTs, as well as other examples that go beyond just the illustrative ones appearing in this paper.

2 The Problem and Its Solution: An Overview

In this section we use well-chosen example instances of the mapping problem for GADTs and deep data structures both to highlight its subtlety and to illustrate the key ideas underlying our algorithm that solves it. For each example considering a function f to be mapped over a term t relative to the essential structure specified by D we explain, intuitively, how to obtain the decomposition of t into the essential and incidental structure specified by D and what the minimal constraints are that ensure that f is mappable over t relative to it. By

[3] An ADT/nested type/GADT is *deep* if it is (possibly mutually inductively) defined in terms of other ADTs/nested types/GADTs (including, possibly, itself). For example, List (List ℕ) is a deep ADT, Bush (List (PTree A)) is a deep nested type, and Seq (PTree A), and List (Seq A) are deep GADTs.

design, we handle the examples only informally in this section. The results obtained by running our algorithm on their formal representations are given in Section 4.

Our algorithm will treat all GADTs in the class $\mathcal{G}$, whose elements have the following general form when written in Agda:

$$\begin{array}{l} \mathsf{data}\ \mathsf{G} : \mathsf{Set}^k \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{c}_1 : \mathsf{t}_1 \\ \quad \vdots \\ \quad \mathsf{c}_m : \mathsf{t}_m \end{array} \tag{3}$$

where k and m can be any natural numbers, including 0. Writing $\overline{v}$ for a tuple $(v_1, ..., v_l)$ when its length l is clear from context, and identifying a singleton tuple with its only element, each data constructor c_i, $i \in \{1, ..., m\}$, has type t_i of the form

$$\forall \overline{\alpha} \to F_1^{\mathsf{c}_i} \overline{\alpha} \to ... \to F_n^{\mathsf{c}_i} \overline{\alpha} \to \mathsf{G}\,(K_1^{\mathsf{c}_i} \overline{\alpha}, ..., K_k^{\mathsf{c}_i} \overline{\alpha}) \tag{4}$$

Here, for each $j \in \{1, ..., n\}$, $F_j^{\mathsf{c}_i} \overline{\alpha}$ is either a closed type, or is α_d for some $d \in \{1, ..., |\overline{\alpha}|\}$, or is $\mathsf{D}_j^{\mathsf{c}_i}\ (\overline{\phi_j^{\mathsf{c}_i} \overline{\alpha}})$ for some user-defined data type constructor $\mathsf{D}_j^{\mathsf{c}_i}$ and tuple $\overline{\phi_j^{\mathsf{c}_i} \overline{\alpha}}$ of type expressions at least one of which is not closed. The types $F_j^{\mathsf{c}_i} \overline{\alpha}$ must not involve any arrow types. However, each $\mathsf{D}_j^{\mathsf{c}_i}$ can be any GADT in $\mathcal{G}$, including G itself, and each of the type expressions in $\overline{\phi_j^{\mathsf{c}_i} \overline{\alpha}}$ can involve such GADTs as well. On the other hand, for each $\ell \in \{1, ..., k\}$, $K_\ell^{\mathsf{c}_i} \overline{\alpha}$ is a type expression whose free variables come from $\overline{\alpha}$, and that involves neither G itself nor any proper GADTs.[4] When $|\overline{\alpha}| = 0$ we suppress the initial quantification over types in (4). All of the GADTs appearing in this paper are in the class $\mathcal{G}$. All GADTs we are aware of from the literature whose constructors' argument types do not involve arrow types are also in $\mathcal{G}$. Our algorithm is easily extended to GADTs without this restriction provided all arrow types involved are strictly positive.

Our first example picks up the discussion for Seq on page 2. Because pair is the only restricted data constructor for Seq, so that the feedback dependencies for Seq are simple, it is entirely straightforward.

Example 2.1 The functions f mappable over

$$t = \mathsf{pair}\,(\mathsf{pair}\,(\mathsf{const}\,\mathsf{tt})\,(\mathsf{const}\,2))\,(\mathsf{const}\,5) : \mathsf{Seq}\,(\,(\mathsf{Bool} \times \mathsf{Int}) \times \mathsf{Int}) \tag{5}$$

relative to the specification $\mathsf{Seq}\,\beta$ are exactly those of the form $f = (f_1 \times f_2) \times f_3$ for some $f_1 : \mathsf{Bool} \to X_1$, $f_2 : \mathsf{Int} \to X_2$, and $f_1 : \mathsf{Int} \to X_3$, and some types X_1, X_2, and X_3. Intuitively, this follows from two analyses similar to that on page 2, one for each occurrence of pair in t. Writing the part of a term comprising its essential structure relative to the given specification in blue and the parts of the term comprising its incidental structure in black, our algorithm also deduces the following essential structure for t:

$$\mathsf{pair}\,(\mathsf{pair}\,(\mathsf{const}\,\mathsf{tt})\,(\mathsf{const}\,2))\,(\mathsf{const}\,5) : \mathsf{Seq}\,(\,(\mathsf{Bool} \times \mathsf{Int}) \times \mathsf{Int})$$

The next two examples are more involved because G has purposely been crafted so that its data constructor pairing can construct a term suitable as input to projpair.

Example 2.2 Consider the GADT

$$\begin{array}{l} \mathsf{data}\ \mathsf{G}\ :\ \mathsf{Set} \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{const}\ \ :\ \mathsf{G}\,\mathbb{N} \\ \quad \mathsf{flat}\ \ :\ \forall\,\mathsf{A} \to \mathsf{List}\,(\mathsf{G}\,\mathsf{A}) \to \mathsf{G}\,(\mathsf{List}\,\mathsf{A}) \\ \quad \mathsf{inj}\ \ :\ \forall\,\mathsf{A} \to \mathsf{A} \to \mathsf{G}\,\mathsf{A} \\ \quad \mathsf{pairing}\ :\ \forall\,\mathsf{A}\,\mathsf{B} \to \mathsf{G}\,\mathsf{A} \to \mathsf{G}\,\mathsf{B} \to \mathsf{G}\,(\mathsf{A} \times \mathsf{B}) \\ \quad \mathsf{projpair}\ :\ \forall\,\mathsf{A}\,\mathsf{B} \to \mathsf{G}\,(\mathsf{G}\,\mathsf{A} \times \mathsf{G}\,(\mathsf{B} \times \mathsf{B})) \to \mathsf{G}\,(\mathsf{A} \times \mathsf{B}) \end{array}$$

The functions mappable over

$$t = \mathsf{projpair}\,(\ \mathsf{inj}\ \ (\mathsf{inj}\,(\mathsf{cons}\,2\,\mathsf{nil}),\ \mathsf{pairing}\,(\mathsf{inj}\,2)\,\mathsf{const})\ ) : \mathsf{G}\,(\mathsf{List}\,\mathbb{N} \times \mathbb{N})$$

relative to the specification $\mathsf{G}\,\beta$ are exactly those of the form $f = f_1 \times id_{\mathbb{N}}$ for some type X and function $f_1 : \mathsf{List}\,\mathbb{N} \to X$. This makes sense intuitively: The call to projpair requires that a mappable function f must

[4] Formally, a GADT is a *proper* GADT if it has at least one *restricted data constructor*, i.e., at least one data constructor c_i with type as in (4) for which $K_\ell^{\mathsf{c}_i} \overline{\alpha} \neq \overline{\alpha}$ for at least one $\ell \in \{1, ..., k\}$.

at top level be a product $f_1 \times f_2$ for some f_1 and f_2, and the outermost call to inj imposes no constraints on $f_1 \times f_2$. In addition, the call to inj in the first component of the pair argument to the outermost call to inj imposes no constraints on f_1, and neither does the call to cons or its arguments. On the other hand, the call to pairing in the second component of the pair argument to the second call to inj must produce a term of type $\mathsf{G}\,(\mathbb{N} \times \mathbb{N})$, so the argument 2 to the rightmost call to inj and the call to const require that $f_2 = id_{\mathbb{N}}$. Our algorithm also deduces the following essential structure for t:

$$\mathsf{projpair}\ (\ \mathsf{inj}\ \ (\mathsf{inj}\ (\mathsf{cons}\,2\,\mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\,2)\ \mathsf{const})\ )\ :\ \mathsf{G}\,(\mathsf{List}\,\mathbb{N}\ \times\ \mathbb{N}) \tag{6}$$

Note that, although the argument to projpair decomposes into essential structure and incidental structure as inj (inj (cons 2 nil), pairing (inj 2) const) when considered as a standalone term relative to the specification $\mathsf{G}\,\beta$, the feedback loop between pairing and projpair ensures that t has the decomposition in (6) relative to $\mathsf{G}\,\beta$ when this argument is considered in the context of projpair. Similar comments apply throughout this paper.

Example 2.3 The functions f mappable over

$$t = \mathsf{projpair}\ (\ \mathsf{inj}\ \ (\mathsf{flat}\ (\mathsf{cons}\ \mathsf{const}\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\,2)\ \mathsf{const})\ )\ :\ \mathsf{G}\,(\mathsf{List}\,\mathbb{N}\ \times\ \mathbb{N})$$

relative to the specification $\mathsf{G}\,\beta$ for G as in Example 2.2 are exactly those of the form $f = map_{\mathsf{List}}\, id_{\mathbb{N}} \times id_{\mathbb{N}}$. This makes sense intuitively: The call to projpair requires that a mappable function f must at top level be a product $f_1 \times f_2$ for some f_1 and f_2, and the outermost call to inj imposes no constraints on $f_1 \times f_2$. In addition, the call to flat in the first component of the pair argument to inj requires that $f_1 = map_{\mathsf{List}}\, f_3$ for some f_3, and the call to cons in flat's argument imposes no constraints on f_3, but the call to const as cons's first argument requires that $f_3 = id_{\mathbb{N}}$. On the other hand, by the same analysis as in Example 2.2, the call to pairing in the second component of the pair argument to inj requires that $f_2 = id_{\mathbb{N}}$. Our algorithm also deduces the following essential structure for t:

$$\mathsf{projpair}\ (\ \mathsf{inj}\ \ (\mathsf{flat}\ (\mathsf{cons}\ \mathsf{const}\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\,2)\ \mathsf{const})\ )\ :\ \mathsf{G}\,(\mathsf{List}\,\mathbb{N}\ \times\ \mathbb{N})$$

The feedback loop between constructors in the GADT G in the previous two examples highlights the importance of the specification relative to which a term is considered. But this can already be seen for ADTs, which feature no such loops. This is illustrated in Examples 2.4 and 2.5 below.

Example 2.4 The functions f mappable over

$$t = \mathsf{cons}\,(\mathsf{cons}\,1\,(\mathsf{cons}\,2\,\mathsf{nil}))\,(\mathsf{cons}\,(\mathsf{cons}\,3\,\mathsf{nil})\,\mathsf{nil})\ :\ \mathsf{List}\,(\mathsf{List}\,\mathbb{N})$$

relative to the specification $\mathsf{List}\,\beta$ are exactly those of the form $f : \mathsf{List}\,\mathbb{N} \to X$ for some type X. This makes sense intuitively since any function from the element type of a list to another type is mappable over that list. The function need not satisfy any particular structural constraints. Our algorithm also deduces the following essential structure for t:

$$\mathsf{cons}\,(\mathsf{cons}\,1\,(\mathsf{cons}\,2\,\mathsf{nil}))\,(\mathsf{cons}\,(\mathsf{cons}\,3\,\mathsf{nil})\,\mathsf{nil})$$

Example 2.5 The functions f mappable over

$$t = \mathsf{cons}\,(\mathsf{cons}\,1\,(\mathsf{cons}\,2\,\mathsf{nil}))\,(\mathsf{cons}\,(\mathsf{cons}\,3\,\mathsf{nil})\,\mathsf{nil})\ :\ \mathsf{List}\,(\mathsf{List}\,\mathbb{N})$$

relative to the specification $\mathsf{List}\,(\mathsf{List}\,\beta)$ are exactly those of the form $f = \mathsf{map}_{\mathsf{List}}\, f'$ for some type X' and function $f' : \mathbb{N} \to X'$. This makes sense intuitively: The fact that any function from the element type of a list to another type is mappable over that list requires that $f : \mathsf{List}\,\mathbb{N} \to X$ for some type X as in Example 2.4. But if the internal list structure of t is also to be preserved when f is mapped over it, as indicated by the essential structure $\mathsf{List}\,(\mathsf{List}\,\beta)$, then X must itself be of the form $\mathsf{List}\,X'$ for some type X'. This, in turn, entails that $f = map_{\mathsf{List}} f'$ for some $f' : \mathbb{N} \to X'$. Our algorithm also deduces the following essential structure for t:

$$\mathsf{cons}\,(\mathsf{cons}\,1\,(\mathsf{cons}\,2\,\mathsf{nil}))\,(\mathsf{cons}\,(\mathsf{cons}\,3\,\mathsf{nil})\,\mathsf{nil})\ :\ \mathsf{List}\,(\mathsf{List}\,\mathbb{N})$$

The specification $\mathsf{List}\,(\mathsf{List}\,\beta)$ determining the essential structure in Example 2.5 is deep *by instantiation*, rather than *by definition*. That is, inner occurrence of List in this specification is not forced by the definition of the data type List that specifies its top-level structure. The quintessential example of a data type that is deep by definition is the ADT

$$\begin{aligned}
&\mathsf{data\ Rose}\ :\ \mathsf{Set} \to \mathsf{Set\ where}\\
&\quad \mathsf{rnil}\ \ \ :\ \forall\,\mathsf{A} \to \mathsf{Rose\ A}\\
&\quad \mathsf{rnode}\ :\ \forall\,\mathsf{A} \to \mathsf{A} \to \mathsf{List}\,(\mathsf{Rose\ A}) \to \mathsf{Rose\ A}
\end{aligned}$$

of rose trees, whose data constructor rnode takes as input an element of Rose at an instance of another ADT. Reasoning analogous to that in the examples above suggests that no structural constraints should be required to map appropriately typed functions over terms whose specifications are given by nested types that are deep

by definition. We will see in Example 4.4 that, although the runs of our algorithm are not trivial on such input terms, this is indeed the case.

With more tedious algorithmic bookkeeping, results similar to those of the above examples can be obtained for data types — e.g., $\mathsf{Bush}\,(\mathsf{List}\,(\mathsf{PTree}\,\mathsf{A}))$, $\mathsf{Seq}\,(\mathsf{PTree}\,\mathsf{A})$, and $\mathsf{List}\,(\mathsf{Seq}\,\mathsf{A})$ — that are deep by instantiation [8].

3 The Algorithm

In this section we give our algorithm for detecting mappable functions. The algorithm adm takes as input a data structure t, a tuple of functions to be mapped over t, and a specification — i.e., a (deep) data type — Φ. It detects the minimal possible shape of t relative to Φ and returns a minimal set C of constraints $\overline{f}$ must satisfy in order to be mappable over t viewed as an element of an instance of Φ. A call

$$\mathsf{adm}\ \ t\ \ \overline{f}\ \ \Phi$$

is to be made only if there exists a tuple $(\Sigma_1\overline{\beta},...,\Sigma_k\overline{\beta})$ of type expressions such that

- $\Phi = \mathsf{G}\,(\Sigma_1\overline{\beta},...,\Sigma_k\overline{\beta})$ for some data type constructor $\mathsf{G} \in \mathcal{G} \cup \{\times,+\}$ and some type expressions $\Sigma_\ell\overline{\beta}$, for $\ell \in \{1,...,k\}$

and

- if $\Phi = \times(\Sigma_1\overline{\beta},\Sigma_2\overline{\beta})$, then $t = (t_1,t_2)$, and $k = 2$, $\overline{f} = (f_1,f_2)$
- if $\Phi = +(\Sigma_1\overline{\beta},\Sigma_2\overline{\beta})$ and $t = \mathsf{inl}\,t_1$, then $k = 2$, $\overline{f} = (f_1,f_2)$
- if $\Phi = +(\Sigma_1\overline{\beta},\Sigma_2\overline{\beta})$ and $t = \mathsf{inr}\,t_2$, then $k = 2$, $\overline{f} = (f_1,f_2)$
- if $\Phi = \mathsf{G}\,(\Sigma_1\overline{\beta},...,\Sigma_k\overline{\beta})$ for some $\mathsf{G} \in \mathcal{G}$ then
  1) $t = \mathsf{c}\,t_1...t_n$ for some appropriately typed terms $t_1,...,t_n$ and some data constructor c for G with type of the form in (4),
  2) $t : \mathsf{G}\,(K_1^{\mathsf{c}}\overline{w},...,K_k^{\mathsf{c}}\overline{w})$ for some tuple $\overline{w} = (w_1,...,w_{|\overline{\alpha}|})$ of type expressions, and $\mathsf{G}\,(K_1^{\mathsf{c}}\overline{w},...,K_k^{\mathsf{c}}\overline{w})$ is exactly $\mathsf{G}\,(\Sigma_1\overline{s},...,\Sigma_k\overline{s})$ for some tuple $\overline{s} = (s_1,...,s_{|\overline{\beta}|})$ of types, and
  3) for each $\ell \in \{1,...,k\}$, f_ℓ has domain $K_\ell^{\mathsf{c}}\overline{w}$

These invariants are clearly preserved for each recursive call to adm.

As an optimization, the free variables in the type expressions $\Sigma_\ell\overline{\beta}$ for $\ell \in \{1,...,k\}$ can be taken merely to be *among* the variables in $\overline{\beta}$, since the calls $\mathsf{adm}\ t\ \ \overline{f}\ \ \mathsf{G}\,(\Sigma_1\overline{\beta},...,\Sigma_k\overline{\beta})$ and $\mathsf{adm}\ t\ \ \overline{f}\ \ \mathsf{G}\,(\Sigma_1\overline{\beta^+},...,\Sigma_k\overline{\beta^+})$ return the same set C (up to renaming) whenever $\overline{\beta}$ is a subtuple of the tuple $\overline{\beta^+}$. We therefore always take $\overline{\beta}$ to have minimal length below.

The algorithm is given as follows by enumerating each of its legal calls. Each call begins by initializing a set C of constraints to $\emptyset$.

A. $\mathsf{adm}\ (t_1,t_2)\ \ (f_1,f_2)\ \ \times(\Sigma_1\overline{\beta},\Sigma_2\overline{\beta})$
   (i) Introduce a tuple $\overline{g} = g_1,...,g_{|\overline{\beta}|}$ of fresh function variables, and add the constraints $\langle\Sigma_1\overline{g},f_1\rangle$ and $\langle\Sigma_2\overline{g},f_2\rangle$ to C.
   (ii) For $j \in \{1,2\}$, if $\Sigma_j\overline{\beta} = \beta_i$ for some i then do nothing and go to the next j if there is one. Otherwise, $\Sigma_j\overline{\beta} = \mathsf{D}\,(\zeta_1\overline{\beta},...,\zeta_r\overline{\beta})$, where D is a data type constructor in $\mathcal{G} \cup \{\times,+\}$ of arity r, so make the recursive call $\mathsf{adm}\ t_j\ \ (\zeta_1\overline{g},...,\zeta_r\overline{g})\ \ \mathsf{D}\,(\zeta_1\overline{\beta},...,\zeta_r\overline{\beta})$ and add the resulting constraints to C.
   (iii) Return C.

B. $\mathsf{adm}\ (\mathsf{inl}\,t)\ \ (f_1,f_2)\ \ +(\Sigma_1\overline{\beta},\Sigma_2\overline{\beta})$
   (i) Introduce a tuple $\overline{g} = (g_1,...,g_{|\overline{\beta}|})$ of fresh function variables, and add the constraints $\langle\Sigma_1\overline{g},f_1\rangle$ and $\langle\Sigma_2\overline{g},f_2\rangle$ to C.
   (ii) If $\Sigma_1\overline{\beta} = \beta_i$ for some i then do nothing. Otherwise, $\Sigma_1\overline{\beta} = \mathsf{D}\,(\zeta_1\overline{\beta},...,\zeta_r\overline{\beta})$, where D is a data type constructor in $\mathcal{G} \cup \{\times,+\}$ of arity r, so make the recursive call $\mathsf{adm}\ t\ \ (\zeta_1\overline{g},...,\zeta_r\overline{g})\ \ \mathsf{D}\,(\zeta_1\overline{\beta},...,\zeta_r\overline{\beta})$ and add the resulting constraints to C.
   (iii) Return C.

C. $\mathsf{adm}\ (\mathsf{inr}\,t)\ \ (f_1,f_2)\ \ +(\Sigma_1\overline{\beta},\Sigma_2\overline{\beta})$
   (i) Introduce a tuple $\overline{g} = (g_1,...,g_{|\overline{\beta}|})$ of fresh function variables, and add the constraints $\langle\Sigma_1\overline{g},f_1\rangle$ and $\langle\Sigma_2\overline{g},f_2\rangle$ to C.
   (ii) If $\Sigma_2\overline{\beta} = \beta_i$ for some i then do nothing. Otherwise, $\Sigma_2\overline{\beta} = \mathsf{D}\,(\zeta_1\overline{\beta},...,\zeta_r\overline{\beta})$, where D is a data type constructor in $\mathcal{G} \cup \{\times,+\}$ of arity r, so make the recursive call $\mathsf{adm}\ t\ \ (\zeta_1\overline{g},...,\zeta_r\overline{g})\ \ \mathsf{D}\,(\zeta_1\overline{\beta},...,\zeta_r\overline{\beta})$

and add the resulting constraints to C.

(iii) Return C.

D. $\mathsf{adm}\ (\mathsf{c}\,t_1, ..., t_n)\ \ (f_1, ..., f_k)\ \ \mathsf{G}\,(\Sigma_1\overline{\beta}, ..., \Sigma_k\overline{\beta})$

(i) Introduce a tuple $\overline{g} = (g_1, ..., g_{|\overline{\beta}|})$ of fresh function variables and add the constraints $\langle \Sigma_\ell \overline{g}, f_\ell \rangle$ to C for each $\ell \in \{1, ..., k\}$.

(ii) If $\mathsf{c}\,t_1, ..., t_n : \mathsf{G}\,(K_1^{\mathsf{c}}\overline{w}, ..., K_k^{\mathsf{c}}\overline{w})$ for some tuple $\overline{w} = (w_1, ..., w_{|\overline{\alpha}|})$ of types, let $\overline{\gamma} = (\gamma_1, ..., \gamma_{|\overline{\alpha}|})$ be a tuple of fresh type variables and solve the system of matching problems

$$\begin{aligned} \Sigma_1\overline{\beta} &\equiv K_1^{\mathsf{c}}\overline{\gamma} \\ \Sigma_2\overline{\beta} &\equiv K_2^{\mathsf{c}}\overline{\gamma} \\ &\vdots \\ \Sigma_k\overline{\beta} &\equiv K_k^{\mathsf{c}}\overline{\gamma} \end{aligned}$$

to get a set of assignments, each of the form $\beta \equiv \psi\overline{\gamma}$ or $\sigma\overline{\beta} \equiv \gamma$ for some type expression ψ or σ. This yields a (possibly empty) tuple of assignments $\overline{\beta_i \equiv \psi_i\overline{\gamma}}$ for each $i \in \{1, ..., |\overline{\beta}|\}$, and a (possibly empty) tuple of assignments $\overline{\sigma_{i'}\overline{\beta} \equiv \gamma_{i'}}$ for each $i' \in \{1, \ldots, |\overline{\gamma}|\}$. Write $\beta_i \equiv \psi_{i,p}\overline{\gamma}$ for the p^{th} component of the former and $\sigma_{i',q}\overline{\beta} \equiv \gamma_{i'}$ for the q^{th} component of the latter. An assignment $\beta_i \equiv \gamma_{i'}$ can be seen as having form $\beta_i \equiv \psi\overline{\gamma}_{i'}$ or form $\sigma\overline{\beta}_i \equiv \gamma_{i'}$, but always choose the latter representation. (This is justified because adm would return an equivalent set of assignments — i.e., a set of assignments yielding the same requirements on $\overline{f}$ — were the former chosen. The latter is chosen because it may decrease the number of recursive calls to adm.)

(iii) For each $i' \in \{1, \ldots, |\overline{\gamma}|\}$, define $\tau_{i'}\overline{\beta}\overline{\gamma}$ to be either $\sigma_{i',1}\overline{\beta}$ if this exists, or $\gamma_{i'}$ otherwise.

(iv) Introduce a tuple $\overline{h} = (h_1, ..., h_{|\overline{\gamma}|})$ of fresh function variables for $i' \in \{1, ..., |\overline{\gamma}|\}$.

(v) For each $i \in \{1, \ldots, |\overline{\beta}|\}$ and each constraint $\beta_i \equiv \psi_{i,p}\overline{\gamma}$, add the constraint $\langle \psi_{i,p}\overline{h}, g_i \rangle$ to C.

(vi) For each $i' \in \{1, \ldots, |\overline{\gamma}|\}$ and each constraint $\sigma_{i',q}\overline{\beta} \equiv \gamma_{i'}$ with $q > 1$, add the constraint $\langle \sigma_{i',q}\overline{g}, \sigma_{i',1}\overline{g} \rangle$ to C.

(vii) For each $j \in \{1, \ldots, n\}$, let $R_j = F_j^{\mathsf{c}}(\tau_1\overline{\beta}\overline{\gamma}, ..., \tau_{|\overline{\gamma}|}\overline{\beta}\overline{\gamma})$.

- if R_j is a closed type, then do nothing and go to the next j if there is one.
- if $R_j = \beta_i$ for some i or $R_j = \gamma_{i'}$ for some i', then do nothing and go to the next j if there is one.
- otherwise $R_j = \mathsf{D}\,(\zeta_{j,1}\overline{\beta}\overline{\gamma}, ..., \zeta_{j,r}\overline{\beta}\overline{\gamma})$, where D is a type constructor in $\mathcal{G} \cup \{\times, +\}$ of arity r, so make the recursive call

$$\mathsf{adm}\ t_j\ \ (\zeta_{j,1}\overline{g}\overline{h}, ..., \zeta_{j,r}\overline{g}\overline{h})\ \ R_j$$

and add the resulting constraints to C.

(viii) Return C.

We note that the matching problems in Step (ii) in the last bullet point above do indeed lead to a set of assignments of the specified form. Indeed, since invariant 2) on page 6 ensures that $\mathsf{G}\,(K_1^{\mathsf{c}}\overline{w}, ..., K_k^{\mathsf{c}}\overline{w})$ is exactly $\mathsf{G}\,(\Sigma_1\overline{s}, ..., \Sigma_k\overline{s})$, each matching problem $\Sigma_\ell\overline{\beta} \equiv K_\ell\overline{\gamma}$ whose left- or right-hand side is not already just one of the βs or one of the γs must necessarily have left- and right-hand sides that are *top-unifiable* [11], i.e., have identical symbols at every position that is a non-variable position in both terms. These symbols can be simultaneously peeled away from the left- and right-hand sides to decompose each matching problem into a unifiable set of assignments of one of the two forms specified in Step (ii). We emphasize that the set of assignments is not itself unified in the course of running adm.

It is only once adm is run that the set of constraints it returns is to be solved. Each such constraint must be either of the form $\langle \Sigma_\ell\overline{g}, f_\ell \rangle$, of the form $\langle \psi_{i,p}\overline{h}, g_i \rangle$, or of the form $\langle \sigma_{i',q}\overline{g}, \sigma_{i',1}\overline{g} \rangle$. Each constraint of the first form must have top-unifiable left- and right-hand components by virtue of invariant 2) on page 6. It can therefore be decomposed in a manner similar to that described in the preceding paragraph to arrive at a unifiable set of constraints. Each constraint of the second form simply assigns a replacement expression $\psi_{i,p}\overline{h}$ to each newly introduced variable g_i. Each constraint of the third form must again have top-unifiable left- and right-hand components. Once again, invariant 2) on page 6 ensures that these constraints are decomposable into a unifiable set of constraints specifying replacement functions for the gs.

Performing first-order unification on the entire system of constraints resulting from the decompositions specified above, and choosing to replace more recently introduced gs and hs with ones introduced later whenever possible, yields a *solved system* comprising exactly one binding for each of the fs in terms of those later-occurring variables. These bindings actually determine the collection of functions mappable over the input term to adm relative to the specification Φ. It is not hard to see that our algorithm delivers the expected results for ADTs

and nested types (when Φ is the type itself), namely, that all appropriately typed functions are mappable over each elements of such types. (See Theorem 3.1 below.) For GADTs, however, there is no *existing* understanding of which functions should be mappable over their terms. We therefore regard the solved system's bindings for the fs as actually *defining* the class of functions mappable over a given term relative to a specification Φ.

Theorem 3.1 *Let* N *be a nested type of arity* k *in* $\mathcal{G}$*, let* $\overline{w} = (w_1, \ldots, w_k)$ *comprise instances of nested types in* $\mathcal{G}$*, let* $t : \mathsf{N}\,\overline{w}$ *where* $\mathsf{N}\,\overline{w}$ *contains* n *free type variables, let* $\overline{\beta} = (\beta_1, \ldots, \beta_n)$*, and let* $\mathsf{N}\,(\Sigma_1\overline{\beta}, \ldots, \Sigma_k\overline{\beta})$ *be in* $\mathcal{G}$*. The solved system resulting from the call* $\mathsf{adm}\ \ t\ \ (\Sigma_1\overline{f}, \ldots, \Sigma_k\overline{f})\ \ \mathsf{N}\,(\Sigma_1\overline{\beta}, \ldots, \Sigma_k\overline{\beta})$ *for* $\overline{f} = (f_1, \ldots, f_n)$ *has the form* $\bigcup_{i=1}^{n}\{\langle g_{i,1}, f_i\rangle, \langle g_{i,2}, g_{i,1}\rangle, \ldots, \langle g_{i,r_i-1}, g_{i,r_i}\rangle\}$*, where each* $r_i \in \mathbb{N}$ *and the* $g_{i,j}$ *are pairwise distinct function variables. It thus imposes no constraints on the functions mappable over terms of ADTs and nested types.*

Proof. The proof is by cases on the form of the given call to adm. The constraints added to $\mathcal{C}$ if this call is of the form A, B, or C are all of the form $\langle \Sigma_j\overline{g}, \Sigma_j\overline{f}\rangle$ for $j = 1, 2$, and the recursive calls made are all of the form $\mathsf{adm}\ t'\ \ (\zeta_1\overline{g}, ..., \zeta_r\overline{g})\ \ \mathsf{D}\,(\zeta_1\overline{\beta}, ..., \zeta_r\overline{\beta})$ for some t', some $(\zeta_1, ..., \zeta_r)$, and some nested type D. Now suppose the given call is of the form D. Then Step (i) adds the constraints $\langle \Sigma_i\overline{g}, \Sigma_i\overline{f}\rangle$ for $i = 1, \ldots, k$ to $\mathcal{C}$. In Step (ii), $|\overline{\alpha}| = k$, and $K_i^{\mathsf{c}}\overline{w} = w_i$ for $i = 1, \ldots, k$ for every data constructor c for every nested type, so that the matching problems to be solved are $\Sigma_i\overline{\beta} \equiv \gamma_i$ for $i = 1, \ldots, k$. In Step (iii) we therefore have $\tau_i\overline{\beta}\overline{\gamma} = \Sigma_i\overline{\beta}$ for $i = 1, \ldots, k$. No constraints involving the variables $\overline{h}$ introduced in Step (iv) are added to $\mathcal{C}$ in Step (v), and no constraints are added to $\mathcal{C}$ in Step (vi) since the γs are all fresh and therefore pairwise distinct. For each R_j that is of the form $\mathsf{D}\,(\zeta_{j,1}\overline{\beta}\overline{\gamma}, \ldots, \zeta_{j,r}\overline{\beta}\overline{\gamma})$, where D is a nested type, the recursive call added to $\mathcal{C}$ in Step (vii) is of the form $\mathsf{adm}\ \ t_j\ \ (\zeta_{j,1}\overline{g}\overline{h}, \ldots, \zeta_{j,r}\overline{g}\overline{h})\ \ \mathsf{D}\,(\zeta_{j,1}\overline{\beta}\overline{\gamma}, \ldots, \zeta_{j,r}\overline{\beta}\overline{\gamma})$, which is again of the same form as in the statement of the theorem. For R_js not of this form there are no recursive calls, so nothing is added to $\mathcal{C}$. Hence, by induction on the first argument to adm, all of the constraints added to $\mathcal{C}$ are of the form $\langle \Psi\overline{\phi}, \Psi\overline{\psi}\rangle$ for some type expression Ψ and some ϕs and ψs, where the ϕs and ψs are all pairwise distinct from one another.

Each constraint of the form $\langle \Psi\overline{\phi}, \Psi\overline{\psi}\rangle$ is top-unifiable and thus leads to a sequence of assignments of the form $\langle \phi_i, \psi_i\rangle$. Moreover, the fact that $\tau_i\overline{\beta}\overline{\gamma} = \Sigma_i\overline{\beta}$ in Step (iii) ensures that no hs appear in any $\zeta_{j,i}\overline{g}\overline{h}$, so the solved constraints introduced by each recursive call can have as their right-hand sides only gs introduced in the call from which they spawned. It is not hard to see that the entire solved system resulting from the original call must comprise the assignments $\langle g_{1,1}, f_1\rangle, ..., \langle g_{1,n}, f_n\rangle$ from the top-level call, as well as the assignments $\langle g_{j_i+1,1}, g_{j_i,1}\rangle, ..., \langle g_{j_i+1,n}, g_{j_i,n}\rangle$, for $j_i = 0, ..., m_i - 1$ and $i = 1, ..., n$, where m_i is determined by the subtree of recursive calls spawned by f_i. Re-grouping this "breadth-first" collection of assignments "depth-first" by the trace of each f_i for $i = 1, ..., n$, we get a solved system of the desired form. □

4 Examples

Example 4.1 For t as in Example 2.1, the call $\mathsf{adm}\ t\ \ f\ \ \mathsf{Seq}\,\beta_1$ results in the sequence of calls:

call 1	adm	t	f	$\mathsf{Seq}\,\beta_1$
call 2.1	adm	pair (const tt) (const 2)	h_1^1	$\mathsf{Seq}\,\gamma_1^1$
call 2.2	adm	const 5	h_2^1	$\mathsf{Seq}\,\gamma_2^1$
call 2.1.1	adm	const tt	$h_1^{2.1}$	$\mathsf{Seq}\,\gamma_1^{2.1}$
call 2.1.2	adm	const 2	$h_2^{2.1}$	$\mathsf{Seq}\,\gamma_2^{2.1}$

The steps of adm corresponding to these call are given in the table below, with the most important components of these steps listed explicitly:

step no.	***matching problems***	$\overline{\tau}$	$\overline{R}$	$\overline{\zeta}$	***constraints added to*** C
1	$\beta_1 \equiv \gamma_1^1 \times \gamma_2^1$	$\tau_1\beta_1\gamma_1^1\gamma_2^1 = \gamma_1^1$ $\tau_2\beta_1\gamma_1^1\gamma_2^1 = \gamma_2^1$	$R_1 = \mathsf{Seq}\,\gamma_1^1$ $R_2 = \mathsf{Seq}\,\gamma_2^1$	$\zeta_{1,1}\beta_1\gamma_1^1\gamma_2^1 = \gamma_1^1$ $\zeta_{2,1}\beta_1\gamma_1^1\gamma_2^1 = \gamma_2^1$	$\langle g_1^1, f\rangle$ $\langle h_1^1 \times h_2^1, g_1^1\rangle$
2.1	$\gamma_1^1 \equiv \gamma_1^{2.1} \times \gamma_2^{2.1}$	$\tau_1\gamma_1^1\gamma_1^{2.1}\gamma_2^{2.1} = \gamma_1^{2.1}$ $\tau_2\gamma_1^1\gamma_1^{2.1}\gamma_2^{2.1} = \gamma_2^{2.1}$	$R_1 = \mathsf{Seq}\,\gamma_1^{2.1}$ $R_2 = \mathsf{Seq}\,\gamma_2^{2.1}$	$\zeta_{1,1}\gamma_1^1\gamma_1^{2.1}\gamma_2^{2.1} = \gamma_1^{2.1}$ $\zeta_{2,1}\gamma_1^1\gamma_1^{2.1}\gamma_2^{2.1} = \gamma_2^{2.1}$	$\langle g_1^{2.1}, h_1^1\rangle$ $\langle h_1^{2.1} \times h_2^{2.1}, g_1^{2.1}\rangle$
2.2	$\gamma_1^2 \equiv \gamma_1^{2.2}$	$\tau_1\gamma_2^1\gamma_1^{2.2} = \gamma_2^1$	$R_1 = \gamma_2^1$		$\langle g_1^{2.2}, h_2^1\rangle$
2.1.1	$\gamma_1^{2.1} \equiv \gamma_1^{2.1.1}$	$\tau_1\gamma_1^{2.1}\gamma_1^{2.1.1} = \gamma_1^{2.1}$	$R_1 = \gamma_1^{2.1}$		$\langle g_1^{2.1.1}, h_1^{2.1}\rangle$
2.1.2	$\gamma_2^{2.1} \equiv \gamma_1^{2.1.2}$	$\tau_1\gamma_2^{2.1}\gamma_1^{2.1.2} = \gamma_2^{2.1}$	$R_1 = \gamma_2^{2.1}$		$\langle g_1^{2.1.2}, h_2^{2.1}\rangle$

Since the solution to the generated set of constraints imposes the requirement that $f = (g_1^{2.1.1} \times g_1^{1.2.1}) \times g_1^{2.2}$, we conclude that the most general functions mappable over t relative to the specification $\mathsf{Seq}\,\beta_1$ are those of the form $f = (f_1 \times f_2) \times f_3$ for some types X_1, X_2, and X_3 and functions $f_1 : \mathsf{Bool} \to X_1$, $f_2 : \mathsf{Int} \to X_2$, and $f_3 : \mathsf{Int} \to X_3$. This is precisely the result obtained informally in Example 2.1.

Example 4.2 For G and t as in Example 2.2 and $f : \mathsf{List}\,\mathbb{N} \times \mathbb{N} \to X$ the call $\mathsf{adm}\ t\ \ f\ \ \mathsf{G}\,\beta_1$ results in the sequence of calls:

call 1	adm	t	f	$\mathsf{G}\beta_1$
call 2	adm	t_2	$\mathsf{G}h_1^1 \times \mathsf{G}(h_2^1 \times h_2^1)$	$\mathsf{G}(\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1))$
call 3	adm	t_3	$(\mathsf{G}g_1^2, \mathsf{G}(g_2^2 \times g_2^2))$	$\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$
call 4.1	adm	inj (cons 2 nil)	g_1^3	$\mathsf{G}\gamma_1^2$
call 4.2	adm	pairing (inj 2) const	$g_2^3 \times g_2^3$	$\mathsf{G}(\gamma_2^2 \times \gamma_2^2)$
call 4.2.1	adm	inj 2	$g_1^{4.2}$	$\mathsf{G}\gamma_2^2$
call 4.2.2	adm	const	$g_1^{4.2}$	$\mathsf{G}\gamma_2^2$

where

$$\begin{aligned} t &= \mathsf{projpair}\ (\ \mathsf{inj}\ (\ \mathsf{inj}\ (\mathsf{cons}\ 2\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\ 2)\ \mathsf{const}\ )\ ) \\ t_2 &= \mathsf{inj}\ (\ \mathsf{inj}\ (\mathsf{cons}\ 2\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\ 2)\ \mathsf{const}\ ) \\ t_3 &= (\ \mathsf{inj}\ (\mathsf{cons}\ 2\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\ 2)\ \mathsf{const}\ ) \end{aligned}$$

The steps of adm corresponding to these call are given in Table 1, with the most important components of these steps listed explicitly. Since the solution to the generated set of constraints imposes the requirement that $f = g_1^{4.1} \times id_{\mathbb{N}}$, we conclude that the most general functions mappable over t relative to the specification $\mathsf{G}\,\beta_1$ are those of the form $f = f' \times id_{\mathbb{N}}$ for some type X and some function $f' : \mathsf{List}\,\mathbb{N} \to X$. This is precisely the result obtained intuitively in Example 2.2.

Example 4.3 For G and t as in Example 2.3 and $f : \mathsf{List}\,\mathbb{N} \times \mathbb{N} \to X$ we have

$$\begin{aligned} K^{\mathsf{const}} &= \mathbb{N} \\ K^{\mathsf{flat}}\,\alpha &= \mathsf{List}\,\alpha \\ K^{\mathsf{inj}}\,\alpha &= \alpha \\ K^{\mathsf{pairing}}\,\alpha_1\,\alpha_2 &= \alpha_1 \times \alpha_2 \\ K^{\mathsf{projpair}}\,\alpha_1\,\alpha_2 &= \alpha_1 \times \alpha_2 \end{aligned}$$

The call $\mathsf{adm}\ t\ \ f\ \ \mathsf{G}\,\beta_1$ results in the sequence of calls:

call 1	adm	t	f	$\mathsf{G}\beta_1$
call 2	adm	t_2	$\mathsf{G}h_1^1 \times \mathsf{G}(h_2^1 \times h_2^1)$	$\mathsf{G}(\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1))$
call 3	adm	t_3	$(\mathsf{G}g_1^2, \mathsf{G}(g_2^2 \times g_2^2))$	$\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$
call 4.1	adm	flat (cons const nil)	g_1^3	$\mathsf{G}\gamma_1^2$
call 4.2	adm	pairing (inj 2) const	$g_2^3 \times g_2^3$	$\mathsf{G}(\gamma_2^2 \times \gamma_2^2)$
call 4.1.1	adm	cons const nil	$\mathsf{G}h_1^{4.1}$	$\mathsf{List}\,(\mathsf{G}\gamma_1^{4.1})$
call 4.2.1	adm	inj 2	$g_1^{4.2}$	$\mathsf{G}\gamma_2^2$
call 4.2.2	adm	const	$g_1^{4.2}$	$\mathsf{G}\gamma_2^2$
call 4.1.1.1	adm	const	$g_1^{4.1.1}$	$\mathsf{G}\,\gamma_1^{4.1}$
call 4.1.1.2	adm	nil	$\mathsf{G}g_1^{4.1.1}$	$\mathsf{List}(\mathsf{G}\gamma_1^{4.1})$

where

$$\begin{aligned} t &= \mathsf{projpair}\ (\ \mathsf{inj}\ (\ \mathsf{flat}\ (\mathsf{cons}\ \mathsf{const}\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\ 2)\ \mathsf{const}\ )\ ) \\ t_2 &= \mathsf{inj}\ (\ \mathsf{flat}\ (\mathsf{cons}\ \mathsf{const}\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\ 2)\ \mathsf{const}\ ) \\ t_3 &= (\ \mathsf{flat}\ (\mathsf{cons}\ \mathsf{const}\ \mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\ 2)\ \mathsf{const}\ ) \end{aligned}$$

call no.	matching problems	$\overline{\tau}$	$\overline{R}$	$\overline{\zeta}$	constraints added to C
1	$\beta_1 \equiv \gamma_1^1 \times \gamma_1^1$	$\tau_1\beta_1\gamma_1^1\gamma_2^1 = \gamma_1^1$ $\tau_2\beta_1\gamma_1^1\gamma_2^1 = \gamma_2^1$	$R_1 = \mathsf{G}\,(\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1))$	$\zeta_{1,1}\beta_1\gamma_1^1\gamma_2^1 = \mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$\langle g_1^1, f\rangle$ $\langle h_1^1 \times h_2^1, g_1^1\rangle$
2	$\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1) \equiv \gamma_1^2$	$\tau_1\gamma_1^1\gamma_2^1\gamma_1^2 = \mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$R_1 = \mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$\zeta_{1,1}\gamma_1^1\gamma_2^1\gamma_1^2 = \mathsf{G}\gamma_1^1$ $\zeta_{1,2}\gamma_1^1\gamma_2^1\gamma_1^2 = \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$\langle \mathsf{G}g_1^2 \times \mathsf{G}(g_2^2 \times g_2^2), \mathsf{G}h_1^1 \times \mathsf{G}(h_2^1 \times h_2^1)\rangle$
3				$\zeta_1\gamma_1^2\gamma_2^2 = \gamma_1^2$ $\zeta_2\gamma_1^2\gamma_2^2 = \gamma_2^2 \times \gamma_2^2$	$\langle \mathsf{G}g_1^3, \mathsf{G}g_1^2\rangle$ $\langle \mathsf{G}(g_2^3 \times g_2^3), \mathsf{G}(g_2^2 \times g_2^2)\rangle$
4.1	$\gamma_1^2 \equiv \gamma_1^{4.1}$	$\tau_1\gamma_1^2\gamma_1^{4.1} = \gamma_1^2$	$R_1 = \gamma_1^2$		$\langle g_1^{4.1}, g_1^3\rangle$
4.2	$\gamma_2^2 \times \gamma_2^2 \equiv \gamma_1^{4.2} \times \gamma_2^{4.2}$	$\tau_1\gamma_2^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$ $\tau_2\gamma_2^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$	$R_1 = \mathsf{G}\gamma_2^2$ $R_2 = \mathsf{G}\gamma_2^2$	$\zeta_{1,1}\gamma_1^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$ $\zeta_{2,1}\gamma_1^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$	$\langle g_1^{4.2} \times g_1^{4.2}, g_2^3 \times g_2^3\rangle$
4.2.1	$\gamma_2^2 \equiv \gamma_1^{4.2.1}$	$\tau_1\gamma_2^2\gamma_1^{4.2.1} = \gamma_2^2$	$R_1 = \gamma_2^2$		$\langle g_1^{4.2.1}, g_1^{4.2}\rangle$
4.2.2	$\gamma_2^2 \equiv \mathbb{N}$		$R_1 = 1$		$\langle g_1^{4.2.2}, g_1^{4.2}\rangle$ $\langle id_{\mathbb{N}}, g_1^{4.2.2}\rangle$

Table 1: Calls for Example 4.2

call no.	matching problems	$\overline{\tau}$	$\overline{R}$	$\overline{\zeta}$	constraints added to C
1	$\beta_1 \equiv \gamma_1^1 \times \gamma_1^1$	$\tau_1\beta_1\gamma_1^1\gamma_2^1 = \gamma_1^1$ $\tau_2\beta_1\gamma_1^1\gamma_2^1 = \gamma_2^1$	$R_1 = \mathsf{G}\,(\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1))$	$\zeta_{1,1}\beta_1\gamma_1^1\gamma_2^1 = \mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$\langle g_1^1, f\rangle$ $\langle h_1^1 \times h_2^1, g_1^1\rangle$
2	$\mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1) \equiv \gamma_1^2$	$\tau_1\gamma_1^1\gamma_2^1\gamma_1^2 = \mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$R_1 = \mathsf{G}\gamma_1^1 \times \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$\zeta_{1,1}\gamma_1^1\gamma_2^1\gamma_1^2 = \mathsf{G}\gamma_1^1$ $\zeta_{1,2}\gamma_1^1\gamma_2^1\gamma_1^2 = \mathsf{G}(\gamma_2^1 \times \gamma_2^1)$	$\langle \mathsf{G}g_1^2 \times \mathsf{G}(g_2^2 \times g_2^2), \mathsf{G}h_1^1 \times \mathsf{G}(h_2^1 \times h_2^1)\rangle$
3				$\zeta_1\gamma_1^2\gamma_2^2 = \gamma_1^2$ $\zeta_2\gamma_1^2\gamma_2^2 = \gamma_2^2 \times \gamma_2^2$	$\langle \mathsf{G}g_1^3, \mathsf{G}g_1^2\rangle$ $\langle \mathsf{G}(g_2^3 \times g_2^3), \mathsf{G}(g_2^2 \times g_2^2)\rangle$
4.1	$\gamma_1^2 \equiv \mathsf{List}\,\gamma_1^{4.1}$	$\tau_1\gamma_1^2\gamma_1^{4.1} = \gamma_1^{4.1}$	$R_1 = \mathsf{List}\,(\mathsf{G}\gamma_1^{4.1})$	$\zeta_{1,1}\gamma_1^2\gamma_1^{4.1} = \mathsf{G}\gamma_1^{4.1}$	$\langle g_1^{4.1}, g_1^3\rangle$ $\langle \mathsf{List}\,h_1^{4.1}, g_1^{4.1}\rangle$
4.2	$\gamma_2^2 \times \gamma_2^2 \equiv \gamma_1^{4.2} \times \gamma_2^{4.2}$	$\tau_1\gamma_2^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$ $\tau_2\gamma_2^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$	$R_1 = \mathsf{G}\gamma_2^2$ $R_2 = \mathsf{G}\gamma_2^2$	$\zeta_{1,1}\gamma_1^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$ $\zeta_{2,1}\gamma_1^2\gamma_1^{4.2}\gamma_2^{4.2} = \gamma_2^2$	$\langle g_1^{4.2} \times g_1^{4.2}, g_2^3 \times g_2^3\rangle$
4.1.1	$\mathsf{G}\gamma_1^{4.1} \equiv \gamma_1^{4.1.1}$	$\tau_1\gamma_1^{4.1}\gamma_1^{4.1.1} = \mathsf{G}\gamma_1^{4.1}$	$R_1 = \mathsf{G}\gamma_1^{4.1}$ $R_2 = \mathsf{List}(\mathsf{G}\gamma_1^{4.1})$	$\zeta_{1,1}\gamma_1^{4.1}\gamma_1^{4.1.1} = \gamma_1^{4.1}$ $\zeta_{2,1}\gamma_1^{4.1}\gamma_1^{4.1.1} = \mathsf{G}\gamma_1^{4.1}$	$\langle \mathsf{G}g_1^{4.1.1}, \mathsf{G}h_1^{4.1}\rangle$
4.2.1	$\gamma_2^2 \equiv \gamma_1^{4.2.1}$	$\tau_1\gamma_2^2\gamma_1^{4.2.1} = \gamma_2^2$	$R_1 = \gamma_2^2$		$\langle g_1^{4.2.1}, g_1^{4.2}\rangle$
4.2.2	$\gamma_2^2 \equiv \mathbb{N}$		$R_1 = 1$		$\langle g_1^{4.2.2}, g_1^{4.2}\rangle$ $\langle id_{\mathbb{N}}, g_1^{4.2.2}\rangle$
4.1.1.1	$\gamma_1^{4.1} \equiv \mathbb{N}$		$R_1 = 1$		$\langle g_1^{4.1.1.1}, g_1^{4.1.1}\rangle$ $\langle id_{\mathbb{N}}, g_1^{4.1.1.1}\rangle$
4.1.1.2	$\mathsf{G}\gamma_1^{4.1} \equiv \gamma_1^{4.1.1.2}$	$\tau_1\gamma_1^{4.1}\gamma_1^{4.1.1.2} = \mathsf{G}\gamma_1^{4.1}$	$R_1 = 1$		$\langle \mathsf{G}g_1^{4.1.1.2}, \mathsf{G}g_1^{4.1.1}\rangle$

Table 2: Calls for Example 4.3

The steps of adm corresponding to these call are given in Table 2, with the most important components of these steps listed explicitly. Since the solution to the generated set of constraints imposes the requirement that $f = map_{\mathsf{List}}\, id_{\mathbb{N}} \times id_{\mathbb{N}}$, we conclude that the only function mappable over t relative to the specification $\mathsf{G}\,\beta_1$ is this f. This is precisely the result obtained informally in Example 2.3.

Example 4.4 For t as in Example 2.4 the call $\mathsf{adm}\ t\ \ f\ \ \mathsf{List}\,\beta_1$ results in the sequence of calls:

call 1	adm	t	f	$\mathsf{List}\,\beta_1$
call 2	adm	cons (cons 3 nil) nil)	g_1^1	$\mathsf{List}\,\beta_1$
call 2.1	adm	nil	g_1^2	$\mathsf{List}\,\beta_1$

The steps of adm corresponding to these call are given in the table below, with the most important components of these steps listed explicitly:

step no.	***matching problems***	$\overline{\tau}$	$\overline{R}$	$\overline{\zeta}$	***constraints added to*** C
1	$\beta_1 \equiv \gamma_1^1$	$\tau_1\beta_1\gamma_1^1 = \beta_1$	$R_1 = \beta_1$ $R_2 = \mathsf{List}\,\beta_1$	$\zeta_{2,1}\beta_1\gamma_1^1 = \beta_1$	$\langle g_1^1, f\rangle$
2	$\beta_1 \equiv \gamma_1^2$	$\tau_1\beta_1\gamma_1^2 = \beta_1$	$R_1 = \beta_1$ $R_2 = \mathsf{List}\,\beta_1$	$\zeta_{2,1}\beta_1\gamma_1^2 = \beta_1$	$\langle g_1^2, g_1^1\rangle$
2.1	$\beta_1 \equiv \gamma_1^{2.1}$	$\tau_1\beta_1\gamma_1^{2.1} = \beta_1$	$R_1 = 1$		$\langle g_1^{2.1}, g_1^2\rangle$

Since the solution to the generated set of constraints imposes the requirement that $f = g_1^{2.1}$, we conclude that any function $f : \mathsf{List}\,\mathbb{N} \to X$ (for some type X) is mappable over t relative to the specification $\mathsf{List}\,\beta_1$.

Example 4.5 For t as in Example 2.5 the call $\mathsf{adm}\ t\ \ f\ \ \mathsf{List}\,(\mathsf{List}\,\beta_1)$ results in the following sequence of calls:

call 1	adm	t	f	$\mathsf{List}\,\beta_1$
call 2.1	adm	cons 1 (cons 2 nil)	g_1^1	$\mathsf{List}\,\beta_1$
call 2.2	adm	cons (cons 3 nil) nil)	$\mathsf{List}\,g_1^1$	$\mathsf{List}\,(\mathsf{List}\,\beta_1)$
call 2.1.1	adm	cons 2 nil	$g_1^{2.1}$	$\mathsf{List}\,\beta_1$
call 2.2.1	adm	cons 3 nil	$g_1^{2.2}$	$\mathsf{List}\,\beta_1$
call 2.2.2	adm	nil	$\mathsf{List}\,g_1^{2.2}$	$\mathsf{List}\,(\mathsf{List}\,\beta_1)$
call 2.1.1.1	adm	nil	$g_1^{2.1.1}$	$\mathsf{List}\,\beta_1$
call 2.2.1.1	adm	nil	$g_1^{2.2.1}$	$\mathsf{List}\,\beta_1$

The steps of adm corresponding to these calls are given in the table below, with the most important components of these steps listed explicitly:

step no.	***matching problems***	$\overline{\tau}$	$\overline{R}$	$\overline{\zeta}$	***constraints added to*** C
1	$\mathsf{List}\,\beta_1 \equiv \gamma_1^1$	$\tau_1\beta_1\gamma_1^1 = \mathsf{List}\,\beta_1$	$R_1 = \mathsf{List}\,\beta_1$ $R_2 = \mathsf{List}\,(\mathsf{List}\,\beta_1)$	$\zeta_{1,1}\beta_1\gamma_1^1 = \beta_1$ $\zeta_{2,1}\beta_1\gamma_1^1 = \mathsf{List}\,\beta_1$	$\langle \mathsf{List}\,g_1^1, f\rangle$
2.1	$\beta_1 \equiv \gamma_1^{2.1}$	$\tau_1\beta_1\gamma_1^{2.1} = \beta_1$	$R_1 = \beta_1$ $R_2 = \mathsf{List}\,\beta_1$	$\zeta_{2,2}\beta_1\gamma_1^{2.1} = \beta_1$	$\langle g_1^{2.1}, g_1^1\rangle$
2.2	$\mathsf{List}\,\beta_1 \equiv \gamma_1^{2.2}$	$\tau_1\beta_1\gamma_1^{2.2} = \mathsf{List}\,\beta_1$	$R_1 = \mathsf{List}\,\beta_1$ $R_2 = \mathsf{List}\,(\mathsf{List}\,\beta_1)$	$\zeta_{1,1}\beta_1\gamma_1^{2.2} = \beta_1$ $\zeta_{2,1}\beta_1\gamma_1^{2.2} = \mathsf{List}\,\beta_1$	$\langle \mathsf{List}\,g_1^{2.2}, \mathsf{List}\,g_1^1\rangle$
2.1.1	$\beta_1 \equiv \gamma_1^{2.1.1}$	$\tau_1\beta_1\gamma_1^{2.1.1} = \beta_1$	$R_1 = \beta_1$ $R_2 = \mathsf{List}\,\beta_1$	$\zeta_{2,2}\beta_1\gamma_1^{2.1.1} = \beta_1$	$\langle g_1^{2.1.1}, g_1^{2.1}\rangle$
2.2.1	$\beta_1 \equiv \gamma_1^{2.2.1}$	$\tau_1\beta_1\gamma_1^{2.2.1} = \beta_1$	$R_1 = \beta_1$ $R_2 = \mathsf{List}\,\beta_1$	$\zeta_{2,2}\beta_1\gamma_1^{2.2.1} = \beta_1$	$\langle g_1^{2.2.1}, g_1^{2.2}\rangle$
2.2.2	$\mathsf{List}\,\beta_1 \equiv \gamma_1^{2.2.2}$	$\tau_1\beta_1\gamma_1^{2.2.2} = \mathsf{List}\,\beta_1$	$R_1 = 1$		$\langle \mathsf{List}\,g_1^{2.2.2}, \mathsf{List}\,g_1^{2.2}\rangle$
2.1.1.1	$\beta_1 \equiv \gamma_1^{2.1.1.1}$	$\tau_1\beta_1\gamma_1^{2.1.1.1} = \beta_1$	$R_1 = 1$		$\langle g_1^{2.1.1.1}, g_1^{2.1.1}\rangle$
2.2.1.1	$\beta_1 \equiv \gamma_1^{2.2.1.1}$	$\tau_1\beta_1\gamma_1^{2.2.1.1} = \beta_1$	$R_1 = 1$		$\langle g_1^{2.2.1.1}, g_1^{2.2.1}\rangle$

Since the solution to the generated set of constraints imposes the requirement that $f = \mathsf{List}\, g_1^{2.2.1.1}$, we conclude that the most general functions mappable over t relative to the specification $\mathsf{List}\,(\mathsf{List}\,\beta_1)$ are those of the form $f = \mathsf{map}_{\mathsf{List}}\, f'$ for some type X and function $f' : \mathbb{N} \to X$.

5 Conclusion and Future Directions

The work reported here is part of a larger effort to develop a single, unified categorical theory of data types. In particular, it can be seen as a first step toward a properly functorial initial algebra semantics for GADTs that specializes to the standard functorial initial algebra semantics for nested types (which itself subsumes the standard such semantics for ADTs) whenever the GADTs in question is a nested type (or ADT).

Categorical semantics of GADTs have been studied in [15] and [17]. Importantly, both of these works interpret a GADT as a fixpoint of a higher-order endofunctor $[U\,, \mathsf{Set}] \to [U\,, \mathsf{Set}]$, where the category U is *discrete*. As discussed in Section 1, this destroys one of the main benefits of interpreting a data type D as a fixpoint μF_{D} of a higher-order endofunctor F_{D}, namely the existence of a non-trivial map function. Indeed, the action on morphisms of μF_{D} should interpret the map function $\mathsf{map}_{\mathsf{D}}$ standardly associated with D. But in the discrete settings of [15] and [17], the resulting endofunctor $\mu F_{\mathsf{D}} : U \to \mathsf{Set}$ has very little to say about the interpretation of $\mathsf{map}_{\mathsf{D}}$, since its functorial action need only specify the result of applying $\mathsf{map}_{\mathsf{D}}$ to a function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ when B is A and f is the identity function on A. In addition, [15] cannot handle truly nested data types such as Bush or the GADT G from Example 2.2. The resulting discrete initial algebra semantics for GADTs thus do not recover the usual functorial initial algebra semantics of nested types (including ADTs and truly nested types) when instantiated to these special classes of GADTs.

In [14] an attempt is made to salvage the method from [15] while taking the aforementioned issues into account. The overall idea is to relax the discreteness requirement on the category U, and to replace dependent products and sums in the development of [15] with left and right Kan extensions, respectively. But then the domain of μF_{D} must be the category of all interpretations of types and all morphisms between them, which in turn leads to the inclusion of unwanted junk elements obtained by map closure, as already described in Section 1 of [20]. So this solution also fails to bring us closer to a semantics of the kind we are aiming for.

Containers [1,2] provide an entirely different approach to describing the functorial action of an ADT or nested type. In this approach an element of such a type is described first by its structure, and then by the data that structure contains. That is, a ADT or nested type D is seen as comprising a set S_{D} of *shapes* and, for each shape $s \in S_{\mathsf{D}}$, a set $P_{\mathsf{D},s}$ of *positions* in s. If A is a type, then an element of $\mathsf{D}\,\mathsf{A}$ consists of a choice of a shape s and a labeling of each of position in s by elements of A. Thus, if A interprets A, then $\mathsf{D}\,\mathsf{A}$ is interpreted as a labeling $\sum_{s \in S_{\mathsf{D}}}(P_{\mathsf{D},s} \to A)$. The interpretation D for D simply abstracts this interpretation over D's input type, and, for any morphism $f : A \to B$, the functorial action $D\, f : \sum_{s \in S_{\mathsf{D}}}(P_{\mathsf{D},s} \to A) \to \sum_{s \in S_{\mathsf{D}}}(P_{\mathsf{D},s} \to B)$ is obtained by post-composition. This functorial action does indeed interpret $\mathsf{map}_{\mathsf{D}}$: given a shape and a labeling of its position by elements of A, we get automatically a data structure of the same shape whose positions are labeled by elements of B as soon as we have a function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ to translate elements of A to elements of B.

GADTs that go beyond ADTs and nested types have been studied from the container point of view as *indexed containers*, both in [4] and again in [15]. The authors of [4] propose encoding strictly positive indexed data types in terms of some syntactic combinators they consider "categorically inspired". However, as far as we understand their claim, map functions and their interpretations as a functorial actions are not worked out for indexed containers. The encoding in [4] nevertheless remains essential to understanding GADTs and other inductive families as "structures containing data". With respect to it, our algorithm can be understood as determining how "containery" a GADT D written in, say, Haskell or Agda is. Indeed, given a term t whose type is an instance of D, our algorithm can determine t's shape and positions, so there is no longer any need to guess or otherwise divine them. Significantly, there appears to be no general technique for determining the shapes and positions of the elements of a data type just from the type's programming language definition, and the ability to determine appropriate shapes and position sets usually comes only with a deep understanding of, and extensive experience with, the data structures at play.

We do not know of any other careful study of the functorial action of type-indexed strictly positive inductive families. The work reported here is the result of such a study for a specific class of such types, namely the GADTs described in Equations (3) and (4). Our algorithm defines map functions for GADTs that coincide with the usual ones for GADTs that are ADTs and nested types. The map functions computed by our algorithm will guide our ongoing efforts to give functorial initial algebra semantics for GADTs that subsume the usual ones for ADTs and nested types as fixpoints of higher-order endofunctors.

Acknowledgments This research was supported in part by NSF award CCR-1906388. It was performed while visiting Aarhus University's Logic and Semantics group, which provided additional support via Villum Investigator grant no. 25804, Center for Basic Research in Program Verification.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of Containers. Proceedings, Foundations of Software Science and Computation Structures, pp. 23-38, 2003.

[2] M. Abbott, T. Altenkirch, and N. Ghani. Containers - Constructing Strictly Positive Types. *Theoretical Computer Science* 342, pp. 3-27, 2005.

[3] The Agda Portal. At http://wiki.portal.chalmers.se/agda/pmwiki.php

[4] T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed Containers. *Journal of Functional Programming* 25 (e5), 2015.

[5] T. Altenkirch and P. Morris. Indexed Containers. Proceedings, Logic in Computer Science, pp. 277-285, 2009.

[6] R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[7] R. Bird and L. Meertens. Nested Datatypes. Proceedings, Mathematics of Program Construction, pp. 52–67, 1998.

[8] P. Cagne and P. Johann. At https://www.normalesup.org/~cagne/gadts/adm/README.html

[9] J. Cheney and R. Hinze. First-class Phantom Types. CUCIS TR2003-1901, Cornell University, 2003.

[10] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation* 76(2/3), pp. 95-120, 1988.

[11] D. J. Dougherty and P. Johann. An Improved General E-unification Method. *Journal of Symbolic Computation* 14, pp. 303-320, 1992.

[12] P. Dybjer. Inductive Families. *Formal Aspects of Computing* 6(4), pp. 440-465, 1994.

[13] The Haskell Homepage. At https://www.haskell.org

[14] M. Fiore. Discrete Generalised Polynomial Functors. Proceedings, Automata, Languages, and Programming, pp. 214-226, 2012.

[15] M. Hamana and M. Fiore. A Foundation for GADTs and Inductive Families: Dependent Polynomial Functor Approach. Proceedings, Workshop on Generic Programming, pp. 59-70, 2011.

[16] P. Johann and N. Ghani. Initial Algebra Semantics is Enough! Proceedings, Typed Lambda Calculus and Applications, pp. 207-222, 2007.

[17] P. Johann and N. Ghani. Foundations for Structured Programming with GADTs. Proceedings, Principles of Programming Languages, pp. 297-308, 2008.

[18] P. Johann and E. Ghiorzi. (Deep) Induction Rules for GADTs. Proceedings, Certified Programs and Proofs, pp. 324-337, 2022.

[19] P. Johann, E. Ghiorzi, and D. Jeffries. GADTs, Functoriality, Parametricity: Pick Two. Proceedings, Logical and Semantic Frameworks with Applications, 2021.

[20] P. Johann and A. Polonsky. Higher-Kinded Data Types: Syntax and Semantics. Proceedings, Logic in Computer Science, pp. 1-13, 2019.

[21] P. Johann and A. Polonsky. Deep Induction: Induction Rules for (Truly) Nested Types. Proceedings, Foundations of Software Science and Computation Structures, pp. 339-358, 2020.

[22] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple Unification-based Type Inference for GADTs. Proceedings, International Conference on Functional Programming, pp. 50-61, 2006.

[23] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. *Information Processing* 83(1), pp. 513-523, 1983.

[24] T. Sheard and E. Pasalic. Meta-programming with Built-in Type Equality. Proceedings, Workshop on Logical Frameworks and Meta-languages, pp. 106-124, 2004.

[25] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. Proceedings, Principles of Programming Languages, pp. 224–235, 2003.