Dualizing Generalized Algebraic Data Types by Matrix Transposition

Klaus Ostermann(✉) and Julian Jabs

University of Tübingen, Germany
{klaus.ostermann, julian.jabs}@uni-tuebingen.de

Abstract. We characterize the relation between generalized algebraic datatypes (GADTs) with pattern matching on their constructors one hand, and generalized algebraic co-datatypes (GAcoDTs) with copattern matching on their destructors on the other hand: GADTs can be converted mechanically to GAcoDTs by refunctionalization, GAcoDTs can be converted mechanically to GADTs by defunctionalization, and both defunctionalization and refunctionalization correspond to a transposition of the matrix in which the equations for each constructor/destructor pair of the (co-)datatype are organized. We have defined a calculus, $GADT^T$, which unifies GADTs and GAcoDTs in such a way that GADTs and GAcoDTs are merely different ways to partition the program.
We have formalized the type system and operational semantics of $GADT^T$ in the Coq proof assistant and have mechanically verified the following results: 1) The type system of $GADT^T$ is sound, 2) defunctionalization and refunctionalization can translate GADTs to GAcoDTs and back, 3) both transformations are type- and semantics-preserving and are inverses of each other, 4) (co-)datatypes can be represented by matrices in such a way the aforementioned transformations correspond to matrix transposition, 5) GADTs are extensible in an exactly dual way to GAcoDTs; we thereby clarify folklore knowledge about the "expression problem".
We believe that the identification of this relationship can guide future language design of "dual features" for data and codata.

1 Introduction

The duality between data and codata, between construction and destruction, between smallest and largest fixed points, is a long-standing topic in the PL community. While some languages, such as Haskell, do not distinguish explicitly between data and codata, there has been a "growing consensus" [1] that the two should not be mixed up. Many ideas that are well-known from the data world have counterparts in the codata world. One work that is particularly relevant for this paper are copatterns, also proposed by Abel et al. [1]. Using copatterns, the language support for codata is very symmetrical to that for data: Data types are defined in terms of constructors, functions consuming data are defined using pattern matching on constructors; codata types are defined in terms of destructors, functions producing codata are defined using copattern matching on destructors.

Another example of designing dual features for codata is the recently proposed codata version of inductive data types [37]. However, coming up with these counterparts requires ingenuity. The overarching goal of this work is to replace the required ingenuity by a mechanical derivation. A key idea towards this goal has been proposed by Rendel et al. [32], namely to relate the data and codata worlds by refunctionalization [16] and defunctionalization [33, 17].

Defunctionalization is a global program transformation to transform higher-order programs into first-order programs. By defunctionalizing a program, higher-order function types are replaced by sum types with one variant per function that exists in the program. For instance, if a program contains two functions of type $Nat \to Nat$, then these functions are represented by a sum type with two variants, one for each function, whereby the type components of each variant store the content of the free variables that show up in the function definition. Defunctionalized function calls become calls to a special first-order *apply* function which pattern-matches on the aforementioned sum type to dispatch the call to the right function body.

Refunctionalization is the inverse transformation, but traditionally it only works (easily) on programs that are in the image of defunctionalization [16]. In particular, it is not clear how to refunctionalize programs when there is more than one function (like *apply*) that pattern-matches on the same data type. Rendel et al. [32] have shown that this problem goes away when functions are generalized to arbitrary codata, because then every pattern-matching function in a program to be refunctionalized can be expressed as another destructor. Functions are the special codata type with only one *apply* destructor. Without parametric polymorphism, however, which is not available in the (co)data languages of Rendel et al. [32], we have to define a new function codata type for each pair of input and output types that we want to define higher-order functions for. The code below shows how a new first-class function `square` (that can be passed as an argument and returned as a result) from `Nat` to `Nat` may be defined by a codata generator function with return type `FunctionNatNat`, where copattern matching is used to define the behavior of the first-class function when it is applied with some input `n`.

```
codata FunctionNatNat where
  apply(FunctionNatNat, Nat): Nat

function square(): Function where
  apply(square(),n) = n * n
```

The main goal of this work is to extend the de- and refunctionalization correspondence between data and codata to generalized algebraic datatypes (GADTs) [41, 8] and their codata counterpart, which we call Generalized Algebraic Codata types (GAcoDTs). More concretely, this paper makes the following contributions.

– We present the syntax, operational semantics, and type system of a language, $GADT^T$, that can express both GADTs and GAcoDTs. In this language,

GADTs and GAcoDTs are unified in such a way that they are merely two different representations of an abstract "matrix" interface.
- We show that the type system is sound by proving progress and preservation [40].
- We formally define defunctionalization and refunctionalization, observe that they correspond to matrix transposition, and prove that GADTs and GAcoDTs are indistinguishable after hiding them behind the aforementioned matrix interface. We conclude that defunctionalization and refunctionalization preserve both operational semantics and typing.
- We prove that both GADTs and GAcoDTs can be extended in a modular way (with separate type checking) by "adding rows" to the corresponding matrix. Due to their matrix transposition relation, this means that the extensibility is exactly dual, which clarifies earlier informal results on the "expression problem" [38, 34, 11].
- The language and all results have been formalized and mechanically verified in the Coq proof assistant. The Coq sources are available in the supplemental material that accompanies this submission.
- As a small side contribution, if one considers only the GADT part of the language, this is to the best of our knowledge the first mechanically verified formalization of GADTs. It is also simpler than previous formalizations of GADTs because it is explicitly typed and hence avoids the complications of type inference.

The remainder of this paper is structured as follows. In Section 2 we give an informal overview of our main contributions by means of an example and using conventional concrete syntax. In Section 3 we present the syntax, operational semantics, and type system of $GADT^T$. Section 4 presents the aforementioned mechanically verified properties of $GADT^T$. In Section 5, we discuss applications and limitations of $GADT^T$, talk about termination/productivity and directions for future work, and describe how we formalized $GADT^T$ in Coq. Finally, Section 6 discusses related work and Section 7 concludes.

2 Informal Overview

Figure 1 illustrates the language design of $GADT^T$ in terms of an example. The left-hand side shows an example using GADTs and functions that pattern-match on GADT constructors. The right-hand side shows the same example using GAcoDTs and functions that copattern-match on GAcoDT destructors. The right-hand side is the refunctionalization of the left hand side; the left-hand side is the defunctionalization of the right-hand side.

Simply-typed (Co)Datatypes. Let us first look at the `Nat` (co)datatype. Every data or codata type has an *arity*: The number of type arguments it receives. Since $GADT^T$ does only feature types of kind `*`, we simply state the number of type arguments in the (co)data type declaration. `Nat` receives zero

```
data Nat[0] where
  zero(): Nat
  succ(Nat): Nat

function add(Nat,Nat): Nat where
  add(zero(), x) = x
  add(succ(y),x) = succ(add(y,x))

data List[1] where
  nil[A]():  List[A]
  cons[A](A, List[A]): List[A]

function length[A](List[A]): Nat w..
  length[_](nil[_]) = 0
  length[B](cons[_](x,xs)) =
    succ(length[B](xs))

function sum(List[Nat]): Nat
  sum(nil[_]) = 0
  sum(cons[_](x,xs)) = x + sum(xs)

data Tree[1] where
  node(Nat): Tree[Nat]
  branch[A](List[Tree[A]])
           : Tree[List[A]]

function unwrap(Tree[Nat]): Nat w..
  unwrap(node(n)) = n
  unwrap(branch[_](xs)) = impossible

function width[A](Tree[A]): Nat w..
  width[_](node(n)) = 0
  width[_](branch[C](xs)) =
    length[C](xs)
```

```
codata Nat[0] where
  add(Nat,Nat) : Nat

function zero(): Nat where
  add(zero(),x) = x

function succ(Nat): Nat where
  add(succ(y),x) = succ(add(y,x))

codata List[1] where
  length[A](List[A]): Nat
  sum(List[Nat]): Nat

function nil[A](): List[A] where
  length[_](nil[_]) = 0
  sum(nil[_]) = 0

function cons[A](A, List[A]): List[A] w..
  length[B](cons[_](x,xs)) =
    succ(length[B](xs))
  sum(cons[_](x,xs)) = x + sum(xs)

codata Tree[1] where
  unwrap(Tree[Nat]) : Nat
  width[A](Tree[A]): Nat

function node(Nat): Tree[Nat] where
  unwrap(node(n)) = n
  width[_](node(n)) = 0

function branch[A](List[Tree[A]])
       : Tree [List[A]] where
  unwrap(branch[_](xs)) = impossible
  width[_](branch[C](xs)) =
    length[C](xs)
```

Fig. 1. The same example in the data fragment (left) and codata fragment (right)

List[1]	nil[A](): List[A]	cons[A](A, List[A]): List[A]
length[A](List[A]): Nat	length[_](nil[_]) = 0	length[B](cons[_](x,xs)) = succ(length[B](xs))
sum(List[Nat]): Nat	sum(nil[_]) = 0	sum(cons[_](x,xs)) = x + sum(xs)

Fig. 2. Matrix representation of List GADT from Figure 1 (left)

`List[1]`	`length[A](List[A]): Nat`	`sum(List[Nat]): Nat`
`nil[A](): List[A]`	`length[_](nil[_]) = 0`	`sum(nil[_]) = 0`
`cons[A](A, List[A]): List[A]`	`length[B](cons[_](x,xs)) = succ(length[B](xs))`	`sum(cons[_](x,xs)) = x + sum(xs)`

Fig. 3. Matrix representation of List GAcoDT from Figure 1 (right). This matrix is the transposition of Figure 2.

type arguments, hence `Nat` illustrates the simply-typed setting with no type parameters. Functions in $GADT^T$, like `add` on the left-hand side, are first-order only; higher-order functions can be encoded as codata instead. Functions always (co)pattern-match on their first argument. (Co)pattern matching on multiple argument as well as nested and deep (co)pattern matching are not supported directly and must be encoded via auxiliary functions. We see that the refunctionalized version of `Nat` on the right-hand side turns constructors into functions, functions into destructors, and pattern matching into copattern matching. Abel et al. [1] use "dot notation" for copattern matching and destructor application; for instance, they would write `succ(y).add(x) = succ(y.add(x))` instead of `add(succ(y),x) = succ(add(y,x))` on the right-hand side of Figure 1. We use the same syntax for constructor calls, function calls, and destructor calls because then the equations are not affected by de- and refunctionalization.

Parametric (Co)Datatypes. The `List` datatype illustrates the classical special case of GADTs with no indexing. Type arguments of constructors, functions, and destructors are both declared and passed via rectangular brackets `[...]` (loosely like in Scala). Like System F, $GADT^T$ has no type inference; all type annotations and type applications must be given explicitly. $GADT^T$ has a redundant way of binding type parameters. When defining an equation of a polymorphic function with a polymorphic first argument, we use square brackets to bind both the type parameters of the function and of the constructor/destructor on which we (co)pattern-match. For instance, in the equation `length[B](cons[_](x,xs)) = ...` on the left hand side, `B` is the type parameter of the `length` function, whereas the underscore (which we use if the type argument is not relevant, we could replace it by a proper type variable name) binds the type argument of the constructor with which the list was created. In this example, we could have also written the equation as `length[_](cons[B](x,xs)) = ...` because both type parameters must necessarily be the same, but in the general case we need access to both sets of type variables (as the next example will illustrate). It is important that we do not (co)pattern-match on type arguments, since this would destroy parametricity; rather, the `[...]` notation on the left hand side of an equation is only a binding construct for type variables.

Type Parameter Binding. Of those two sets of type parameter bindings, the one for functions is in a way always redundant because we could use the type variable declaration inside the function declaration instead. For instance, in the equation `length[B](cons[_](x,xs)) = succ(length[B](xs))` on the left hand side we

could use the type parameter `A` of the enclosing function declaration instead. However, in $GADT^T$ the scope of the type variables in the function declaration does not extend to the equations and the type arguments must be bound anew in every equation. The reason for that is that we want to design the equations in such a way that they do not need to be touched when de/refunctionalizing a (co)datatype. For instance, when refunctionalizing a datatype, a function declaration is turned into a destructor declaration and what used to be a type argument that was bound in the enclosing function declaration becomes a type argument that is bound in a remote destructor declaration; to make type-checking modular we hence need a local binding construct. Our main goal in designing $GADT^T$ was not to make it convenient for programmers but to make the relation between GADTs and GAcoDTs as simple as possible; furthermore, a less verbose surface syntax could easily be added on top.

If we look at the corresponding `List` codatatype on the right-hand side, we see that the `sum` function from the left-hand side, which accepts only a list of numbers, turns into a destructor that is only applicable to those instances of `List` whose type parameter is `Nat`. This is similar to methods in object-oriented programming whose availability depends on type parameters [29], but here we see that this feature arises "mechanically" by the de/refunctionalization correspondence.

GA(co)DTs. The `Tree` (co)datatype illustrates a usage of GA(co)DTs that cannot be expressed with traditional parametric data types. We can see that by looking at the return type of the constructors of the `Tree` datatype; they are `Tree[Nat]` and `Tree[List[A]]` instead of `Tree[A]`. The `Tree` codatatype is also using the power of GAcoDTs in the `unwrap` destructor[1] because its first argument is different from `Tree[A]`. The GADT constructor `node(Nat): Tree[Nat]` turns into a function that returns a `Tree[Nat]` on the right hand side. The `Tree` example illustrates two additional issues that did not show up in the earlier examples.

First, it illustrates that type unification may make some pattern matches impossible, as illustrated by the `unwrap(branch[_](xs)) = impossible` equation on the left hand side. The equation is impossible, because the function argument type `Tree[Nat]` cannot be unified with the constructor return type `Tree[List[A]]`.[2] In $GADT^T$, we require that pattern matching is always complete, but impossible equations are not type-checked; the right-hand side can hence be filled with any dummy term. Second, the equation `width[_](branch[C](xs)) = length[C](xs)` illustrates the case where it is essential that we can bind constructor type arguments; otherwise we would have no name for the type argument we need to pass to `length`. Such type arguments are sometimes called *existential* or *phantom* [8] because if we have a branch of type `Tree[A]`, we only

[1] The `unwrap` destructor is meant to be used to extract the number from a tree that directly contains a number, i.e., a tree constructed with constructor `node`.

[2] This fits with our intention that `unwrap` should only work on a `node` (which directly contains a number).

know that there exists some type that was used in the invocation of the `branch` constructor, but that type does not show up in the structure of `Tree[A]`.

We see again how both impossible equations and the need to access constructor type arguments translate naturally into corresponding features in the codata world. For impossible equations, we need to check whether the first destructor argument type can be unified with the function return type. Access to existential constructor type arguments turns into access to local function types; conversely, access to existential destructor type arguments in the codata world turns into access to local function type arguments.

$GADT = GAcoDT^T$. We can see that the relation between GADTs and GAcoDTs is as promised when looking at Figure 2 and Figure 3. These two figures show a slightly different representation of the `List` (co)datatype and associated functions from Figure 1. In this presentation, we have dropped all keywords from the language, such as `function`, `data` and `codata`. The reason for dropping these keywords is that now function signatures in the data fragment look the same as destructor signatures in the codata fragment, and constructor signatures in the data fragment look the same as function signatures in the codata fragment. Figure 2 organizes the datatype in the form of a matrix: the first row lists the datatype and its constructor signatures, the first column lists the signatures of the functions that pattern-match on the datatype, the inner cells represent the equations for each combination of constructor and function. Figure 3 does the same for the `List` codatatype: The first row lists the codatatype and its destructor signatures, the first column lists the signatures of functions that copattern-match on the codatatype, the inner cells represent the equations for each combination of function and destructor. We can now see that the relation between GADTs and GAcoDTs is now indeed rather simple: It is just matrix transposition.

An essential property of this transformation is that other (co)datatypes and functions are completely unaffected by the transformation. For instance, the `Tree` datatype (or codatatype, regardless of which version we use) looks the same, regardless of whether we encode `List` in data or in codata style. Defunctionalization and refunctionalization are still global transformations in that we need to find all functions that pattern-match on a datatype (for refunctionalization) or find all functions that copattern-match on a codatatype (for defunctionalization), but the rest of the program, including all clients of those (co)datatypes and functions, remain the same.

Infinite codata, termination, productivity. The semantics of codata is usually defined via greatest fixed point constructions that include the possibility to represent "infinite" structures, such as streams. This is not the focus of this work, but since our examples so far did not feature such "infinite" structures but we do not want to give the impression that our codata types do somehow lack the expressiveness to express streams and the like, hence we show here an example of how to encode a stream of zeros, both in the codata representation (left) and, defunctionalized, in the data representation (right).

```
codata Stream where
  head(Stream) : Nat
  tail(Stream) : Stream

function zeros() : Stream
  head(zeros()) = zero()
  tail(zeros()) = zeros()
```

```
data Stream where
  zeros() : Stream

function head(Stream) : Nat
  head(zeros()) = zero()

function tail(Stream) : Stream
  tail(zeros()) = zeros()
```

Codata is also often associated with guarded corecursion to ensure productivity. In the copattern formulation of codata, productivity and termination coincide [2]. Due to our unified treatment of data and codata, a single check is sufficient for both termination/productivity of programs. In Section 5.3, we discuss a simple syntactic check that corresponds to both structural recursion and guarded corecursion.

Properties of $GADT^T$. In the remainder of this paper, we formalize $GADT^T$ in a style similar to the matrix representation of (co)datatypes we have just seen. We define typing rules and a small-step operational semantics and prove formal versions of the following informal theorems: 1) The type system of $GADT^T$ is sound (progress and preservation), 2) Defunctionalization and refunctionalization (that is, matrix transposition) of (co)datatypes preserves well-typedness and operational semantics, 3) Both types of matrices are modularly extensible in one dimension, namely by adding more rows to the matrix. This means that we can modularly add constructors or destructors and their respective equations without breaking type soundness as long as the new equations are sound themselves.

3 Formal Semantics

We have formalized $GADT^T$ and all associated theorems and proofs in Coq[3]. Here we present a traditional representation of the formal syntax using context-free grammars, a small-step operational semantics, and a type system.

We have formalized the language in such a way that we abstract over the physical representation of matrices as described in the previous section, hence we do not need to distinguish between GADTs and GAcoDTs. In the following, we say *constructor* to denote either a constructor of a datatype, or a function that copattern-matches on a codatatype. We say *destructor* to denote either a function that pattern-matches on a datatype, or a destructor of a codatatype. The language is defined in terms of constructors and destructors; we will later see that GADTs and GAcoDTs are merely different organizations of destructors and constructors.

3.1 Language Design Rationale

Our main goal in the formalization is to clarify the relation between GADTs and GAcoDTs, and not to design a calculus that is convenient to use as a program-

[3] Full Coq sources are available in the supplemental material.

ming language. Hence we have left out many standard features of programming calculi that would have made the description of that relation more complicated. In particular:

- Like System F, $GADT^T$ requires explicit type annotations and explicit type application. Type inference could be added on top of the calculus, but this is not in the scope of this work.
- (Co)pattern matching is restricted in that every function must necessarily (co)pattern-match on its first argument, hence (co)pattern-matching on multiple arguments or "deep" (co)pattern matching must be encoded by auxiliary functions. Pattern matching is only supported for top-level function definitions; there is no "case" or "match" construct. Functions that are not supposed to (co)pattern-match (like the polymorphic identity function) must be encoded by a function that (co)pattern-matches on a dummy argument of type `Unit`.
- First-class functions are supported in the form of codata, but anonymous local first-class functions must be encoded via lambda lifting [3, 26], that is, they must be encoded as top-level functions where the bindings for the free variables are passed as an extra parameter.
- Due to the abstraction over the physical representation of matrices we have not fixed the physical modular structure (a linearization of the matrix as text) of programs . Type checking of matrices simply iterates over all cells in an unspecified order. However, later on we will characterize GADTs and GAcoDTs as two physical renderings of matrices and formally prove the way in which those program organizations are extensible.

3.2 Notational Conventions

As usual, we use the same letters for both non-terminal symbols and meta-variables, e.g., t stands both for the non-terminal in the grammar for terms but inside inference rules it is a meta-variable that stands for any term. We use the notation $\overline{t}$ to denote a list $t_1, t_2, \ldots, t_{|\overline{t}|}$, where $|\overline{t}|$ is the length of the list. We also use list notation to denote iteration, e.g., $P, \Gamma \vdash \overline{t} : \overline{T}$ means $P, \Gamma \vdash t_1 : T_1, \ldots, P, \Gamma \vdash t_{|\overline{t}|} : T_{|\overline{t}|}$. To keep the notation readable, we write $\overline{x} : \overline{T}$ instead of $\overline{x : T}$ to denote $x_1 : T_1, \ldots, x_n : T_n$.

We use the notation $t[x := t']$ to denote the substitution of all free occurrences of x in t by t', and similarly $T[X := T']$ and $t[X := T']$ for the substitution of type variables in types and terms, respectively.

3.3 Syntax

The syntax of $GADT^T$ is defined in Figure 4. Types have the form $m[\overline{T}]$, where m is the name of a GADT or GAcoDT (in the following referred to as *matrix name*), and square brackets to denote type application. Types can contain type variables X. In the syntax of terms t, x denotes parameters that are bound by (co)pattern matching and y denotes other parameters. A constructor call $c[\overline{T}](\overline{t})$ takes zero or

Syntax

$$
\begin{array}{lll}
S, T & ::= m[\overline{T}] \mid X & \textit{Types} \\
t & ::= x \mid y \mid c[\overline{T}](\overline{t}) \mid d[\overline{T}](t, \overline{t}) & \textit{Terms} \\
C & ::= c[\overline{X}](\overline{T}) : m[\overline{T}] & \textit{Constructor Signature} \\
D & ::= d[\overline{X}](m[\overline{T}], \overline{T}) : T & \textit{Destructor Signature} \\
e & ::= d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t & \textit{Equations} \\
M & = (a, \gamma \in \overline{C}, \delta \in \overline{D}, \gamma \to \delta \to e) & \textit{Matrices} \\
P & = m \mapsto_{\mathit{fin}} M & \textit{Programs} \\
m & \in \text{Matrix names} & \\
d & \in \text{Destructor names} & \\
c & \in \text{Constructor names} & \\
x & \in \text{Pattern Variable Names} & \\
y & \in \text{Variable Names} & \\
X, Y & \in \text{Type Variables} & \\
a & \in \mathbb{N} & \textit{Arities}
\end{array}
$$

Operational Semantics : $P \vdash t \to t'$

$$
\begin{array}{lll}
u, v & ::= c[\overline{T}](\overline{v}) & \textit{Values} \\
E & ::= c[\overline{T}](\overline{v}, [], \overline{t}) \mid d[\overline{T}](\overline{v}, [], \overline{t}) & \textit{Evaluation Context}
\end{array}
$$

$$
\frac{P \vdash t \to t'}{P \vdash E[t] \to E[t']} \quad \text{(E-Ctx)}
$$

$$
\frac{\begin{array}{c} m \mapsto (a, \overline{C}, \overline{D}, \mathit{lookup}) \in P \\ D \in \overline{D} \qquad D = d[\ldots](m[\ldots],\ldots) \\ C \in \overline{C} \qquad C = c[\ldots](\ldots) \\ \mathit{lookup}(C, D) = d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t \end{array}}{P \vdash d[\overline{S}](c[\overline{T}](\overline{v}), \overline{u}) \to t[\overline{X} := \overline{S}, \overline{Y} := \overline{T}][\overline{x} := \overline{v}, \overline{y} := \overline{u}]} \quad \text{(E-Fire)}
$$

Fig. 4. Syntax and Operational Semantics of $GADT^T$

more arguments, whereas a destructor call $d[\overline{T}](t, \overline{t})$ takes at least one argument (namely the one to be destructed). Both destructors and constructors can have type parameters, which must be passed via square brackets.

A constructor signature $c[\overline{X}](\overline{T}) : m[\overline{T}]$ defines the number and types of parameters and the type parameters to the constructed type. Its output type cannot be a type variable but must be some concrete matrix type $m[\overline{T}]$. A destructor signature, on the other hand, must have a concrete matrix type as its first argument and can have an arbitrary return type. Equations $d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t$ define what happens when a constructor c meets a destructor d. The $\overline{x}$ bind the components of the constructor, whereas the $\overline{y}$ bind the remaining parameters of the destructor call. We also bind both the type arguments to the constructor $\overline{X}$ and the destructor $\overline{Y}$, such that they can be used inside t. In many cases, the $\overline{X}$ will provide access to the same types as $\overline{Y}$, but in the general case we need both because both constructors and destructors may contain phantom types [8].

Matrices M are an abstract representation of both GADTs and GAcoDTs, together with the functions that pattern-match (for GADTs) or copattern-match (for GAcoDTs) on the GA(co)DTs. A matrix has an arity a (the number of type parameters it receives), a list of constructors γ, and a list of destructors δ. It also has a lookup function that returns an equation for every constructor/destructor pair on which the matrix is defined (hence the type of matrices is a dependent type). There must be an equation for each constructor/destructor pair, but in the case of impossible combinations, the equations are not type-checked and some dummy term can be inserted. A program P is just a finite mapping from matrix names to matrices.

3.4 Operational Semantics

We define the operational semantics, also in Figure 4, via an evaluation context E, which, together with E-Ctx, defines a standard call-by-value left-to-right evaluation order. Not surprisingly, the only interesting rule is E-Fire, which defines the reduction behavior when a destructor meets a constructor. We look up the corresponding matrix in the program and look up the equation for that constructor/destructor pair. In the body of the equation, t, we perform two substitutions: 1) We substitute the formal type arguments $\overline{X}$ and $\overline{Y}$ by the current type arguments $\overline{S}$ and $\overline{T}$, and 2) we substitute the pattern variables $\overline{x}$ by the components $\overline{v}$ of the constructor and the variables $\overline{y}$ by the current arguments $\overline{u}$.

3.5 Typing

The typing and well-formedness rules are defined in Figure 5. Let us first look at the typing of terms. The rules for variable lookup are standard. The constructor rule T-Const checks that the number of type- and term arguments matches the declaration and checks the type of all arguments, whereby the type variables

Term Typing : $P, \Gamma \vdash t : T$

$$\Gamma ::= \epsilon \mid x : T, \Gamma \mid y : T, \Gamma \;\; \textit{Typing Contexts}$$

$$\frac{x : T \in \Gamma}{P, \Gamma \vdash x : T}\ \text{(T-Pvar)} \qquad \frac{y : T \in \Gamma}{P, \Gamma \vdash y : T}\ \text{(T-Var)}$$

$$\frac{\begin{array}{c} P, \Gamma \vdash t : m[\overline{T}][\overline{X} := \overline{S}] \\ m \mapsto (\ldots, \ldots d[\overline{X}](m[\overline{T}], \overline{U}) : T \ldots, \ldots) \in P \\ \forall i. P, \Gamma \vdash t_i : U_i[\overline{X} := \overline{S}] \\ |\overline{X}| = |\overline{S}| \qquad |\overline{U}| = |\overline{t}| \end{array}}{P, \Gamma \vdash d[\overline{S}](t, \overline{t}) : T[\overline{X} := \overline{S}]}\ \text{(T-Dest)}$$

$$\frac{\begin{array}{c} \ldots \mapsto (\ldots c[\overline{X}](\overline{T}) : T \ldots, \ldots, \ldots) \in P \\ \forall i. P, \Gamma \vdash t_i : T_i[\overline{X} := \overline{S}] \\ |\overline{X}| = |\overline{S}| \qquad |\overline{T}| = |\overline{t}| \end{array}}{P, \Gamma \vdash c[\overline{S}](\overline{t}) : T[\overline{X} := \overline{S}]}\ \text{(T-Const)}$$

Well-Formedness

$$\frac{\begin{array}{c} C = c[\overline{X'}](\overline{T}) : m[\overline{S}] \qquad |\overline{X}| = |\overline{X'}| \\ D = d[\overline{Y'}](m[\overline{S'}], \overline{T'}) : T \qquad |\overline{Y}| = |\overline{Y'}| \\ \textit{all-distinct}(\overline{X}, \overline{Y}) \qquad \textit{all-distinct}(\overline{X'}, \overline{Y'}) \\ \textit{most-general-unifier}(m[\overline{S}], m[\overline{S'}]) = \sigma \\ P, \overline{x} : \sigma(\overline{T}), \overline{y} : \sigma(\overline{T'}) \vdash \sigma(t[\overline{X} := \overline{X'}, \overline{Y} := \overline{Y'}]) : \sigma(T) \end{array}}{P, m \vdash d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t \text{ OK in } C, D}\ \text{(Wf-Eq)}$$

$$\frac{\begin{array}{c} C = \ldots : m[\overline{S}] \qquad D = \ldots (m[\overline{S'}], \ldots) : \ldots \\ \textit{most-general-unifier}(m[\overline{S}], m[\overline{S'}]) = \text{error} \end{array}}{P, m \vdash d[\overline{Y}](c[\overline{X}](\overline{x}), \overline{y}) = t \text{ OK in } C, D}\ \text{(Wf-Infsble)}$$

$$\frac{|\overline{S}| = a \qquad FV(\overline{T}) \subseteq \overline{X} \qquad FV(\overline{S}) \subseteq \overline{X}}{c[\overline{X}](\overline{T}) : m[\overline{S}] \text{ OK in } m, a}\ \text{(Wf-Constr)}$$

$$\frac{|\overline{S}| = a \qquad FV(\overline{S}) \subseteq \overline{Y} \qquad FV(\overline{T}) \subseteq \overline{Y}}{d[\overline{Y}](m[\overline{S}], \overline{T}) : T \text{ OK in } m, a}\ \text{(Wf-Destr)}$$

$$\frac{\begin{array}{c} \forall C \in \overline{C}, \forall D \in \overline{D}, \\ C \text{ OK in } m, a \\ D \text{ OK in } m, a \\ P, m \vdash \textit{lookup}(C, D) \text{ OK in } C, D \\ \textit{all-names-distinct}(\overline{D}) \end{array}}{m \mapsto (a, \overline{C}, \overline{D}, \textit{lookup}) \text{ OK in } P}\ \text{(Wf-Matr)}$$

$$\frac{\begin{array}{c} \forall m \in \textit{dom}(P), m \mapsto P(m) \text{ OK in } P \\ \textit{all-names-distinct}(\textit{ctors}(P)) \end{array}}{P \text{ OK}}\ \text{(Wf-Prog)}$$

Fig. 5. Typing and Well-Formedness

are substituted by the type arguments of the actual constructor call. Constructor names must be globally unique, hence the matrix to which the constructor belongs is not relevant.

This is different for typing destructor calls (T-Dest). A destructor is resolved by first determining the matrix m of the first destructor argument, and then the destructor is looked up in that matrix. It is hence OK if the same destructor name shows up in multiple matrices. When considering codata as "objects" like in object-oriented programming [25], this corresponds to the familiar situation that different classes can define methods with the same name. In the GADT case, this corresponds to allowing multiple pattern-matching functions of the same name that are disambiguated by the type of their first argument.

In Wf-Eq, we construct the appropriate typing context to type-check the right hand side of equations. We allow implicit α-renaming of type variables to prevent accidental name clashes (checked by *all-distinct*). We compute the most general unifier of the two matrix types in the constructor and destructor, respectively, to combine the type knowledge about the matrix type from the constructor and destructor type. If no such unifier exists, the equation is vacuously well-formed because the particular combination of constructor and destructor can never occur during execution of well-typed terms (Wf-Infsble). Otherwise, we use the unifier σ and apply it to the given type annotations to type-check the term t. A unifier σ is a mapping from type variables to types, but we also use the notation $\sigma(t)$ and $\sigma(T)$ to apply σ to all occurrences of type variables inside a term t or a type T, respectively.

Constructor and destructor signatures are well-formed if they apply the correct number of type parameters to the matrix type and contain no free type variables (Wf-Constr and Wf-Destr). A matrix is type-checked by making sure that all constructor and destructor signatures are well-formed, that all equations are well-formed for every constructor/destructor combination, and that destructor names are unique in the matrix (Wf-Matr). To check uniqueness of names, we use *all-names-distinct*, which checks for a given list of signatures that all of their names are distinct. A program is well-formed if all of its matrices typecheck and the constructor signatures of the program (retrieved by *ctors*) are globally unique (Wf-Prog).

3.6 GADTs and GAcoDTs

In the formalization so far, we have deliberately kept matrices abstract as a kind of abstract data type. Now we can bring in the harvest of our language design. GADTs and GAcoDTs are two different physical representations of matrices, see Figure 6. They both contain nested vectors of equations and differ only in the order of the indices. With GADTs, the column labels are constructors and the row labels functions and a row corresponds to a function defined by pattern matching, with one equation for each case of the GADT. With GAcoDTs, the column labels are destructors, the row labels are functions, and a row corresponds to a function defined by copattern matching, with one equation for each case of

$$
\begin{array}{ll}
M_{GADT} & = (a, \gamma \in \overline{C}, \delta \in \overline{D}, \{e_{D,C} | D \in \delta, C \in \gamma\}) \\
M_{GAcoDT} & = (a, \gamma \in \overline{C}, \delta \in \overline{D}, \{e_{C,D} | C \in \gamma, D \in \delta\}) \\
 & \\
\mathit{mkmatrix} & : M_{GADT} + M_{GAcoDT} \to M \\
\mathit{mkmatrix} & = \text{— obvious; omitted} \\
 & \\
\mathit{refunctionalize} & : M_{GADT} \to M_{GAcoDT} \\
\mathit{refunctionalize} & = \mathit{transpose} \\
 & \\
\mathit{defunctionalize} & : M_{GAcoDT} \to M_{GADT} \\
\mathit{defunctionalize} & = \mathit{transpose}
\end{array}
$$

Fig. 6. GADTs and GAcoDTs

the GAcoDT. Hence both *defunctionalize* and *refunctionalize*, which swap the respective organization of the matrix, are just matrix transposition.

4 Properties of $GADT^T$

In this section, we prove type soundness for $GADT^T$, the preservation of typing and operational semantics under de- and refunctionalization, and that our physical matrix representations of GADTs and GAcoDTs are accurate with respect to extension. All of these properties have been formalized and proven in Coq, based upon our Coq formalization of the previous section's formal syntax, semantics, and type system.

4.1 Type Soundness

We start with the usual progress and preservation theorems.

Theorem 1 (Progress). *If P is a well-formed program and t is a term with no free type variables and $P, \epsilon \vdash t : T$, then t is either a value v, or there exists a term t' such that $P \vdash t \to t'$.*

The proof of this theorem is a simple induction proof using a standard canonical forms lemma [31].

Preservation is much harder to prove. Often, preservation is proved using a substitution lemma which states that the substitution of a (term) variable by a term of the same type does not change the type of terms containing that term variable [31]. In $GADT^T$, this lemma looks as follows:

Lemma 1 (Term Substitution). *If $\overline{t}$ is a list of terms with $P, \epsilon \vdash \overline{t} : \overline{T}$ and $\overline{t'}$ is a list of terms with $P, \epsilon \vdash \overline{t'} : \overline{T'}$ and t is a term with $P, \overline{x} : \overline{T}, \overline{y} : \overline{T'} \vdash t : T$, then $P, \epsilon \vdash t[\overline{x} := \overline{t}, \overline{y} := \overline{t'}] : T$*

However, in E-Fire we perform both a substitution of terms and of types, hence the term substitution lemma is not enough to prove preservation; we also need a type substitution lemma.

Lemma 2 (Type Substitution). *If* $P, \Gamma \vdash t : T$*, then* $P, \Gamma[\overline{X} := \overline{T}] \vdash t[\overline{X} := \overline{T}] : T[\overline{X} := \overline{T}]$

The proof of this lemma requires various auxiliary lemmas about properties (such as associativity) of type substitution. Taken together, these two lemmas are the two main intermediate results to prove the desired preservation theorem.

Theorem 2 (Preservation). *If* P *is a well-formed program and* t *is a term with no free type variables and* $P, \epsilon \vdash t : T$ *and* $P \vdash t \to t'$*, then* $P, \epsilon \vdash t' : T$*.*

4.2 Defunctionalization and Refunctionalization

The preservation of typing and operational semantics by de/refunctionalization is a trivial consequence of the lemma below, which holds due to the fact that both de- and refunctionalization is merely matrix transposition, see Figure 6, and that the embedding *mkmatrix* of the physical matrices into the abstract representation ignores the organization of the physical matrices.

Lemma 3 (Matrix Transposition).
$\forall m \in M_{GADT}$, $mkmatrix(m) = mkmatrix(refunctionalize(m))$.
$\forall m \in M_{GAcoDT}$, $mkmatrix(m) = mkmatrix(defunctionalize(m))$.

Corollary 1 (Preservation of typing and reduction).
De/refunctionalization of a matrix does not change the well-typedness of a program or the operational semantics of a term.

4.3 Extensibility

So far, we have seen that our chosen physical matrix representations are amenable to easy proofs of the preservation of properties under de- and refunctionalization. However, are they also indeed accurate representations of GADTs and GAcoDTs? GADTs and GAcoDTs are utilized due to their *extensibility* along the destructor or constructor dimension, respectively, so we want this to be reflected by our representations.

We assume that matrices are represented as a traditional linear program by reading them row-by-row. Adding a new row is a non-invasive operation (adding to the program), whereas adding a column requires changes to the existing program.

We want to be able to extend our matrix representations with a new row, respectively representing the addition of a new destructor or constructor, without breaking well-typedness as long as the *newly added* equations typecheck with

respect to the complete new program, and uniqueness of destructor/constructor names is preserved (globally, in the constructor case)[4].

In order to formally state that this is indeed the case, we first formally capture extension of GADT and GAcoDT matrices with the following definitions. These already include the preservation of local uniqueness as a condition, i.e., the name of the newly added destructor or constructor must be fresh within the matrix.

Definition 1 (GADT extension). *Consider an* $m \in M_{GADT}$ *with* $m = (a, \gamma, \delta, \{e_{D,C} | D \in \delta, C \in \gamma\})$. *For any* $D' \in \overline{D}, D' \notin \delta$, *and equations* $e_{D',C}$, *for each* $C \in \gamma$, *we call* $(a, \gamma, \delta \cup \{D'\}, \{e_{D,C} | D \in \delta \cup \{D'\}, C \in \gamma\})$ *a* GADT extension of m with D' and $\{e_{D',C} | C \in \gamma\}$.

Definition 2 (GAcoDT extension). *Consider an* $m \in M_{GAcoDT}$ *with* $m = (a, \gamma, \delta, \{e_{C,D} | C \in \gamma, D \in \delta\})$. *For any* $C' \in \overline{C}, C' \notin \gamma$, *and equations* $e_{C',D}$, *for each* $D \in \delta$, *we call* $(a, \gamma \cup \{C'\}, \delta, \{e_{C,D} | C \in \gamma \cup \{C'\}, D \in \delta\})$ *a* GAcoDT extension of m with C' and $\{e_{C',D} | D \in \delta\}$.

We now straightforwardly lift these definitions to programs: A program P' *is a GA(co)DT extension (with some signature and equations) of another program* P if their matrices are identical except for one matrix name, and the underlying physical matrix (packed with *mkmatrix*) assigned to this name under P' is GA(co)DT extension (with this signature and equations) of the underlying physical matrix assigned under P.

Using this terminology we can now formally state and prove the extensibility of GADTs and GAcoDTs:

Theorem 3 (Datatype Extensibility).
If P *is a well-formed program, and* P' *is a GADT extension of* P *with* D' *and equations* $\{e_{D',C} | C \in \gamma\}$, *for the constructor signatures* γ *of the matrix to be extended, such that* $P', m \vdash e_{D',C}$ *OK in C,D' for each* $C \in \gamma$, *then* P' *is well-formed.*

Theorem 4 (Codatatype Extensibility).
If P *is a well-formed program, and* P' *is a GAcoDT extension of* P *with* C', *where the name of* C' *is different from all constructor names in* P, *and equations* $\{e_{C',D} | D \in \delta\}$, *for the destructor signatures* δ *of the matrix to be extended, such that* $P', m \vdash e_{C',D}$ *OK in C',D for each* $D \in \delta$, *then* P' *is well-formed.*

In other words, in both cases we can type-check each row of a matrix in isolation, and if we put those rows together the resulting matrix and program containing that matrix will be well-formed. The results justify the familiar physical representation of programs where the variants of a GADT are fixed but we can freely add new functions that pattern-match on that GADT (and correspondingly for GAcoDTs).

[4] The counterpart to this property on the side of the operational semantics is that the reduction relation of the new program restricted to terms befitting the old program equals the reduction relation of the old program; this however we omitted as it holds trivially when uniqueness is preserved.

5 Discussion

In this section we discuss applications and limitations of our work, talk about directions for future work, and describe the Coq formalization of the definitions and proofs.

5.1 Applications

Language Design. The most obvious application of our approach is to guide programming language design, namely by designing its features in such a way that the correspondence by de/refunctionalization is preserved. We believe that we can find "gaps" in existing languages by checking whether the corresponding dual feature exists, or massaging the language feature in such a way that a clear dual exists. For instance, on the datatype and pattern matching side, many features exist that have no clear counterpart on the codata side yet, such as pattern matching on multiple arguments, non-linear pattern matching, or pattern guards [22]. Some vaguely dual features exist on the codata side understood as "objects", e.g. in the form of multi dispatch (such as [10]) or predicate dispatch [21]. We believe that the relation between pattern matching on multiple arguments and multi dispatch is a particularly interesting direction for future work, since it would entail generalizing our two-dimensional matrices to matrices of arbitrary dimension.

Arguably, codata is the essence of object-oriented programming [12]. In any case, we believe that our design can also help to design object-oriented language features. For instance, there has been previous works on "object-oriented" GADTs [27, 20] using extensions of generic types with certain classes of constraints. For instance, in Kennedy and Russo's work, a list interface could be defined like this:

```
interface List<A> {
  Integer size();
  Integer sum() where A=Integer; // Kennedy & Russo's syntax
}
```

If we compare this interface with the `List` codata type in Figure 1 (right hand side), then we can see that such constraints are readily supported by GAcoDTs; not because this feature was explicitly added but because it arises mechanically from dualizing GADTs.

As another potential influence on language design, we believe that "closedness" under defunctionalization and refunctionalization can be a desirable language design quality that prevents oddities that things can be expressed better using codata than using data (or vice versa). For instance, Carette et al. [5] propose a program representation (basically again a form of Church encoding, hence a codata encoding) that works in a simple Haskell'98 language but whose datatype representation would require GADTs. This suggests a language design flaw in that the codata fragment of functions supports a more powerful type

system than the data fragment of (non-generalized) algebraic data types. That is, the type arguments of a codata generator function's result type may be arbitrarily specialized, e.g., the result type might be `List[Nat]`, while the type of a constructor must be fully generic, e.g., `List[A]`. Our approach gives a criterion on when the type systems for both sides are "in sync".

```
codata Func[2] where
  apply[A,B](Func[A,B], A) : B

codata Nat[0] where
  fold[A](Nat,A,Func[A,A]) : A

fun zero(): Nat where
  fold[A](zero(),z,s) = z

fun succ(Nat): Nat where
  fold[A](succ(n),z,s) =
     apply[A,A](s,fold[A](n,z,s))
```

```
data Nat[0] where
  zero() : Nat
  succ(Nat) : Nat

fun fold[A](Nat,A,Func[A,A]) : A where
  fold[A](zero(),z,s) = z
  fold[A](succ(n),z,s) =
     apply[A,A](s, fold[A](n,z,s))
```

Fig. 7. Defunctionalizing Church-encoded numbers (left) yields Peano numbers with a fold function (right)

De/Refunctionalization as a Programmer Tool. Semantics-preserving program transformations are not only interesting on the meta-level of programming language design but also because they define an equivalence relation on programs. For instance, consider the program on the left-hand side of Figure 7, written in our GAcoDT language. `Nat` is a representation of Church-encoded[5] natural numbers as a GAcoDT with arity zero and a singular destructor `fold` with a type parameter `A`. Defunctionalizing `Nat` yields the familiar Peano numbers with the standard fold function (right-hand side).

Such equivalences have been identified as being useful to identify different forms of programs that are "the same elephant". For instance, Olivier Danvy and associates [17, 16] have used defunctionalization, refunctionalization, and some other transformations such as CPS-transformation to inter-derive "semantic artifacts" such as big-step semantics, small-step semantics, and abstract machines ("The inter-derivations illustrated here witness a striking unity of computation, be this for reduction semantics, abstract machines, and normalization function: they all truly define the same elephant." – Danvy et al. [15]).

The applicability of these transformations is widened by our approach since we support arbitrary codata and not just functions. Exploring these new possibilities is an interesting area of future work.

[5] This form of typed Church encoding is sometimes called Böhm-Berarducci encoding [4].

Furthermore, programmers can employ our transformation as a tool for a more practical purpose. Consider that at some point during the development of a large software, it might have been determined that the extensibility dimension for a particular aspect should be switched. That is, it is now thought that instead of allowing to add new variants (constructors), the software would be better poised by fixing the variants and allowing the addition of new operations (destructors), or vice versa. In the case that at this point it is further possible to make a closed-world assumption with regards to the particular type (represented as a matrix), since clients of the code are known and can be dealt with, it might seem reasonable to transpose the matrix representing that type. With $GADT^T$, it is possible to do this independently of the other matrices in the program. (As already discussed, $GADT^T$ in its present form doesn't aim to be particularly developer-friendly, but we expect further language layers to be placed on top of $GADT^T$ to remedy this eventually.)

Compiler Optimizations. To be able to use our automatizable transformation as a programmer tool, it was important to be able to make a closed-world assumption, where we have the entire program, or more precisely, the part which involves the matrix under consideration, at our disposal. A more automated process where such a kind of assumption can often be readily made is compilation. There, our matrix transposition transformation can be employed for a whole program optimization (such as [6]), as follows. An opportunity for optimization presents itself to the compiler when it is basically able to recognize an abstract machine in the code; optimizing this abstract machine is then an intermediate step, more generally applicable, that precedes hardware-specific optimizations [18]. As outlined above, defunctionalization can turn higher-order programs into first-order programs where this machine might be apparent. With our pair of languages, using our readily automatizable defunctionalization (matrix transposition), it is possible to turn GAcoDT code into GADT code during the compilation phase. Then the compiler can leverage the potentially recognizable abstract machine form of the GADT code for its optimizations.

5.2 Limitations

As we said, our design rationale for $GADT^T$ was to clarify the relation between GADTs and GAcoDTs, not to provide a convenient language for developers. Here we discuss some ways to address the limitations resulting from that decision.

Local (Co)Pattern Matching, Including λ. A significant limitation of $GADT^T$ is that (co)pattern matching is only allowed on the top-level; we don't have "case" (or "match") constructs on the term level. Any local (co)pattern matching, however, can be converted to the top-level form by extracting it to a new top-level function definition. Variables free within the (co)pattern matching term must be passed to this function as arguments. In particular, anonymous local first-class functions, i.e., λ expressions, are a form of local copattern matching which can be encoded in this way; this particular conversion is traditionally called lambda lifting.

(Co)Pattern Matching on Zero or More Arguments. (Co)pattern matching in $GADT^T$ is only possible on a single, distinguished argument (in our presentation, the first, but this is not important). Nested and multiple-argument matching can be encoded by *unnesting* à la Setzer et al. [36], producing auxiliary functions.

In $GADT^T$, it is further not possible to define a function without any (co) pattern matching entirely. The workaround of (co)pattern matching on a dummy argument of type `Unit` is simple, but it is not obvious how to reconcile this encoding with the symmetry of de/refunctionalization.

Type Inference. We have deliberately avoided the question of type inference in this work. In general, we expect that the ample existing works on type inference for GADTs (such as Peyton Jones et al. [30], Schrijvers et al. [35], Chen and Erwig [7]) can be adapted to our setting and will also work for GAcoDTs. We see one complication, though: Due to the fact that destructors are only locally unique in $GADT^T$, the (co)datatype the destructor belongs to must first be found via the type inferred for its distinguished, destructed argument. In other words, we do not know which destructor signature to consider before we know the destructed argument's type. This means that a type inference system which works inwards only, i.e., it discovers the types of the destructor arguments by looking at the signature, possibly leaving unification variables, and then checks that the recursively discovered types for the arguments conform, will not work.

5.3 Termination and Productivity

While termination and productivity are not in the focus of this paper, we want to mention that our unified treatment of data and codata can also lead to a unified treatment of termination and productivity.

Here we want to illustrate informally that a simple syntactic criterion is sufficient to allow structural recursion and guarded corecursion. Syntactic termination checks are not expressive enough for many situations, hence we leave a proper treatment of termination/productivity checking (such as with sized types [2]) for future work; the purpose of this discussion is merely to illustrate that termination checking could also benefit from unified data and codata and not to propose a practically useful termination checker.

The basic idea is to restrict destructor calls in the right-hand sides of equations to have the form $d[\overline{T}](x, \overline{t})$ instead of $d[\overline{T}](t, \overline{t})$. That is to say, in destructor calls, we only allow variables from *within* the constructor pattern of the left-hand side This criterion already guarantees termination (and hence also productivity [2]) in our system, i.e. the finiteness of all reduction sequences, which can be shown with the usual argument of a property that strictly decreases under reduction. A reduction step in $GADT^T$ with right-hand sides restricted like that strictly decreases, under lexicographic order, the pair of

1. the maximum of all the first (destructed) arguments depths in destructor calls of the term, and

2. the sequence which counts how often each destructed argument depth appears in the term, starting with the maximum depth and going downward; those sequences are themselves lexicographically ordered.

This strict decrease can be proved by induction on the derivation of the reduction step. Since there are no infinitely decreasing sequences of these pairs, any reduction sequence must be finite. Note that our criterion in itself excludes far too many programs to be anywhere near practical, but it is readily conceivable how to relax it to only *recursive* calls together with a check that excludes mutual recursion.[6]

Let's look at Figure 7 once more to illustrate that this criterion corresponds to both structural recursion and guarded corecursion. In the right-hand side of Figure 7 we see that the first argument to the recursive call in the last line is `n`, which is allowed by our restriction because it is a syntactic part of the original input, `succ(n)` (structural recursion). The call to `apply` is not a problem because it is not a recursive call.[7] At the same time, if we look at the last line in the left-hand side of Figure 7, we see that the criterion also corresponds to guarded corecursion. With copatterns, guarded corecursion means that we do not destruct the result of a recursive call (the "guard" itself is implicit in the pattern on the left-hand side of the equation). However, destructing that result would mean that we would have to call a destructor with the recursive call as its first argument, which is again forbidden by the syntactic criterion.

5.4 Going beyond System F-like polymorphism

A particularly interesting direction for future work is to extend $GADT^T$ and go beyond the System F-like polymorphism. For instance, F_ω contains a copy of the simply-typed lambda calculus on the type level. Could one also generalize type-level functions to arbitrary codata and maybe use a variant of $GADT^T$ on the type level? Can dependent products like in the calculus of constructions [13] be generalized in a similar way? Can inductive types like in the calculus of inductive constructions be formulated such that there is a dual that is also related by de/refunctionalization? Thibodeau et al. [37] have formulated such a dual, but whether it can be massaged to fit into the setting described here is not obvious.

5.5 Coq Formalization

Our Coq formalization is quite close to the traditional presentation chosen for this paper, but there are some technical differences. Both term and type variables are encoded via de Bruijn indices, which is rather standard for programming

[6] For instance one might request the programmer to order the destructor names such that in equations for a certain destructor only destructors of lower order may be called.

[7] As long as we avoid mutual recursion, for instance by ensuring `fold` > `apply`.

language mechanization. More interestingly, the syntax of the language in the Coq formalization expresses some of the constraints we express here via typing rules instead via dependent types. Specifically, terms and types are indexed by the type variables that can appear inside. To represent matrices, we have developed a small library of dependently typed tables (where the cell types can depend on the row and column labels), such that the matrix type already guarantees that all type variables that show up in terms and types are bound. An earlier version of the formalization and the soundness proof used explicit well-formedness constraints to guarantee that all type variables are bound; the type soundness proof for this version was about twice as long as the one using dependent types. On the flip side, we had to "pay" for using the dependent types in the form of many annoying "type casts" in definitions and theorems owing to the fact that Coq's equality is intensional and not extensional [9, Sec. 10.3]. Finally, instead of using an evaluation context to define evaluation order like we did in Figure 4, we have used traditional congruence rules. In the reduction relation as formalized in Coq, a single step can actually correspond to multiple steps in the formalization presented in the paper; however, this is just a minor technicality to slightly simplify the proofs.

6 Related Work

"Theoreticians appreciate duality because it reveals deep symmetries. Practitioners appreciate duality because it offers two-for-the-price-of-one economy." This quote from Wadler [39] describes the spirit behind the design of $GADT^T$, but of course this is not the first paper to talk about duality in programming languages. We have already discussed the most closely related works in previous sections; here, we compare $GADT^T$ with theoretical calculi with related duality properties and point out an aspect of practical programming for which the duality of $GADT^T$ is relevant.

Codata. Hagino [23] pioneered the idea of dualizing data types: Whereas data types are used to define a type by the ways to *construct* it, codatatypes are dual to them in the sense that they are specified by their *deconstructions*. Abel et al. [1] introduce copatterns which allow functions producing codata to be defined by matching on the destructors of the result codatatype, dually to matching on the constructors of the argument datatype. All these developments occur in a world where function types are a given. The symmetric codata and data language fragments proposed by Rendel et al. [32] deviate from this: By enhancing destructor signatures with argument types, they provide a form of codata that is a generalization of first-class functions. Both the works by Rendel et al. and Abel et al. are simply-typed.

The (co)datatypes in the calculus of Downen and Ariola [19] also allow for user-defined function types. Their focus is different from ours, though, as they are mostly interested in evaluation strategies and their duality, and with regards to their calculus itself they work in an untyped setting. What is interesting in

comparison with $GADT^T$ is how their (co)datatype declarations and signatures are inherently more symmetric as they essentially describe a type system for the parametric sequent calculus. As such, the position of additional arguments in the destructor signatures has a mirror counterparts in constructor signatures (to highlight this, Downen and Ariola refer to destructors as "co-constructors").

Duality of Computations and Values. Staying on with the idea of avoiding function types as primitives for a moment, Wadler [39] presents a "dual calculus" in which the previously astonishing result that call-by-name is De Morgan-dual to call-by-value [14] is clarified by defining implication (corresponding to function types via the Curry-Howard isomorphism) in two different ways dependent on the intended corresponding evaluation regime. A somewhat similar approach, but perhaps more directly related to the data/codata duality, that also deals with the "troubling" coexistence of call-by-value and call-by-name, was proposed by Levy [28]. Levy presents a calculus with a new evaluation regime, *call-by-push value* (CBPV), which subsumes call-by-value and call-by-name by encoding the local choice for either in the terms of the calculus. More specifically, there are two kinds of terms in the CBPV calculus: computations and values, which can be inter-converted by "thunking" and "forcing". The terms for computations and values are said to be of *positive* type and of *negative* type, respectively. Thibodeau et al. [37] have built their calculus, which extends codatatypes to indexed codatatypes, on top of CBPV, with datatypes being positive and codatatypes being negative. We think that, when extending $GADT^T$ with local (co)pattern matching on the term level, perhaps with pattern and copattern matching terms mixed, it might be helpful to similarly recast the resulting language as a modification of the CBPV calculus of Levy.

7 Conclusions

We have presented a formal calculus, $GADT^T$, which uniformly describes both GADTs and their dual, GAcoDTs. GADTs and GAcoDTs can be converted back and forth by defunctionalization and refunctionalization, both of which correspond to a transposition of the matrix of the equations for each pair of constructor/destructor. We have formalized the calculus in Coq and mechanically verified its type soundness, its extensibility properties, and the preservation of typing and operational semantics by defunctionalization and refunctionalization.

We believe that our work can be of help for future language design since it describes a methodology to get a kind of "sweet spot" where data and codata constructs (including functions) are "in sync". We think that it can also be useful as a general program transformation tool, both on the program level as a kind of refactoring tool, but also as part of compilers and runtime systems. Finally, since codata is quite related to objects in object-oriented programming, we hope that our approach can help to clarify their relation and design languages which subsume both traditional functional and object-oriented languages.

Acknowledgments. We would like to thank Tillmann Rendel and Julia Trieflinger for providing some early ideas for the design of what eventually became $GADT^T$. This work was supported by DFG project OS 293/3-1.

References

1. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: Proceedings of the Symposium on Principles of Programming Languages. pp. 27–38. ACM (2013)
2. Abel, A.M., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 185–196. ICFP '13, ACM, New York, NY, USA (2013)
3. Augustsson, L.: A compiler for lazy ml. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. pp. 218–227. LFP '84, ACM, New York, NY, USA (1984)
4. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theoretical Computer Science 39, 135–154 (1985)
5. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming 19(5), 509–543 (Sep 2009)
6. Chambers, C., Dean, J., Grove, D.: Whole-program optimization of object-oriented languages. University of Washington Seattle, Technical Report 96-06 2 (1996)
7. Chen, S., Erwig, M.: Principal type inference for gadts. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 416–428. POPL '16, ACM, New York, NY, USA (2016)
8. Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell University (2003)
9. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2017), http://adam.chlipala.net/cpdt/
10. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: Modular open classes and symmetric multiple dispatch for Java. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. pp. 130–145. ACM (2000)
11. Cook, W.R.: Object-oriented programming versus abstract data types. In: Proceedings of the REX Workshop / School on the Foundations of Object-Oriented Languages. pp. 151–178. Springer-Verlag (1990)
12. Cook, W.R.: On understanding data abstraction, revisited. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. pp. 557–572. ACM (2009)
13. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. 76(2-3), 95–120 (Feb 1988)
14. Curien, P.L., Herbelin, H.: The duality of computation. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 233–243. ICFP '00, ACM, New York, NY, USA (2000)
15. Danvy, O., Johannsen, J., Zerny, I.: A walk in the semantic park. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 1–12. PEPM '11, ACM, New York, NY, USA (2011)

16. Danvy, O., Millikin, K.: Refunctionalization at work. Science of Computer Programming 74(8), 534–549 (2009)
17. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the Conference on Principles and Practice of Declarative Programming. pp. 162–174 (2001)
18. Diehl, S., Hartel, P., Sestoft, P.: Abstract machines for programming language implementation. Future Generation Computer Systems 16(7), 739–751 (2000)
19. Downen, P., Ariola, Z.M.: The duality of construction. In: ESOP. pp. 249–269 (2014)
20. Emir, B., Kennedy, A., Russo, C., Yu, D.: Variance and generalized constraints for c^# generics. In: ECOOP. pp. 279–303. Springer (2006)
21. Ernst, M.D., Kaplan, C., Chambers, C.: Predicate dispatching: a unified theory of dispatch. In: Proceedings of the European Conference on Object-Oriented Programming. pp. 186–211. Springer LNCS 1445 (1998)
22. Erwig, M., Jones, S.P.: Pattern guards and transformational patterns. Electronic Notes in Theoretical Computer Science 41(1), 3 (2001)
23. Hagino, T.: Codatatypes in ml. Journal of Symbolic Computation 8(6), 629–650 (1989)
24. Hirzel, M., Nagpurkar, P.: Dualities in programming languages. In: Proc. of ACM Conference on Programming Language Design and Implementation, PLDI (2010)
25. Jacobs, B.: Objects and classes, coalgebraically. In: Object Orientation with Parallelism and Persistence, pp. 83–103. Springer-Verlag (1995)
26. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.P. (ed.) Functional Programming Languages and Computer Architecture: Nancy, France, September 16–19, 1985. pp. 190–203. Springer (1985)
27. Kennedy, A., Russo, C.V.: Generalized algebraic data types and object-oriented programming. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. pp. 21–40. ACM (2005)
28. Levy, P.B.: Call-by-push-value: A subsuming paradigm. In: TLCA. vol. 99, pp. 228–242. Springer (1999)
29. Oliveira, B.C., Moors, A., Odersky, M.: Type classes as objects and implicits. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 341–360. OOPSLA '10, ACM, New York, NY, USA (2010)
30. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming. pp. 50–61. ICFP '06, ACM, New York, NY, USA (2006)
31. Pierce, B.C.: Types and Programming Languages. Massachusetts Institute of Technology (2002)
32. Rendel, T., Trieflinger, J., Ostermann, K.: Automatic refunctionalization to a language with copattern matching: With applications to the expression problem. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 269–279. ICFP 2015, ACM, New York, NY, USA (2015)
33. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM annual conference. pp. 717–740. ACM (1972)
34. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to data abstraction. In: Schuman, S. (ed.) New Directions in Algorithmic Languages 1975. pp. 157–168. IFIP Working Group 2.1 on Algol, INRIA, Rocquencourt, France (1975)

35. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for gadts. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 341–352. ICFP '09, ACM, New York, NY, USA (2009)
36. Setzer, A., Abel, A., Pientka, B., Thibodeau, D.: Unnesting of copatterns. In: Dowek, G. (ed.) Proceedings of the Joint Conference on Rewriting Techniques and Applications and Typed Lambda Calculi and Applications. pp. 31–45. Springer LNCS 8560 (2014)
37. Thibodeau, D., Cave, A., Pientka, B.: Indexed codata types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 351–363. ICFP 2016, ACM, New York, NY, USA (2016)
38. Wadler, P.: The expression problem (Nov 1998), note to Java Genericity mailing list
39. Wadler, P.: Call-by-value is dual to call-by-name. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. pp. 189–201. ICFP '03, ACM, New York, NY, USA (2003)
40. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (Nov 1994)
41. Xi, H.X., Chiyan, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the Symposium on Principles of Programming Languages. pp. 224–235. ACM (2003)