Characterizing Functions Mappable Over GADTs

Patricia Johann and Pierre Cagne[0000−0003−3990−726X]

Appalachian State University, Boone NC 28608, USA
{johannp,cagnep}@appstate.edu

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 that characterizes those functions on a (deep) GADT's type arguments that are mappable over its elements. The algorithm takes as input a term t whose type is an instance of a (deep) GADT G, and returns a set of constraints a function must satisfy to be mappable over t. This algorithm, and thus this paper, can in some sense be read as *defining* what it means for a function to be mappable over t: f is mappable over an element t of G precisely when it satisfies the constraints returned when our algorithm is run on t and G. This is significant: to our knowledge, there is no existing definition or other characterization of the intuitive notion of mappability for functions over GADTs.

Keywords: GADTs · map functions · initial algebra semantics

1 Introduction

Initial algebra semantics [5] 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 [15] — for algebraic data types (ADTs). Initial algebra semantics has also been developed for the syntactic generalization of ADTs known as nested types [6], and it has been shown to deliver analogous tools and techniques for them as well [11]. Generalized algebraic data types (GADTs) [14,16,17] generalize nested types — and thus further generalize ADTs — syntactically:

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

Given their ubiquity in modern functional programming, an important open question is whether or not an initial algebra semantics can be defined for GADTs in such a way that a semantic analogue of (1) holds as well.

The standard initial algebra semantics of ADTs provides a functor $D : \mathit{Set} \to \mathit{Set}$ interpreting each such data type D, where Set is the category of sets and functions between them interpreting types [4]. The construction of D is sufficiently

uniform in its set argument to ensure not only that the data constructors of D are interpreted as natural transformations, but also that the standard map function[1]

$$\mathsf{map_D} : \forall \mathsf{A}\,\mathsf{B} \to (\mathsf{A} \to \mathsf{B}) \to (\mathsf{D}\,\mathsf{A} \to \mathsf{D}\,\mathsf{B})$$

for D is interpreted by D's functorial action. The naturality of the interpretations of D's constructors, which captures their polymorphic behavior, is reflected in syntax by the pattern-matching clauses defining $\mathsf{map_D}$ on data constructed using them.

As a concrete example, consider the standard data type

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

The data type List is interpreted as a functor $\mathit{List} : \mathit{Set} \to \mathit{Set}$ mapping each set A to the set of finite sequences of elements of A. The data constructor nil is interpreted as the natural transformation whose component at a set A is the function of type $1 \to \mathit{List}\,A$ mapping the single element of the singleton set 1 to the empty sequence, and the data constructor cons is interpreted as the natural transformation whose component at a set A is the function of type $A \times \mathit{List}\,A \to \mathit{List}\,A$ mapping the pair $(a, (a_1, \ldots, a_n))$ to $(a, a_1, \ldots, a_n)$. The functorial action of List on a function $f : A \to B$ is the function of type $\mathit{List}\,A \to \mathit{List}\,B$ taking a sequence $(a_1, \ldots, a_n)$ to the sequence $(f\,a_1, \ldots, f\,a_n)$. This functorial action indeed interprets the standard map function for lists, defined by pattern matching as follows:

$$\begin{aligned} &\mathsf{map_{List}\ :\ \forall A\,B \to (A \to B) \to (List\ A \to List\ B)} \\ &\mathsf{map_{List}\ f\ nil = nil} \\ &\mathsf{map_{List}\ f\ (cons\ a\ l) = cons\ (f\ a)\ (map_{List}\ f\ l)} \end{aligned}$$

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 definitions of the nested types PTree of perfect trees and Bush of bushes:

$$\begin{aligned} &\mathsf{data\ PTree\ :\ Set \to Set\ where} \\ &\quad \mathsf{pleaf\ \ :\ \forall\, A \to A \to PTree\ A} \\ &\quad \mathsf{pnode\ :\ \forall\, A \to PTree\ (A \times A) \to PTree\ A} \end{aligned}$$

$$\begin{aligned} &\mathsf{data\ Bush\ :\ Set \to Set\ where} \\ &\quad \mathsf{bnil\ \ \ :\ \forall\, A \to Bush\ A} \\ &\quad \mathsf{bcons\ :\ \forall\, A \to A \to Bush\ (Bush\ A) \to Bush\ A} \end{aligned}$$

[1] Although our results apply to GADTs in any programming language, we use Agda syntax for code in this paper. But while Agda allows type parameters to be implicit, we always write all type parameters explicitly. Throughout, we use sans serif font for code snippets and *italic* font for mathematics (specifically, for meta-variables).

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 very same instance as the one being defined.

The initial algebra semantics of nested types given in [11] provides a semantic analogue of the first inclusion in (1). Every nested type N has a map function $\mathsf{map_N} : \forall \mathsf{A}\,\mathsf{B} \to (\mathsf{A} \to \mathsf{B}) \to (\mathsf{N}\,\mathsf{A} \to \mathsf{N}\,\mathsf{B})$, and $\mathsf{map_N}$ coincides with the standard map function when N is an ADT. If we think of each element of a nested 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, then, given a function f of type $\mathsf{A} \to \mathsf{B}$, the function $\mathsf{map_N}\,f$ is 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 function $\mathsf{map_N}$ is interpreted by the functorial action of the functor $N : \mathit{Set} \to \mathit{Set}$ interpreting N whose existence is guaranteed by [11]. Like $\mathsf{map_N}$ itself, this functor specializes as expected when N is an ADT.

Since GADTs, like nested types, can also be regarded as containers in which data can be stored, we might expect every GADT G to support a shape-preserving-but-possibly-data-changing map function

$$\mathsf{map_G} : \forall \mathsf{A}\,\mathsf{B} \to (\mathsf{A} \to \mathsf{B}) \to (\mathsf{G}\,\mathsf{A} \to \mathsf{G}\,\mathsf{B}) \tag{3}$$

We might also expect to have initial algebra semantics interpreting G's constructors as natural transformations and $\mathsf{map_G}$ as the functorial action of the functor interpreting G. But this expection is perhaps too ambitious; see Section 5 for a discussion. In particular, a *proper* GADT — i.e., a GADT that is not a nested type (and thus is not an ADT) — need not support a map function as in (3). For example, the GADT[2]

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

of sequences does not. If it did, then the clause of $\mathsf{map_{Seq}}$ for an element of Seq of the form $\mathsf{pair}\,x\,y$ for $x : \mathsf{Seq}\,\mathsf{A}$ and $y : \mathsf{Seq}\,\mathsf{B}$ would be such that if $f : (\mathsf{A} \times \mathsf{B}) \to \mathsf{C}$

[2] The type of Seq is actually $\mathsf{Set} \to \mathsf{Set_1}$, but to aid readability we elide the explicit tracking of Agda universe levels in this paper. The data type Seq may be familiar to Haskell programmers as a fragment of the GADT Term introduced in [14] to represent terms in a simply-typed language. This fragment of Term is enough for the purposes of our discussion.

then $\mathsf{map}_{\mathsf{Seq}}\, f\, (\mathsf{pair}\, x\, y) = \mathsf{pair}\, u\, v : \mathsf{Seq}\, \mathsf{C}$ 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', $u : \mathsf{Seq}\, \mathsf{A}'$ and $v : \mathsf{Seq}\, \mathsf{B}'$, and $f = f_1 \times f_2$ for some $f_1 : \mathsf{A} \to \mathsf{A}'$ and $f_2 : \mathsf{B} \to \mathsf{B}'$. 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.

Although not every function on a proper GADT's type arguments is mappable over its elements, it is nevertheless reasonable to ask: which functions *can* be mapped over an element of a proper GADT in a shape-preserving-but-possibly-data-changing way? To answer this question we first rewrite the type of $\mathsf{map}_{\mathsf{G}}$ in (3) as $\forall \mathsf{A}\, \mathsf{B} \to \mathsf{G}\, \mathsf{A} \to (\mathsf{A} \to \mathsf{B}) \to \mathsf{G}\, \mathsf{B}$. This rewriting mirrors our observation above that those functions that are mappable over an element of a GADT can *depend* on that element. More precisely, it suggests that the type of the map function for G should actually be

$$\mathsf{map}_{\mathsf{G}} \;:\; \forall \mathsf{A}\, \mathsf{B} \to (\mathsf{e} : \mathsf{G}\, \mathsf{A}) \to (\mathsf{A} \to_{\mathsf{e}} \mathsf{B}) \to \mathsf{G}\, \mathsf{B} \tag{4}$$

where $\mathsf{A} \to_{\mathsf{e}} \mathsf{B}$ is a type, dependent on an element $\mathsf{e} : \mathsf{G}\, \mathsf{A}$, containing exactly those functions from A to B that can be successfully mapped over e.

In this paper we develop an algorithm characterizing those functions that should be in the type $\mathsf{A} \to_{\mathsf{e}} \mathsf{B}$. Our algorithm takes as input a term t whose type is (an instance of) a GADT G at type A, and returns a set of constraints a function must satisfy in order to be mappable over t. Our algorithm, and thus this paper, can in some sense be read as *defining* what it means for a function to be mappable over t. This is significant: to our knowledge, there is no existing definition or other characterization of the intuitive notion of mappability for functions over GADTs.

The crux of our 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 that must be met in order for f to be mappable come only from t's essential structure as an element of G. The separation of a term into essential and incidental structure 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). But if G is a proper GADT then such a dependency between constructor inputs and outputs 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 and 3 below, both of which involve a GADT G whose data constructor $\mathsf{pairing}$ can construct a term suitable as input to $\mathsf{projpair}$.

Our algorithm is actually far more flexible than just described. Rather than simply considering t to be an element of the top-level GADT in its type, the

algorithm instead takes as an additional argument a *specification* — i.e., a type expression over designated variables — one of whose instances t should be considered an element of. The specification D can either be a "shallow" data type of the form $\mathsf{G}\,\beta$ (with designated variable β) indicating that t is to be considered an element of a simple variable instance of G, or a deep[3] data type such as $\mathsf{G}\,(\mathsf{G}\,\beta)$ (with designated variable β) indicating that t should be considered an element of a more complex instance of G. The algorithm then returns a set of constraints a function must satisfy in order to be mappable over t relative to that given specification. We emphasize that the separation of essential and incidental structure in terms can become quite complicated when D is a deep data type. For example, if D is $\mathsf{G}\,(\mathsf{G}\,\beta)$ then those functions that are mappable over its relevant subterms relative to $\mathsf{G}\,\beta$ must be computed before those that are mappable over the term itself relative to $\mathsf{G}\,(\mathsf{G}\,\beta)$ can be computed. Runs of our algorithm on deep specifications are given in Examples 5 and 10 below, as well as in our accompanying artifact [7].

Although we use Agda syntax in this paper for convenience, the overarching setting for this work is not intended to be dependent type theory, but rather a language akin to a small pure fragment of Haskell. We deliberately remain language-agnostic, but the intended type system should be an impredicative extension of System F containing a fixpoint operator `data_where` as described in (5) and (6) below.

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. An Agda implementation of our algorithm is available at [7], along with a collection of examples on which it has been run. This 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 constraints are that ensure that f is mappable over t relative to it. Example 1 illustrates the fundamental challenge that arises when mapping functions over GADTs. Example 2 and Example 3 highlight the difference between a term's essential structure and its incidental structure. Example 4 and Example 5 show why the specification is

[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, $\mathsf{List}\,(\mathsf{List}\,\mathbb{N})$ is a deep ADT, $\mathsf{Bush}\,(\mathsf{List}\,(\mathsf{PTree}\,\mathsf{A}))$ is a deep nested type, and $\mathsf{Seq}\,(\mathsf{PTree}\,\mathsf{A})$, and $\mathsf{List}\,(\mathsf{Seq}\,\mathsf{A})$ are deep GADTs.

an important input to the algorithm. By design, we handle the examples only informally in this section to allow the reader to build intuition. 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:

$$\begin{array}{l}\mathsf{data}\ \mathsf{G} : \mathsf{Set}^k \to \mathsf{Set}\ \mathsf{where}\\ \quad \mathsf{c}_1 : \mathsf{t}_1\\ \quad \vdots\\ \quad \mathsf{c}_m : \mathsf{t}_m\end{array} \tag{5}$$

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

$$\forall \overline{\alpha} \to F_1^{\mathsf{c}_i}\overline{\alpha} \to ... \to F_{n_i}^{\mathsf{c}_i}\overline{\alpha} \to \mathsf{G}\,(K_1^{\mathsf{c}_i}\overline{\alpha}, ..., K_k^{\mathsf{c}_i}\overline{\alpha}) \tag{6}$$

Here, for each $j \in \{1, ..., n_i\}$, $F_j^{\mathsf{c}_i}\overline{\alpha}$ 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 (6). All of the GADTs appearing in this paper are in the class $\mathcal{G}$; this class is implemented in the accompanying code [7] as part of type-expr, with formation constraints as described immediately following the definition of _‖_⊢_. 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 3. Because the types of the inputs of const and pair are not deep, it is entirely straightforward.

Example 1. The functions f mappable over

$$t = \mathsf{pair}\,(\mathsf{pair}\,(\mathsf{const}\,\mathsf{tt})\,(\mathsf{const}\,2))\,(\mathsf{const}\,5) : \mathsf{Seq}\,((\mathsf{Bool} \times \mathsf{Int}) \times \mathsf{Int}) \tag{7}$$

relative to the specification $\mathsf{Seq}\,\beta$ are exactly those of the form $(f_1 \times f_2) \times f_3$ 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 3, one for each occurrence of pair in t. Writing the part of a term comprising its

[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 (6) for which $K_\ell^{\mathsf{c}_i}\overline{\alpha} \neq \overline{\alpha}$ for at least one $\ell \in \{1, ..., k\}$.

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: G has purposely been crafted so that its data constructor pairing can construct a term suitable as the second component of a pair whose image by inj can be input to projpair.

Example 2. Consider the GADT

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

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 $f_1 \times \mathsf{id}_{\mathbb{N}}$ 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 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 is $\mathsf{id}_{\mathbb{N}}$. Critically, it is the naturality of the constructor const that forces f_2 to be $\mathsf{id}_{\mathbb{N}}$ and not just any function of type $\mathbb{N} \to \mathbb{N}$ here. Our algorithm also deduces the following essential structure for t:

$$\mathsf{projpair}\ (\ \mathsf{inj}\ \ (\mathsf{inj}\ (\mathsf{cons}\,2\,\mathsf{nil}),\ \mathsf{pairing}\ (\mathsf{inj}\,2)\ \mathsf{const})\ ) : \mathsf{G}\,(\mathsf{List}\,\mathbb{N} \times \mathbb{N}) \qquad (8)$$

Note that, although the argument to projpair decomposes into essential structure and incidental structure as $\mathsf{inj}\,(\mathsf{inj}\,(\mathsf{cons}\,2\,\mathsf{nil}),\ \mathsf{pairing}\,(\mathsf{inj}\,2)\,\mathsf{const})$ when considered as a standalone term relative to the specification $\mathsf{G}\,\beta$, the fact that the output of pairing can be an input for projpair ensures that t has the decomposition in (8) relative to $\mathsf{G}\,\beta$ when this argument is considered in the context of projpair. Similar comments apply throughout this paper.

Example 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 are exactly those of the form $\mathsf{map}_{\mathsf{List}}\,\mathsf{id}_{\mathbb{N}} \times \mathsf{id}_{\mathbb{N}}$. 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 is $\mathsf{map}_{\mathsf{List}}\,f_3$ 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 is $\mathsf{id}_{\mathbb{N}}$. On the other hand, by the same analysis as in Example 2, the call to pairing in the second component of the pair argument to inj requires that f_2 is $\mathsf{id}_{\mathbb{N}}$. Our algorithm also deduces the following essential structure for t:

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

Again, the fact that the output of pairing can be a input for projpair 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 4 and 5 below.

Example 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 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 $\mathsf{map}_{\mathsf{List}}\,f'$ 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 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 is $\mathsf{map}_{\mathsf{List}}f'$ 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 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

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

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 9 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., $\mathsf{Bush\,(List\,(PTree\,A))}$, $\mathsf{Seq\,(PTree\,A)}$, and $\mathsf{List\,(Seq\,A)}$ — that are deep by instantiation [7].

3 The Algorithm

In this section we give our algorithm characterizing the functions that are mappable over GADTs. The algorithm adm takes as input a data structure t, a tuple of type expressions $\overline{f}$ representing functions to be mapped over t, and a specification Φ. Recall from the introduction that a *specification* is a type expression over designated variables in the ambient type calculus. It recursively traverses the term t recording the set C of constraints $\overline{f}$ must satisfy in order to be mappable over t viewed as an element of an instance of Φ. The elements of C are ordered pairs of the form $\langle \_, \_ \rangle$, whose components are compatible in the sense made precise in the paragraphs immediately following the algorithm. A call

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

is 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 (6),
  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 $K^{\mathsf{c}}_\ell \overline{w}$

These invariants will be preserved by each recursive call to adm below.

The free variables in the type expressions $\Sigma_\ell \overline{\beta}$ for $\ell \in \{1, ..., k\}$ can be taken 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 can therefore take $\overline{\beta}$ to have minimal length.

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. $\mathsf{adm}\ (t_1, t_2)\ \ (f_1, f_2)\ \ {\times}(\Sigma_1\overline{\beta}, \Sigma_2\overline{\beta})$
   1. Introduce a tuple $\overline{g} = g_1, ..., g_{|\overline{\beta}|}$ of fresh variables, and add the constraints $\langle \Sigma_1\overline{g}, f_1\rangle$ and $\langle \Sigma_2\overline{g}, f_2\rangle$ to C.
   2. 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.
   3. Return C.

B. $\mathsf{adm}\ (\mathsf{inl}\, t)\ \ (f_1, f_2)\ \ {+}(\Sigma_1\overline{\beta}, \Sigma_2\overline{\beta})$
   1. Introduce a tuple $\overline{g} = (g_1, ..., g_{|\overline{\beta}|})$ of fresh variables, and add the constraints $\langle \Sigma_1\overline{g}, f_1\rangle$ and $\langle \Sigma_2\overline{g}, f_2\rangle$ to C.
   2. 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.
   3. Return C.

C. $\mathsf{adm}\ (\mathsf{inr}\, t)\ \ (f_1, f_2)\ \ {+}(\Sigma_1\overline{\beta}, \Sigma_2\overline{\beta})$
   1. Introduce a tuple $\overline{g} = (g_1, ..., g_{|\overline{\beta}|})$ of fresh variables, and add the constraints $\langle \Sigma_1\overline{g}, f_1\rangle$ and $\langle \Sigma_2\overline{g}, f_2\rangle$ to C.
   2. 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.
   3. Return C.

D. $\mathsf{adm}\ (\mathsf{c}\, t_1, ..., t_n)\ \ (f_1, ..., f_k)\ \ \mathsf{G}\,(\Sigma_1\overline{\beta}, ..., \Sigma_k\overline{\beta})$
   1. Introduce a tuple $\overline{g} = (g_1, ..., g_{|\overline{\beta}|})$ of fresh variables and add the constraints $\langle \Sigma_\ell\overline{g}, f_\ell\rangle$ to C for each $\ell \in \{1, ..., k\}$.
   2. If $\mathsf{c}\, t_1, ..., t_n : \mathsf{G}\,(K^{\mathsf{c}}_1\overline{w}, ..., K^{\mathsf{c}}_k\overline{w})$ 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

$$\begin{aligned} \Sigma_1\overline{\beta} &\equiv K^{\mathsf{c}}_1\overline{\gamma} \\ \Sigma_2\overline{\beta} &\equiv K^{\mathsf{c}}_2\overline{\gamma} \\ &\vdots \\ \Sigma_k\overline{\beta} &\equiv K^{\mathsf{c}}_k\overline{\gamma} \end{aligned}$$

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.)

3. 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.
4. Introduce a tuple $\overline{h} = (h_1, ..., h_{|\overline{\gamma}|})$ of fresh variables for $i' \in \{1, ..., |\overline{\gamma}|\}$.
5. 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.
6. 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.
7. For each $j \in \{1, \ldots, n\}$, let $R_j = F^{\mathsf{c}}_j(\tau_1\overline{\beta}\overline{\gamma}, ..., \tau_{|\overline{\gamma}|}\overline{\beta}\overline{\gamma})$.
   - 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.
8. Return C.

We note that the matching problems in Step D.2 above do indeed lead to a set of assignments of the specified form. Indeed, since invariant 2) on page 9 ensures that $\mathsf{G}\,(K^{\mathsf{c}}_1\overline{w}, ..., K^{\mathsf{c}}_k\overline{w})$ 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* [8], 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 D.2. 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 9. 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 9 ensures that these constraints are decomposable into a unifiable set of constraints specifying replacements 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 1 below.) But since there is no already *existing* understanding of which functions should be mappable over the elements of GADTs, we actually regard the solved system's bindings for the fs as *defining* the class of functions mappable over a given element of a GADT relative to a specification Φ.

Theorem 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 variables. It thus imposes no constraints on the functions mappable over elements of 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 D.1 adds the constraints $\langle \Sigma_i\overline{g}, \Sigma_i\overline{f}\rangle$ for $i = 1, \ldots, k$ to $\mathcal{C}$. In Step D.2, $|\overline{\alpha}| = k$, 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 D.3 we therefore have $\tau_i\overline{\beta\gamma} = \Sigma_i\overline{\beta}$ for $i = 1, \ldots, k$. No constraints involving the variables $\overline{h}$ introduced in Step D.4 are added to $\mathcal{C}$ in Step D.5, and no constraints are added to $\mathcal{C}$ in Step D.6 since the γs are all fresh and therefore pairwise distinct. For each R_j that is of the form $\mathsf{D}\,(\zeta_{j,1}\overline{\beta\gamma}, \ldots, \zeta_{j,r}\overline{\beta\gamma})$, where D is a nested type, the recursive call added to $\mathcal{C}$ in Step D.7 is of the form $\mathsf{adm}\ \ t_j\ \ (\zeta_{j,1}\overline{g}\overline{h}, \ldots, \zeta_{j,r}\overline{g}\overline{h})\ \ \mathsf{D}\,(\zeta_{j,1}\overline{\beta\gamma}, \ldots, \zeta_{j,r}\overline{\beta\gamma})$, 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 D.3 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 6. For t as in Example 1, the call $\mathsf{adm}\ t\quad f\quad \mathsf{Seq}\,\beta_1$ 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 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	$\beta_1 \equiv \gamma_1^1 \times \gamma_2^1$	$\tau_1\beta_1\gamma_1^1\gamma_2^1 = \gamma_1^1$ $\tau_2\beta_1\gamma_1^1\gamma_2^1 = \gamma_2^1$	$R_1 = \mathsf{Seq}\,\gamma_1^1$ $R_2 = \mathsf{Seq}\,\gamma_2^1$	$\zeta_{1,1}\beta_1\gamma_1^1\gamma_2^1 = \gamma_1^1$ $\zeta_{2,1}\beta_1\gamma_1^1\gamma_2^1 = \gamma_2^1$	$\langle g_1^1, f\rangle$ $\langle h_1^1 \times h_2^1, g_1^1\rangle$
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 requires that f has the form $(g_1^{2.1.1} \times g_1^{1.2.1}) \times g_1^{2.2}$, we conclude that the most general functions mappable over t relative to the specification $\mathsf{Seq}\,\beta_1$ are those of the form $(f_1 \times f_2) \times f_3$ 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 1.

Example 7. For G and t as in Example 2 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 calls are given in Table 1, with the most important components of these steps listed explicitly. Since the solution to the generated set of constraints requires that f has the form $g_1^{4.1} \times \mathbb{N}$, we conclude that the most general functions mappable over t relative to the specification $\mathsf{G}\,\beta_1$ are those of the form $f' \times \mathsf{id}_{\mathbb{N}}$ for some type X and some function $f' : \mathsf{List}\,\mathbb{N} \to X$. This is precisely the result obtained intuitively in Example 2.

Example 8. For G and t as in Example 3 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	$\gamma_2^2 \equiv \mathbb{N}$		$R_1 = 1$		$\langle g_1^{4.2.2}, g_1^{4.2}\rangle$ $\langle \mathbb{N}, g_1^{4.2.2}\rangle$

Table 1. Calls for Example 7

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	$\gamma_2^2 \equiv \mathbb{N}$		$R_1 = 1$		$\langle g_1^{4.2.2}, g_1^{4.2}\rangle$ $\langle \mathbb{N}, g_1^{4.2.2}\rangle$
4.1.1.1	$\gamma_1^{4.1} \equiv \mathbb{N}$		$R_1 = 1$		$\langle g_1^{4.1.1.1}, g_1^{4.1.1}\rangle$ $\langle \mathbb{N}, g_1^{4.1.1.1}\rangle$
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 8

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 requires that f has the form $\mathsf{List}\,\mathbb{N} \times \mathbb{N}$, we conclude that the only function mappable over t relative to the specification $\mathsf{G}\,\beta_1$ is $\mathsf{id}_{\mathsf{List}\,\mathbb{N}} \times \mathsf{id}_{\mathbb{N}}$. This is precisely the result obtained informally in Example 3.

Example 9. For t as in Example 4 the call $\mathsf{adm}\ t\ \ f\ \ \mathsf{List}\,\beta_1$ results in the sequence of calls:

call 1	$\mathsf{adm}\ t$	f	$\mathsf{List}\,\beta_1$
call 2	$\mathsf{adm}\ \mathsf{cons}\,(\mathsf{cons}\,3\,\mathsf{nil})\ \mathsf{nil})$	g_1^1	$\mathsf{List}\,\beta_1$
call 2.1	$\mathsf{adm}\ \mathsf{nil}$	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 requries that f has the form $g_1^{2.1}$, we conclude that any function of type $\mathsf{List}\,\mathbb{N} \to X$ (for some type X) is mappable over t relative to the specification $\mathsf{List}\,\beta_1$.

Example 10. For t as in Example 5 the call $\mathsf{adm}\ t\ \ f\ \ \mathsf{List}\,(\mathsf{List}\,\beta_1)$ results in the following sequence of calls:

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

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 requires that f has the form $\mathsf{List}\, g_1^{2.2.1.1}$, 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 $\mathsf{map}_{\mathsf{List}}\, f'$ for some type X and function $f' : \mathbb{N} \to X$.

5 Conclusion, Related Work, and Future Directions

This paper develops an algorithm for characterizing those functions on a deep GADT's type arguments that are mappable over its elements. This algorithm, and thus this paper, can in some sense be read as *defining* what it means for a function to be mappable over t. It thus makes a fundamental contribution to the study of GADTs since there is to our knowledge, no already existing definition of characterization of the intuitive notion of mappability for functions over them. More generally, we know of no other careful study of mappability for GADTs.

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

Initial algebra semantics for GADTs have been studied in [10] and [12]. Both of these works interpret GADTs as *discrete* functors; as a result, the functorial action of the functor interpreting a GADT G cannot correctly interpret applications of G's map function to non-identity functions. In addition, [10] cannot handle truly nested data types such as Bush or the GADT G from Example 2. These discrete initial algebra semantics for GADTs thus do not recover the usual initial algebra semantics of nested types when instantiated to them.

The functorial completion semantics of [13], by contrast, does interpret GADTs as non-discrete functors. However, this is achieved at the cost of adding "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 $\mathsf{map}_{\mathsf{Seq}}\, f\, (\mathsf{pair}\, x\, y)$ even though these may not be of the form $\mathsf{pair}\, u\, v$ for any terms u and v. Importantly, functorial completion adds no junk to interpretations of nested types or ADTs, so unlike the semantics of [12], that of [13] does extend the usual initial algebra semantics for them. But since the interpretations of [13] are bigger than expected for proper GADTs, this semantics, too, is unacceptable.

A similar attempt to recover functoriality is made in [9] to salvage the method from [10]. The overall idea is to relax the discreteness of the functors interpreting GADTs by replacing the dependent products and sums in the development of [10] with left and right Kan extensions, respectively. Unfortunately, this entails that the domains of the functors interpreting GADTs must be the category of all interpretations of types and all morphisms between them, which again leads to the inclusion of unwanted junk elements in the GADT's interpretation.

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 data 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 of *shapes* and, for each shape S, a set of *positions* in S. The functorial action of the functor D interpreting D 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 $f : \mathsf{A} \to \mathsf{B}$ 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 [3] and again in [10]. The authors of [3] propose encoding strictly positive indexed data types in terms of some syntactic combinators they consider "categorically inspired". But as far as we understand their claim, map functions and their interpretations as functorial actions are not worked out for indexed containers. The encoding in [3] is nevertheless essential to understanding GADTs and other inductive families as "structures containing data". With respect to it, our algorithm actually dicovers the shape of its input element, and thus can be understood as determining how "containery" a given GADT is.

Acknowledgements 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. Abbott, M., Altenkirch, T., Ghani, N.: Categories of Containers. In: Foundations of Software Science and Computation Structures, pp. 23-38 (2003). https://doi.org/10.1007/3-540-36576-1_2

2. Abbott, M., Altenkirch, T., Ghani, N.: Containers - Constructing Strictly Positive Types. Theoretical Computer Science **342**, 3-27 (2005). https://doi.org/10.1016/j.tcs.2005.06.002
3. Altenkirch, T., Ghani, N., Hancock, P., McBride, C., Morris, P.: Indexed Containers. Journal of Functional Programming **25 (e5)** (2015). https://doi.org/10.1017/S095679681500009X
4. Arbib. M., Manes, E.: Algebraic Approaches to Program Semantics. Springer (1986). https://doi.org/10.1007/978-1-4612-4962-7
5. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall (1997).
6. Bird, R., Meertens, L.: Nested Datatypes. In: Mathematics of Program Construction, pp. 52–67 (1998). https://doi.org/10.1007/BFb0054285
7. Cagne, P., Johann, P.: Accepted Artifact (APLAS 2022) supporting "Characterizing Functions Mappable Over GADTs". https://doi.org/10.5281/zenodo.7004589
8. Dougherty, D., Johann, P.: An Improved General E-unification Method. Journal of Symbolic Computation **14**, 303-320 (1992). DOI: https://doi.org/10.1016/0747-7171(92)90010-2
9. Fiore, M.: Discrete Generalised Polynomial Functors. In: Automata, Languages, and Programming, pp. 214-226 (2012). https://doi.org/10.1007/978-3-642-31585-5_22
10. Hamana, M., Fiore, M.: A Foundation for GADTs and Inductive Families: Dependent Polynomial Functor Approach. In: Workshop on Generic Programming, pp. 59-70 (2011). https://doi.org/10.1145/2036918.2036927
11. Johann, P., Ghani, N.: Initial Algebra Semantics is Enough! In: Typed Lambda Calculus and Applications, pp. 207-222 (2007). https://doi.org/10.1007/978-3-540-73228-0_16
12. Johann, P., Ghani, N.: Foundations for Structured Programming with GADTs. In: Principles of Programming Languages, pp. 297-308 (2008). https://doi.org/10.1145/1328897.1328475
13. Johann, P., Polonsky, A.: Higher-Kinded Data Types: Syntax and Semantics. In: Logic in Computer Science, pp. 1-13 (2019). https://doi.org/10.1109/LICS.2019.8785657
14. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple Unification-based Type Inference for GADTs. In: International Conference on Functional Programming, pp. 50-61 (2006). https://doi.org/10.1145/1160074.1159811
15. Reynolds, J. C.: Types, Abstraction, and Parametric Polymorphism. Information Processing **83(1)**, 513-523 (1983).
16. Sheard, T., Pasalic, E.: Meta-programming with Built-in Type Equality. In: Workshop on Logical Frameworks and Meta-languages, pp. 49-65 (2008). https://doi.org/10.1016/j.entcs.2007.11.012
17. Xi, H., Chen, C., Chen, G.: Guarded Recursive Datatype Constructors. In: Principles of Programming Languages, pp. 224–235 (2003). https://doi.org/10.1145/604131.604150