Decomposition Diversity with Symmetric Data and Codata

DAVID BINDER, University of Tübingen, Germany
JULIAN JABS, University of Tübingen, Germany
INGO SKUPIN, University of Tübingen, Germany
KLAUS OSTERMANN, University of Tübingen, Germany

The expression problem describes a fundamental trade-off in program design: Should a program's primary decomposition be determined by the way its domain objects are constructed ("functional" decomposition), or by the way they are destructed ("object-oriented" decomposition)? We argue that programming languages should not force one of these decompositions on the programmer; rather, a programming language should support both ways of decomposing a program in a symmetric way, with an easy translation between these decompositions. However, current programming languages are usually not symmetric and hence make it unnecessarily hard to switch the decomposition.

We propose a language that is symmetric in this regard and allows a fully automatic translation between "functional" and "object-oriented" decomposition. We present a language with algebraic data types and pattern matching for "functional" decomposition and codata types and copattern matching for "object-oriented" decomposition, together with a bijective translation that turns a data type into a codata type ("destructorization") or vice versa ("constructorization"). We present the first symmetric programming language with support for local (co)pattern matching, which includes local anonymous function or object definitions, that allows an automatic translation as described above. We also present the first mechanical formalization of such a language and prove i) that the type system is sound, that the translations between data and codata types are ii) type-preserving, iii) behavior-preserving and iv) inverses of each other. We also extract a mechanically verified implementation from our formalization and have implemented an IDE with direct support for these translations.

CCS Concepts: • **Software and its engineering** → **Object oriented languages**; **Functional languages**; **Formal language definitions**; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: Types, Codata, Defunctionalization, Refunctionalization

ACM Reference Format:
David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. 2020. Decomposition Diversity with Symmetric Data and Codata. *Proc. ACM Program. Lang.* 4, POPL, Article 30 (January 2020), 28 pages. https://doi.org/10.1145/3371098

1 INTRODUCTION

Should you `fold` over the input or `unfold` over the output? Should a program be structured according to how its input is constructed or how its output is destructed? Should one use algebraic data types with pattern matching as available in functional languages or classes and methods as

Authors' addresses: David Binder, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, david.binder@uni-tuebingen.de; Julian Jabs, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, julian.jabs@uni-tuebingen.de; Ingo Skupin, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, skupin@informatik.uni-tuebingen.de; Klaus Ostermann, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, klaus.ostermann@uni-tuebingen.de.

2475-1421/2020/1-ART30
https://doi.org/10.1145/3371098

available in object-oriented languages? Should a program be extensible in its set of constructors or in its set of destructors?

For instance, the standard *map* functions on infinite streams can be written by pattern-matching on the constructors of the input (assuming that streams are defined as a data type with a `::` constructor):

```
map(f,x :: xs) = f(x) :: map(f,xs)
```

or it can be written by copattern-matching [Abel et al. 2013] on the destructors of the output (assuming that streams are defined as a codata type with a `head` and `tail` destructor):

```
map(f,s).head = f(s.head)
map(f,s).tail = map(f,s.tail)
```

In the constructor-centric first version, it is easy to add new consumers (pattern-matching functions) but hard to add new producers (constructors); in the destructor-centric version it is the other way around. This trade-off has been discussed in many forms [Cook 1990; Krishnamurthi et al. 1998; Reynolds 1975] and is today widely known as the *expression problem* [Wadler 1998], which has resulted in a long string of works on programming techniques and programming language design to support extensibility of both constructors and destructors.

In this work, we aim to analyze the problem on a more fundamental level. We want to understand the exact relation between the two different decompositions described by the expression problem. These decompositions essentially encapsulate a data- and codata-centric view, respectively. The goal of this work is to increase our understanding of the relation between those different decompositions and their impact on language design, programming, and language implementation. A promising first step in that direction is an analysis of two traditional global program transformations, defunctionalization [Danvy and Nielsen 2001; Reynolds 1972] and refunctionalization [Danvy and Millikin 2009]. These transformations change the modularity and extensiblity of a program (for instance, defunctionalization collects all function definitions in a program and arranges them into a single pattern match). Also, defunctionalization changes a destructor (function application) of a codata type (functions) - into constructors of a data type.

In this work, we propose generalizations of defunctionalization and refunctionalization, which we call *constructorization* and *destructorization*. They refer to the transformation of a constructor-centric data type definition and the program that uses it into a destructor-centric codata type definition together with a transformed program (destructorization), or vica versa (constructorization). Furthermore, we use the term *transposition* and the verb *transpose* to refer to either constructorization or destructorization.

We also present a programming language that is *symmetric* in its support for these decompositions in the sense that any program in constructor-centric form can be mechanically transformed into a unique destructor-centric program and vice versa, and that these transformations are total, type-preserving, behavior-preserving, and inverses of each other. We consider the features of our language, particularly the transformations, to be generally useful in practice, but we are focusing on their use to enable us to explore the relation between the different decompositions.

We believe that a programming language which is symmetric in this sense is relevant for programmers, language designers, and language implementers:

- It is relevant for *programmers* because non-symmetric languages encourage or force one of the decompositions. For instance, in Haskell 98 (without GADTs), constructors of an algebraic data type `data Exp a` always return an expression of type `Exp a`, but functions (=destructors) can have types like `Int -> Exp Int` for a type constructor `Exp`, which has led programmers to use (typed) Church encodings of data types instead of algebraic data types to profit from the more powerful type system on the destructor (in this case: function) side [Carette et al. 2007]. Object-oriented languages strongly encourage a destructor-based decomposition into objects; a constructor-based decomposition requires awkward designs such as the "visitor pattern" [Gamma et al. 1995]. For constructor-based design implemented with pattern matching, various features such as linear pattern matching or guards have been developed with no obvious counterpart in destructor-based designs, and vice versa. Even in languages with direct support for codata [Abel et al. 2013] (to be discussed in detail later in the related work section), symmetry is destroyed by intermingling codata types and function types. These asymmetries needlessly restrict the design choices of the programmer for purely technical reasons that have nothing to do with the problem domain. Another reason why symmetry is important to programmers is that it enables a new class of programming tools that is based on automatically translating between the decompositions, as we will illustrate with the prototypical IDE which we implemented for our language.

- It is relevant for *language designers* because the symmetry can be used to identify language design "holes" (features that are available for one decomposition but not the other). There is also a conceptual "two for the price of one" economy: By identifying a feature for the constructor side to be the exact counterpart to a feature on the destructor side, the design becomes simpler and meta-theoretic properties for one side may by construction carry over to the other side. We also believe that this work can clarify the relation between object-oriented languages and functional data type-oriented languages [Cook 2009].
- It is relevant for *language implementers* due to the possibility of using the transformations between the decompositions as a compilation technique and hence realizing two corresponding features with just one shared implementation. The transformation itself may also be a guide on how to implement cross-compilers between functional and object-oriented languages in a systematic way.

Concretely, this paper makes the following contributions:

- We present the first full symmetric programming language that allows invertible destructorization and constructorization.
- We have fully formalized the language in the Coq theorem prover and mechanically verified that the language is type-sound. We have implemented the transposition algorithms in Coq and have proven that they are total, preserve typing and behavior, and are inverses of each other. All "difficult" parts of the proofs have been mechanically verified in Coq, with a few rather obvious but very laborious to mechanize 'plumbing' proofs left as ordinary paper proofs.[1] The Coq proofs, which consumed by far the largest part of the authors' time investment, constitute more than 60 kloc.
- We synthesize a mechanically verified implementation of the language from our Coq formalization and have integrated it into a browser-based IDE that supports constructorization and destructorization.

Related work is discussed in detail later in the paper, but we want to discuss two related previous lines of work here. Rendel et al. [2015] presented an extension of defunctionalization and refunctionalization to arbitrary codata (not just functions). In Rendel et al.'s work, the transformations only work on a language that allows only top-level definitions such that programs can be arranged in a kind of matrix, which can then be transposed to flip the decomposition. Our transformation extends Rendel et al.'s algorithm for a "full" programming language that allows block structure/nesting/local definitions. Another line of work that bears a superficial similarity to this paper is work on compiling codata to data (and/or vice versa) [Downen et al. 2019; Laforgue and Régis-Gianas 2017]. We defer to the related work section for details, but for now we want

[1]The sources for the formalization and implementation were submitted as supplementary material.

to emphasize that these works aim for a *compositional* encoding of codata in terms of data (or vice versa) that therefore does not change the modularity or extensibility of the program, whereas we are interested in *global* transformations that switch the modular structure in the sense of the expression problem.

The remainder of this paper is structured as follows: In section 2, we present background on de- and refunctionalization and describe the language design issues that need to be addressed to turn these techniques into total transposition algorithms. In section 3 we present an informal overview of the solution to the problems discussed in section 2. Section 4 presents a case study to illustrate how the language works in terms of a useful and realistic example. We also use the case study to present the IDE which we have developed for this work. Section 5 contains the formalization of the language on which we base our development. The presentation of the transposition algorithms is contained in section 6. Section 7 presents the theorems that we have proven about our formalization. Section 8 discusses implications and future work, and section 9 presents related work. We conclude in section 10.

2 PROBLEM STATEMENT

Constructorization and destructorization are extensions and generalizations of defunctionalization [Danvy and Nielsen 2001; Reynolds 1972] and refunctionalization [Danvy and Millikin 2009], respectively. These traditional whole-program transformations turn programs with higher-order functions (that is, with function application as destructors of functions) into first-order programs with constructors of algebraic data types and pattern matching (defunctionalization) or back (refunctionalization) and as such are a useful step towards constructorization and destructorization. In this section, we revisit these transformations and describe the problems in turning them into transpositions of a full-fledged programming language.

To illustrate defunctionalization, consider this program in Haskell-like syntax which maps two anonymous functions over a list.

```
map :: (Int -> Int) -> [Int] -> [Int]
map f xs = ... f (head xs) ...

let x=7
    y=12
in map (\z.z+x) (map (\z.z*y) [1,2,3])
```

The traditional way to defunctionalize a function type is to first turn all local function declarations of that type into top-level definitions by lambda lifting [Johnsson 1985]. The values for the free variables in the function bodies are passed as parameters to these functions. Top-level declarations need a name, hence we need to synthesize/invent two new names in our example, `plus` and `mult`.

```
plus x = \z.z+x
mult y = \z.z*y

let x=7
    y=12
in map (plus x) (map (mult y) [1,2,3])
```

The next step is to create an algebraic data type with one constructor for each top-level function, whereby each constructor has a parameter for the free variable in the original function body. A special *apply* function is created, which pattern-matches on the synthesized algebraic data type to determine the correct function body for that call and to get access to the values that would have otherwise been stored in the closure. Function definitions are replaced by invocations of the matching synthesized constructor, and function applications are replaced by invocations of the first-order *apply* function.

```
data Int2Int = Plus Int | Mult Int

apply :: Int2Int -> Int -> Int
apply (Plus x) z = z+x
apply (Mult x) z = z*y

map :: Int2Int -> [Int] -> [Int]
map f xs = ... apply f (head xs) ...

let x=7
    y=12
in map (Plus x) (map (Mult y) [1,2,3])
```

Refunctionalization tries to turn first-order data types back into functions, i.e., reverse the process of defunctionalization, but the attempt to make it a total function and the inverse of defunctionalization fails for several reasons:

a) It is only a partial function in traditional functional languages because it is not clear what to do if there is more than one pattern match on the argument type of the function type to be refunctionalized [Danvy and Millikin 2009]. In our example, imagine a second function pattern matching on `Int2Int`, such as

   ```
   isPlus :: Int2Int -> Bool
   isPlus (Plus _) = True
   isPlus (Mult _) = False
   ```

   This extended program can no longer be refunctionalized.

b) Lambda-lifting requires the synthesis of new names not in the original program, which then show up as constructor names in the refunctionalized program, and it is not obvious how to make that process invertible. In our example, we had to invent the names `plus` and `mult`. If we would refunctionalize and then defunctionalize, it is not clear how to guarantee that we get the same names back.

c) It is not clear how to "undo" the lambda-lifting because the defunctionalized program contains no information about which functions ought to be de-lambda-lifted. In our example, it is not possible to reconstruct from the defunctionalized program whether `plus` and `mult` were originally defined locally or as top-level functions.

d) If the `apply` function is changed such that the top-level operation is no longer a pattern match on its first argument, it is not clear how to deal with the function body when refunctionalizing the program.

e) Finally, if the arguments of the newly generated constructors are changed to be not just variable names but general expressions, it is not clear how to preserve the evaluation order when refunctionalizing the program. For instance, if we change `(Plus x)` to `Plus (x + 1)` in the invocation of `map` above, a naive refunctionalization would refunctionalize the first argument of the first `map` call to `(\z.z+(x+1))`, thereby changing the order of evaluation by moving the addition inside a λ-abstraction.[2]

Previous work by Rendel et al. [2015] has addressed problem a) by generalizing function types to general codata types with copattern matching [Abel et al. 2013]. Function types are a special case of codata types with a single *apply* destructor. The case of multiple pattern matches on algebraic data types can be solved with codata types by synthesizing one destructor per pattern match; something that was not possible with traditional functions due to the inherent limitation of functions having only one destructor. However, Rendel et al. [2015] left open a solution to problems b)-e): their proposal assumed a language in which all definitions and (co)pattern matches were on the level of

[2]We assume a call-by-value language in the remainder of this paper.

top-level functions only, that is, no local definitions are possible. In the next section, we review how Rendel et al. have addressed a) and describe our novel solutions for b)-e), which together enable fully invertible transposition for a full functional language with local pattern matching and copattern matching (including λ-abstraction).

3 OVERVIEW

We now give an informal overview of how we addressed problems a) to e) as outlined in the previous section. We address

a) by generalizing functions to codata
b) by adding names to matches and comatches
c) by distinguishing local and global constructor and destructor names
d) by adding consumer and generator functions
e) by adding let-like functionality to match/comatch

The next subsections describe these ideas in detail. Since all features we describe apply dually to both the data and codata features of the language, we use the prefix x to abstract over which side of the duality we refer to by replacing a concrete (possibly empty) prefix by x. For instance, an *x*tor is a *construc*tor or a *destruc*tor, *x*pattern matching is pattern matching or *co*pattern matching, and so forth.

We propose a language with a symmetric data and codata types with (co)pattern matching, together with *first-order* functions.

3.1 Generalizing Functions to Codata

While most functional languages support data types and function types, we support data types and codata types, which are strictly more general than function types. This means that even though we do not mention function types, lambda abstraction, and the application of a lambda-defined function to an argument explicitly in the formalization in section 5, these could be provided by desugarings into the more fundamental codata type declarations, copattern matches and destructor applications, respectively.

Functions are the special case of codata with just one destructor (typically called `apply`). In our example, we can define such a codata type for functions from `Int` to `Int` as follows:

```
codata Int2Int where
  apply(Int): Int
```

This works similarly to how function types are provided in Java, where lambda abstractions are objects which implement the `Function<T,R>` interface, which provides the `R apply(T t)` method.

Codata types are instantiated by copattern matching, and destructors can be called on values of codata types using dot notation. The following example shows how to apply the function `f` on its argument `head xs` by calling the destructor `apply` on `f`, and a copattern match that mimics the λ-abstraction `\z.z+x`.

```
map :: Int2Int -> [Int] -> [Int]
map f xs = ... f.apply(head xs) ...
...
in map (comatch Int2Int with
          apply(z) => z+x) ...
```

If we consider the example of the additional `isPlus` function from the previous section, destructorization can now be restored by adding an additional destructor to the codata type and corresponding destructor implementations in the copattern matches for that codata type. The result of destructorizing `Int2Int` after adding the `isPlus` function to the constructorized program from the introduction looks as follows:

```
codata Int2Int
  apply(Int): Int
  isPlus() : Bool

let x=7
    y=12
in map (comatch Int2Int with
          apply(z) => z + x
          isPlus() => true)
       (map (comatch Int2Int with
               apply(z) => z * y
               isPlus() => false)
             [1,2,3])
```

3.2 Adding Names to Matches/Comatches

We solve the problem of synthesizing names for new constructors/destructors by requiring programmers to give a unique name to each (co)pattern match in the program. For instance, here we give the name `Plus` to the first comatch. This name is then turned into the constructor name `Plus`. In order to visually separate the name of the comatch from the codata type on which we comatch, we use the keyword **on**.

```
... in map (comatch Plus on Int2Int with
              apply(z) => z+x ) ...
```

Strictly speaking, this step is not required when writing code, since these names are only required when transposing the program. Instead, it would suffice to generate them on the fly during the transformation process by either prompting the user or having a generator for fresh names. However, names specified in the program text have the distinct advantage that these names will be turned into constructor or destructor names, respectively; autogenerated names decrease the readability of the generated program.

3.3 Local and Global Xtor Names

When destructorizing a data type, it is not obvious whether a constructor invocation should be turned into an inlined copattern match (which would lead to code duplication if the same constructor is invoked multiple times) or whether it should be turned into a function call of a function that does the copattern match (which would avoid code duplication if the same constructor is invoked multiple times). Dually, the same problem arises for constructorization and destructor invocations. To keep destructorization and constructorization total and inverses of each other, we distinguish *local* names (denoted by names that start with an underscore _) from *global* names (names that do not start with underscore). Names of generator and consumer functions (introduced in the next subsection) will always be global, while names of matches and comatches will always be local.

Local xtors can only be invoked in one place in the program; transposing the corresponding xdata type leads to an inlined xmatch of the opposite polarity. Conversely, global xtors can be invoked in many places in the program; transposing the corresponding xdata type yields a top-level first-order function definition containing the xmatch, which is called in all places that used to invoke the xtor. All xtors that result from local xpattern matches are local, thereby guaranteeing that a transposition roundtrip will again yield the same program.

In our running example, the `Plus` and `Mult` constructors are local constructors.

```
data Int2Int where
  _Plus(Int)
  _Mult(Int)
```

Let us consider an extension of the data type with another constructor that is global (in this case for the identity function):

```
data Int2Int where
  _Plus(Int)
  _Mult(Int)
  Identity()
...
apply (Identity()) z = z
```

If we destructorize `Int2Int`, then the invocations of `_Plus` and `_Mult` are turned into the copattern matches we started with. Since the `Identity` constructor is global, a top-level function

```
Identity = comatch Identity on Int2Int with
             apply(z) => z
```

is generated and this function is invoked in all places that invoked the `Identity` constructor.

3.4 Consumer and Generator Functions

As we have seen in the previous subsection, global xtors are turned into top-level functions containing an xpattern match. However, in the next round of transposition, how can we know which top-level functions must be turned back into a global xtor? And what do we do about top-level functions that do not contain a top-level xpattern match in their body?

We solve both problems by partitioning functions into three different kinds:

- Ordinary functions are not affected by transposition (except that their bodies are transposed).
- *Consumer functions* are syntactically restricted to contain a top-level pattern match on their first argument. Destructorization turns consumer functions into global destructors; the cases of the pattern match are distributed to the corresponding copattern matches.
- *Generator functions* are syntactically restricted to contain a top-level copattern match. Constructorization turns generator functions into global constructors; the cases of the copattern match are distributed to the corresponding pattern matches.

All three kinds of functions are not first-class, i.e. they cannot be passed as an argument or returned as a value. In our running example, `apply` is a global destructor of `Int2Int`, hence constructorization turns `apply` into a consumer function (denoted by the keyword **`cfun`**).

```
cfun Int2Int: apply(z : Int) : Int :=
  Plus(x) => z+x
  Mult(x) => z*x
```

Transposing `Int2Int` once more turns the consumer function `apply` back into a global destructor, as intended.

In a similar fashion, the additional `Identity` constructor from subsection 3.3 would now be destructorized into a generator function (denoted by the keyword **`gfun`**).

```
gfun Identity() : Int2Int :=
  apply(z) => z
```

Regarding ordinary functions without a top-level xpattern match, in our language we keep them as a separate construct because we consider it to be notationally and conceptually convenient to distinguish them from consumer and generator functions. However, for completeness sake we want to point out that they can be easily desugared using a `Unit` data type (with a single no-argument constructor, as usual): An ordinary function with body e becomes a consumer function of type `Unit` with e in the single pattern match case (a similar desugaring to generator functions would also be possible).

3.5 Let-like Functionality for Match/Comatch

Let us now reconsider the example from the introduction, changing `(Plus x)` to `Plus (x + 1)` in the invocation of `map`. In our CBV language, `x + 1` is evaluated when the constructor call is evaluated. However, if we were to destructorize the constructor invocation to **comatch** `plus` **on** `Int2Int {apply(z)=> z+(x+1)}`, then the `x + 1` would be evaluated only when the `apply` destructor is invoked.

We solve this problem by extending both the pattern match and the copattern match construct with a name binding construct similar to an enclosing **let** binding. These bindings are evaluated when the xpattern match itself is evaluated, which restores the desired evaluation order.

In our example, we add a corresponding binding to the comatch:

```
...comatch Plus on Int2Int using x:=x+1 with
      apply(z) => z+x
```

Adding these annotations to xmatches instead of simply using **let**-bindings around them corresponds to the idea that xmatches are essentially a type of local **gfun** of **cfun** and thus should be thought of as *closures*. Furthermore, since they need to be closed in order to apply transpositions on them, this removes the need to search for all relevant **let**s around them which are required for this precondition to hold.

4 CASE STUDY

The symmetric design of our language gives programmers the possibility to view the domain they are modelling from different angles, depending on the decompositions that are chosen for the domain objects occurring in the program. Being able to switch decompositions makes it more convenient to change or add functionality, and gives new insights into the structure of the program. In order to illustrate this point, we present a case study which is inspired by Danvy et al. [2011], who inter-derive reduction-based and reduction-free negational normalization functions. The original case study used de- and refunctionalization at several places to change the perspective on the program, which was done manually. By contrast, we can leverage the symmetric design of our language to perform corresponding changes (i.e. transposition) mechanically. Thus, this case study focuses on parts of the original case study by Danvy et al. [2011] which hinged on the use of de- and refunctionalization. The goal is to obtain a program which computes the negation normal form of a boolean formula with conjunction, disjunction and negation by repeatedly searching for a redex of the form $\neg(\phi \wedge \psi)$, $\neg(\phi \vee \psi)$ or $\neg\neg\phi$ and replacing it by $\neg\phi \vee \neg\psi$, $\neg\phi \wedge \neg\psi$ and ϕ, respectively.[3] We will semi-mechanically derive this program iteratively after first manually writing a program which searches for one such redex. After (mechanical) constructorization, it only requires small modifications to adapt this code into a reduction-based evaluator which evaluates a boolean formula to its negation normal form. This approach highlights how the development of the final solution was greatly simplified by switching our *view* on the program by changing to a different decomposition, which was aided by the use of a mechanical transformation.

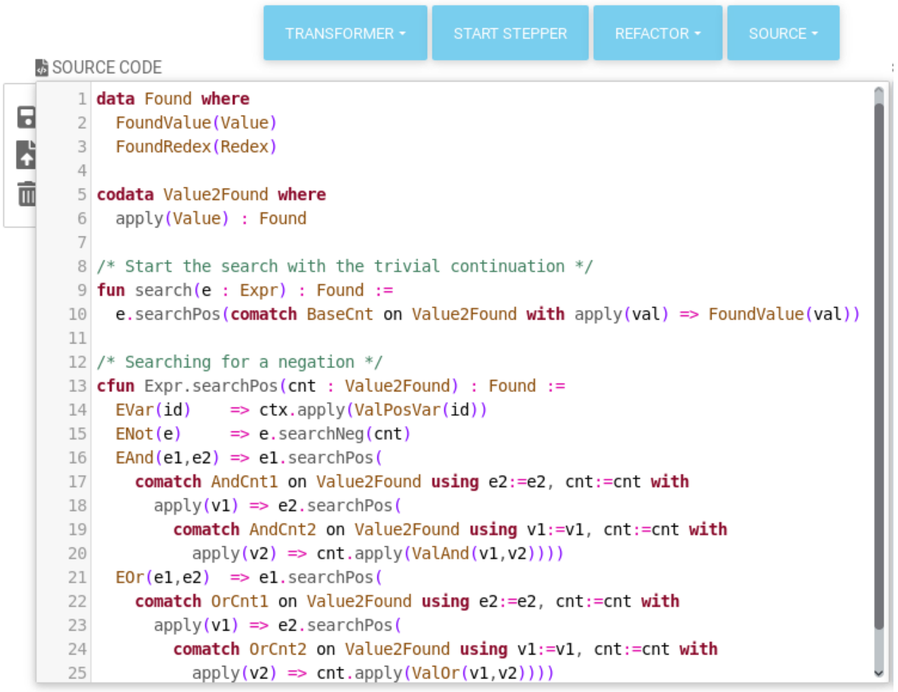
Our representation of the domain is given in Figure 1 and will not change during the subsequent development. The rest of the code, which is subject to the transformation, is given in Figure 3. As a first step, we write the functions `search`,[4] `searchPos` and `searchNeg`, which search for the leftmost outermost redex in an expression. The function `search` starts the search by calling `searchPos` with the initial continuation; `searchPos` searches for the first negation, recursively building up a continuation along the way, and passes the computation to `searchNeg` after the first negation has been encountered. The `Found` data type represents the result of searching for a redex in an expression.

[3] A simpler, "big-step" style implementation of this problem is of course possible and also described by Danvy; here we focus on the "small-step" reduction-style solution because it is well-suited to illustrate the features of our language.

[4] The `search` function is actually the result of CPS-transforming and then simplifying a direct-style function; the simplification amounts to only applying a continuation when a value is found.

```
data Expr where                 cfun Value:asExpr() : Expr :=
    EVar(Identifier)              ValPosVar(id)  => EVar(id)
    ENot(Expr)                    ValNegVar(id)  => ENot(EVar(id))
    EAnd(Expr, Expr)              ValAnd(e1,e2)  => EAnd(e1.asExpr(), e2.asExpr())
    EOr(Expr, Expr)               ValOr(e1,e2)   => EOr(e1.asExpr(), e2.asExpr())

data Redex where                cfun Redex:eval() : Expr :=
    RedNot(Expr)                  RedNot(e)      => e
    RedAnd(Expr, Expr)            RedAnd(e1,e2)  => EOr(ENot(e1), ENot(e2))
    RedOr(Expr, Expr)             RedOr(e1,e2)   => EAnd(ENot(e1), ENot(e2))

data Value where
    ValPosVar(Identifier)
    ValNegVar(Identifier)
    ValAnd(Value, Value)
    ValOr(Value, Value)
```

Fig. 1. The definitions of expressions, redexes, values, the embedding of values in expressions, and the reduction of immediate redexes will not change during the subsequent steps.



```
data Found where
  FoundValue(Value)
  FoundRedex(Redex)

codata Value2Found where
  apply(Value) : Found

/* Start the search with the trivial continuation */
fun search(e : Expr) : Found :=
  e.searchPos(comatch BaseCnt on Value2Found with apply(val) => FoundValue(val))

/* Searching for a negation */
cfun Expr.searchPos(cnt : Value2Found) : Found :=
  EVar(id)     => ctx.apply(ValPosVar(id))
  ENot(e)      => e.searchNeg(cnt)
  EAnd(e1,e2) => e1.searchPos(
    comatch AndCnt1 on Value2Found using e2:=e2, cnt:=cnt with
      apply(v1) => e2.searchPos(
        comatch AndCnt2 on Value2Found using v1:=v1, cnt:=cnt with
          apply(v2) => cnt.apply(ValAnd(v1,v2))))
  EOr(e1,e2)  => e1.searchPos(
    comatch OrCnt1 on Value2Found using e2:=e2, cnt:=cnt with
      apply(v1) => e2.searchPos(
        comatch OrCnt2 on Value2Found using v1:=v1, cnt:=cnt with
          apply(v2) => cnt.apply(ValOr(v1,v2))))
```

Fig. 2. The IDE provides functionality to automatically switch the decomposition.

Following the approach of Danvy et al. [2011], we constructorize the codata type `Value2Found`, since it is known that applying defunctionalization (which we generalize to constructorization) results in interesting semantic artifacts, bringing us closer to an abstract machine representation. This results in the transformed program in Figure 4.

Under this decomposition we realize that `Context` is an appropriate name for the new data type. For example, the term `_OrCnt1(e,_AndCnt2(v,_BaseCnt()))` corresponds to the evaluation

```
data Found where                          codata Value2Found where
  FoundValue(Value)                         apply(Value) : Found
  FoundRedex(Redex)

/* Start the search with the trivial continuation */
fun search(e : Expr) : Found :=
  e.searchPos(comatch BaseCnt on Value2Found with apply(val) => FoundValue(val))

/* Searching for a negation */
cfun Expr:searchPos(cnt : Value2Found) : Found :=
  EVar(id)     => cnt.apply(ValPosVar(id))
  ENot(e)      => e.searchNeg(cnt)
  EAnd(e1,e2)  => e1.searchPos(
      comatch AndCnt1 on Value2Found using e2:=e2, cnt:=cnt with
        apply(v1) => e2.searchPos(
            comatch AndCnt2 on Value2Found using v1:=v1, cnt:=cnt with
              apply(v2) => cnt.apply(ValAnd(v1, v2))))
  EOr(e1,e2)   => e1.searchPos(
      comatch OrCnt1 on Value2Found using e2:=e2, cnt:=cnt with
        apply(v1) => e2.searchPos(
            comatch OrCnt2 on Value2Found using v1:=v1, cnt:=cnt with
              apply(v2) => cnt.apply(ValOr(v1, v2))))

/* Searching a redex under a negation */
cfun Expr:searchNeg(cnt : Value2Found) : Found :=
  EVar(id)     => cnt.apply(ValNegVar(id))
  ENot(e)      => FoundRedex(RedNot(e))
  EAnd(e1,e2)  => FoundRedex(RedAnd(e1,e2))
  EOr(e1,e2)   => FoundRedex(RedOr(e1,e2))
```

Fig. 3. The main part of the code before constructorization.

context $v \land (\Box \lor e)$, where v already is a value but e might contain further redexes (note that `Context`s compose from the inside outwards, similarly to a stack). We therefore rename the data type and its constructors, as seen in Figure 5. The `apply` function takes a context, and returns the next redex if a value is plugged into the hole, we therefore rename it to `findNext`. We also extend the definition of the constructor `FoundRedex` to also return the enclosing context of the redex, and modify the `searchNeg` function accordingly.

In order to evaluate an expression to normal form we need one additional function which substitutes an expression (in our case, the result of reducing a redex) into an evaluation context, as seen in Figure 6. Now it is easy to give the definition of the evaluation function.

Bringing the data type `Context` back into destructor form results in the program shown in Figure 7, which we might not have originally written, since it corresponds to the addition of a destructor to an existing codata type. Adding an additional cfun to the constructorized version was easy.

Since the transpositions can be performed mechanically instead of doing them by hand, as it was done in the original case study, this development can easily be recreated in an IDE which provides the necessary capabilities, as showcased in Figure 2. Such an IDE was created in the authors' lab. Most prominently, the IDE provides switching the decomposition by the touch of a button, as well as the following features:

- (Partial) type inference for the language.
- Syntactic sugar for codata types corresponding to functions, and the corresponding lambda abstractions.
- Moving local matches / comatches to the toplevel.

```
data Value2Found where          cfun Value2Found:apply(v : Value) : Found :=
  _BaseCnt()                      _BaseCnt()        => FoundValue(v)
  _AndCnt1(Expr, Value2Found)     _AndCnt1(e,cnt)   => e.searchPos(_AndCnt2(v,cnt))
  _AndCnt2(Value, Value2Found)    _AndCnt2(v',cnt)  => cnt.apply(ValAnd(v',v))
  _OrCnt1(Expr, Value2Found)      _OrCnt1(e,cnt)    => e.searchPos(_OrCnt2(v,cnt))
  _OrCnt2(Value, Value2Found)     _OrCnt2(v', cnt)  => cnt.apply(ValOr(v',v))

                fun search(e : Expr) : Found :=
                  e.searchPos(_BaseCnt())

                cfun Expr:searchPos(cnt : Value2Found) : Found :=
                  EVar(id)     => cnt.apply(ValPosVar(id))
                  ENot(e)      => e.searchNeg(cnt)
                  EAnd(e1,e2)  => e1.searchPos(_AndCnt1(e2,cnt))
                  EOr(e1,e2)   => e1.searchPos(_OrCnt2(e2,cnt))

                cfun Expr:searchNeg(cnt : Value2Found) : Found :=
                  EVar(id)     => cnt.apply(ValNegVar(id))
                  ENot(e)      => FoundRedex(RedNot(e))
                  EAnd(e1,e2)  => FoundRedex(RedAnd(e1,e2))
                  EOr(e1,e2)   => FoundRedex(RedOr(e1,e2))
```

Fig. 4. The main part of the code after constructorization.

- Adding a constructor / destructor to a data type / codata type by completing a template generated from the switched decomposition, see Figure 8.
- Changing the signature of an existing constructor or destructor, by changing a similar template.

The fact that our language is designed with these automatic transformations in mind enables rapid experimentation and prototyping, instead of requiring a manual, and error-prone, transformation done on paper. Since we also generalize the transformations Danvy et al. used, we also implicitly extend their approach to interderiving semantic artifacts. For instance, the addition and transformation of the `substitute` function in our case study would not have been possible in their approach due to their usage of traditional refunctionalization.

5 FORMALIZATION

In this section, we present the language on which we define the transposition algorithms.

5.1 Syntactic Conventions

We use the usual convention that a horizontal bar over a syntactic category or metavariable indicates a (possibly empty) list of occurrences of its argument, hence $\overline{x}$ is shorthand notation for $x_1, \ldots, x_{|\overline{x}|}$. As usual, we abuse syntax and let list notation for composite expressions, such as $\overline{e : T}$, stand for $e_1 : T_1, \ldots, e_n : T_n$ (where $n = |\overline{e : T}|$). We also use list notation for the repeated application/conjunction of a judgement over a list. For instance, a typing judgement $\Gamma \vdash \overline{e : T}$ stands for the sequence of judgements $\Gamma \vdash e_1 : T_1 \ldots \Gamma \vdash e_n : T_n$.

5.2 Syntax

The syntax of programs is given in Figure 9. A full program consists of a set of data type declarations, a set of codata type declarations, and a set of function declarations, which can be ordinary functions, consumer functions, or generator functions. Staying close to the representation that we chose in our Coq formalization, we represent the binding structure of expressions using de Bruijn indices. Hence, variables are just natural numbers indicating the distance to the occurrence of the binding

```
data Context where                      data Found where
   _EmptyCtx()                             FoundValue(Value)
   _AndCtx1(Expr, Context)                 FoundRedex(Redex, Context)
   _AndCtx2(Value, Context)
   _OrCtx1(Expr, Context)
   _OrCtx2(Value, Context)

cfun Expr:searchNeg(ctx : Context) : Found :=
   EVar(id)     => ctx.findNext(ValNegVar(id))
   ENot(e)      => FoundRedex(RedNot(e), ctx)
   EAnd(e1,e2)  => FoundRedex(RedAnd(e1,e2), ctx)
   EOr(e1,e2)   => FoundRedex(RedOr(e1,e2), ctx)
```

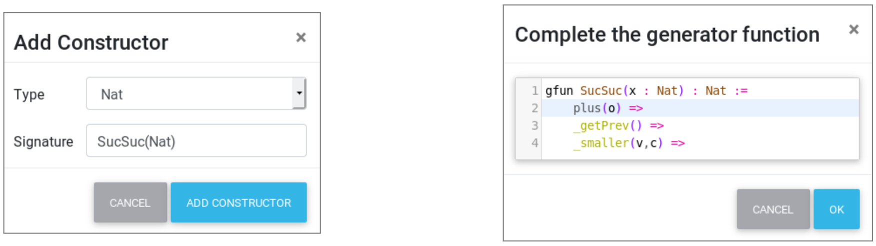
Fig. 5. The changed part of the code after renaming `Value2Found` to `Context` and `apply` to `findNext` and modifying `FoundRedex`.

```
cfun Context:substitute(e : Expr) : Expr :=
  _EmptyCtx() => e
  _AndCtx1(e', ctx) => ctx.substitute(EAnd(e,e'))
  _AndCtx2(v, ctx)  => ctx.substitute(EAnd(v.asExpr(), e))
  _OrCtx1(e', ctx)  => ctx.substitute(EOr(e, e'))
  _OrCtx2(v, ctx)   => ctx.substitute(EOr(v.asExpr(), e))

fun evaluate(e : Expr) : Value :=
  match _ on search(e) with
    FoundValue(v) => v
    FoundRedex(r,ctx) => evaluate(ctx.substitute(r.eval()))
```

Fig. 6. The new functions added after changing names

```
codata Context where
   findNext(Value) : Found
   substitute(Expr) : Expr

cfun Expr:searchPos(ctx : Context) : Found :=
   EVar(id)     => ctx.findNext(ValPosVar(id))
   ENot(e)      => e.searchNeg(ctx)
   EAnd(e1,e2) => e1.searchPos(
      comatch AndCtx1 on Context using e2:=e2, ctx:=ctx with
         findNext(v1) => e2.searchPos(
            comatch AndCtx2 on Context using v1:=v1, ctx:=ctx with
               findNext(v2) => ctx.findNext(ValAnd(v1, v2))
               substitute(ex) => ctx.substitute(EAnd(v1.asExpr(), ex)))
         substitute(ex) => ctx.substitute(EAnd(ex, e2)))
   EOr(e1,e2)  => e1.searchPos(
      comatch OrCtx1 on Context using e2:=e2, ctx:=ctx with
         findNext(v1) => e2.searchPos(
            comatch OrCtx2 on Context using v1:=v1, ctx:=ctx with
               findNext(v2) => ctx.findNext(ValOr(v1, v2))
               substitute(ex) => ctx.substitute(EOr(v1.asExpr(), ex)))
         substitute(ex) => ctx.substitute(EOr(ex, e2)))
```

Fig. 7. The final result of our case study



Add Constructor

Type: Nat

Signature: SucSuc(Nat)

Complete the generator function

```
gfun SucSuc(x : Nat) : Nat :=
    plus(o) =>
    _getPrev() =>
    _smaller(v,c) =>
```

Fig. 8. Adding a constructor to a data type by completing a generator function template in the other decomposition.

$$
\begin{array}{lcll}
P & ::= & \overline{D}, \overline{E}, \overline{F}, \overline{G}, \overline{H} & \textit{Program} \\
D & ::= & \textbf{data}\ T\ \textbf{where}\ \overline{Ctor} & \textit{Data type} \\
Ctor & ::= & C(\overline{T}) & \textit{Constructor} \\
E & ::= & \textbf{codata}\ T\ \textbf{where}\ \overline{Dtor} & \textit{Codata type} \\
Dtor & ::= & d(\overline{T}) : T & \textit{Destructor} \\
F & ::= & \textbf{fun}\ f(\overline{T}) : T := e & \textit{Function} \\
G & ::= & \textbf{cfun}\ T{:}d(\overline{T}) : T := \overline{C \Rightarrow e} & \textit{Consumer Function} \\
H & ::= & \textbf{gfun}\ C(\overline{T}) : T := \overline{d \Rightarrow e} & \textit{Generator Function} \\
e & ::= & x & \textit{Variable (de Bruijn index)} \\
 & | & C(\overline{e}) & \textit{Constructor and gfun calls} \\
 & | & e.d(\overline{e}) & \textit{Destructor and cfun calls} \\
 & | & f(\overline{e}) & \textit{Function call} \\
 & | & \textbf{match}\ d\ \textbf{on}\ e\ \textbf{using}\ \overline{e}\ \textbf{with}\ \overline{C \Rightarrow e} & \textit{Pattern match} \\
 & | & \textbf{comatch}\ C\ \textbf{on}\ T\ \textbf{using}\ \overline{e}\ \textbf{with}\ \overline{d \Rightarrow e} & \textit{Copattern match} \\
 & | & \textbf{let}\ e\ \textbf{in}\ e & \textit{Let expression} \\
v & ::= & C(\overline{v}) & \textit{Constructor and gfun calls} \\
 & | & \textbf{comatch}\ C\ \textbf{on}\ T\ \textbf{using}\ \overline{v}\ \textbf{with}\ \overline{d \Rightarrow e} & \textit{Copattern match} \\
T & ::= & \textit{type names} & \\
C & ::= & \textit{constructor, gfun or comatch names (local or global)} & \\
f & ::= & \textit{function names} & \\
d & ::= & \textit{destructor, cfun or match names (local or global)} &
\end{array}
$$

Fig. 9. Formal syntax

construct. For example, the expression **let** $x := e$ **in** (**let** $y := e'$ **in** $f(x, y)$) is represented in our formalization as **let** e **in** (**let** e' **in** $f(1, 0)$). For the same reason, the bindings in `using` clauses are just a list of expressions. In our example programs, we will continue to use ordinary variable names instead of de Bruijn indices to improve readability, at the cost of a small deviation between the examples and the formal syntax. Also due to de Bruijn variables, the cases and cocases of (co)pattern matches list only the name of the xtor to match on instead of binding names for the arguments of the xtor.

We use the convention that type names, constructor names, and generator function names start with uppercase letters, whereas function names, destructor names, and consumer function names

*		$\textsc{Dt}$	Data type names defined in the program
*		$\textsc{CoDt}$	Codata type names defined in the program
*	$\forall\, T \in \textsc{Dt}:$	$\textsc{Ctor}(T)$	Constructors of type T
*	$\forall\, T \in \textsc{Dt}:$	$\textsc{Cfun}(T)$	Consumer functions for type T
*	$\forall\, T \in \textsc{CoDt}:$	$\textsc{Dtor}(T)$	Destructors of type T
*	$\forall\, T \in \textsc{CoDt}:$	$\textsc{Gfun}(T)$	Generator functions for type T
*		$\textsc{Fun}$	(Ordinary) function signatures
	$\forall\, f \in \textsc{Fun}:$	$\textsc{Body}(f)$	Body of function f
	$\forall\, (\_{:}d(\_) : \_) \in \textsc{Cfun}(-):$	$\textsc{Cases}(d)$	Body of consumer function d
	$\forall\, (C(\_) : \_) \in \textsc{Gfun}(-):$	$\textsc{Cocases}(C)$	Body of generator function C

Fig. 10. Global sets to query the program. Typechecking expressions depends on the starred sets only, that is, only on declarations.

start with lowercase letters. Names in C and d may be prepended with an underscore to denote local names as described in subsection 3.3.

To avoid cluttering the definitions, we assume the program to be a global constant. In the remaining definitions, we query the global program via the sets defined in Figure 10. The sets are mostly self-explanatory, hence we refer to the Coq code for a formal definition and instead suggest to consider the example in Figure 12, whose representation in terms of these set functions can be seen in Figure 13, as illustration. The only noteworthy aspect of these sets is that $\textsc{Gfun}(T)$ and $\textsc{Ctor}(T)$ have been set up in such a way that they have the same codomain, such that we can form the set union $\textsc{Gfun}(T) \cup \textsc{Ctor}(T)$. The same holds for $\textsc{Cfun}(T)$ and $\textsc{Dtor}(T)$.

Together, the first seven items of the bottom half of Figure 10 contain all the static information of a program which is necessary to typecheck expressions.

5.3 Typing and Reduction

Typechecking of expressions is defined in Figure 11. Expressions are typechecked in the context of all function signatures, meaning that arbitrary recursion, including non-termination, between functions is possible.[5] While the rules for let bindings, xtors and the different kinds of function calls are pretty standard, the rules for xmatches are slightly more involved. Firstly, the rule for matches is set up to ensure that every constructor occurs in exactly one case of the match. While it would be possible to allow non-exhaustive pattern matches, we have restricted ourselves to exhaustive pattern matches for the sake of simplicity.[6] Secondly, it is important to note that the bodies inside the case clauses are typechecked using only the variables bound by the binding lists and the variables provided by the respective case, i.e. it ensures that matches are closed terms. These remarks apply equally to comatches.

Typechecking a full program involves typechecking of function bodies in the context of the types of the arguments, as usual. Typechecking consumer and generator functions is a straightforward extension of typechecking local pattern (copattern) matches. Program well-formedness furthermore involves entirely unsurprising checks that all type names that are used are actually defined, that generator and consumer functions have a branch for each xtor of the corresponding (co)data type

[5] Inductive and coinductive types are often used in conjunction with termination/productivity checks (e.g. Atkey and McBride [2013]), but for our purposes those checks are orthogonal to this work.

[6]Destructorization, for example, would translate an exception arising from an incomplete pattern match into the invocation of a destructor on a copattern match with a missing cocase. Semantic preservation under transposition might therefore still be possible.

$\boxed{\text{Expression Typing: } \Gamma \vdash e : T}$

$$\frac{\mathbf{lookup}(x, \Gamma) = T}{\Gamma \vdash x : T} \quad (\text{T-Var})$$

$$\frac{C(\overline{T'}) \in \textsc{Ctor}(T) \cup \textsc{Gfun}(T) \qquad \Gamma \vdash \overline{e : T'}}{\Gamma \vdash C(\overline{e}) : T} \quad (\text{T-Ctor/Gfun})$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, T_1 \vdash e_2 : T_2}{\Gamma \vdash \mathbf{let}\ e_1\ \mathbf{in}\ e_2 : T_2} \quad (\text{T-Let})$$

$$\frac{d(\overline{T'}) : T'' \in \textsc{Dtor}(T) \cup \textsc{Cfun}(T) \qquad \Gamma \vdash e : T \quad \Gamma \vdash \overline{e' : T'}}{\Gamma \vdash e.d(\overline{e'}) : T''} \quad (\text{T-Dtor/Cfun})$$

$$\frac{f(\overline{T}) : T' \in \textsc{Fun} \qquad \Gamma \vdash \overline{e : T}}{\Gamma \vdash f(\overline{e}) : T'} \quad (\text{T-Fun})$$

$$\frac{\Gamma \vdash \overline{e : T'''} \qquad \Gamma \vdash e' : T \qquad T \in \textsc{Dt} \qquad \forall\ (C(\overline{T'}) \in \textsc{Ctor}(T)).\ \exists i.\ C = C_i\ \wedge \overline{T'''}, \overline{T'} \vdash e''_i : T''}{\Gamma \vdash \mathbf{match}\ d\ \mathbf{on}\ e'\ \mathbf{using}\ \overline{e}\ \mathbf{with}\ \overline{C \Rightarrow e''} : T''} \quad (\text{T-Match})$$

$$\frac{\Gamma \vdash \overline{e : T} \qquad T' \in \textsc{CoDt} \qquad \forall\ (d(\overline{T''}) : T' \in \textsc{Dtor}(T)).\ \exists i.\ d = d_i\ \wedge\ \overline{T}, \overline{T''} \vdash e_i : T'}{\Gamma \vdash \mathbf{comatch}\ C\ \mathbf{on}\ T'\ \mathbf{using}\ \overline{e}\ \mathbf{with}\ \overline{d \Rightarrow e'} : T'} \quad (\text{T-Comatch})$$

$$\frac{[\textit{see text}]}{\vdash P\ \ \text{OK}} \quad (\text{Wf-Prog})$$

$\boxed{\text{Evaluation contexts}}$

$$\begin{array}{lll} E & ::= & \square \mid C(\overline{v}, \square, \overline{e}) \ \mid\ \square.d(\overline{e}) \ \mid\ v.d(\overline{v}, \square, \overline{e}) \ \mid\ f(\overline{v}, \square, \overline{e}) \ \mid\ \mathbf{let}\ \square\ \mathbf{in}\ e \\ & \mid & \mathbf{match}\ d\ \mathbf{on}\ \square\ \mathbf{using}\ \overline{e}\ \mathbf{with}\ \overline{C \Rightarrow e} \\ & \mid & \mathbf{match}\ d\ \mathbf{on}\ v\ \mathbf{using}\ (\overline{v}, \square, \overline{e})\ \mathbf{with}\ \overline{C \Rightarrow e} \\ & \mid & \mathbf{comatch}\ C\ \mathbf{on}\ T\ \mathbf{using}\ (\overline{v}, \square, \overline{e})\ \mathbf{with}\ \overline{d \Rightarrow e} \end{array}$$

$\boxed{\text{Small-step Operational Semantics: } e \longrightarrow e'}$

$$\frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]} \ (\text{E-Congr}) \qquad \frac{\textsc{Body}(f) = e}{f(\overline{v}) \longrightarrow e[\overline{v}]} \ (\text{E-Fun}) \qquad \mathbf{let}\ v\ \mathbf{in}\ e \longrightarrow e[v] \ (\text{E-Let})$$

$$\mathbf{match}\ \ldots\ \mathbf{on}\ C(\overline{v})\ \mathbf{using}\ \overline{v'}\ \mathbf{with}\ \ldots, C \Rightarrow e, \ldots \longrightarrow e[\overline{v}][\overline{v'}] \qquad (\text{E-Match})$$

$$\frac{C \Rightarrow e \in \textsc{Cases}(d)}{C(\overline{v}).d(\overline{v'}) \longrightarrow e[\overline{v}][\overline{v'}]} \ (\text{E-Cfun}) \qquad \frac{d \Rightarrow e \in \textsc{Cocases}(C)}{C(\overline{v}).d(\overline{v'}) \longrightarrow e[\overline{v}][\overline{v'}]} \ (\text{E-Gfun})$$

$$(\mathbf{comatch}\ \ \mathbf{on}\ T\ \mathbf{using}\ \overline{v}\ \mathbf{with}\ \ \ldots, d \Rightarrow e, \ldots).d(\overline{v'}) \longrightarrow e[\overline{v}][\overline{v'}] \qquad (\text{E-Comatch})$$

Fig. 11. Typing and small-step operational semantics.

(exhaustiveness), and that names in local xpattern matches are globally unique, as explained in section 3.2. We have omitted the formal definition of the rule WF-PROG from the paper because it is not very interesting; the full definition is of course part of our Coq formalization.

Figure 11 also gives the small-step operational semantics formulated with evaluation contexts.[7] Our language uses call-by-value evaluation, so arguments to functions and terms bound in let expressions and binding lists are evaluated first. Substitution is particularily simple, since it only needs to be defined for values, which are closed terms.[8] Continuing the example from above, if e is a value, then we write the evaluation of the let expression as $\textbf{let } v \textbf{ in } (\textbf{let } e' \textbf{ in } f(1,0)) \rightarrow (\textbf{let } e' \textbf{ in } f(1,0))[v] = \textbf{let } e' \textbf{ in } f(v,0)$. If the body of the function f is the expression e, then we use the suggestive notation $f(\overline{v}) \rightarrow e[\overline{v}]$ to indicate the substitution of the arguments into the body of the function.

The expressions in the cocases of a comatch do not have to be evaluated to values in order for the comatch to be a value. This corresponds to not evaluating expressions under a lambda abstraction, when the function type is generalized to arbitrary codata.

5.4 Type Soundness

For the type soundness of our language (progress and preservation), refer to the overview of our Coq results (subsection 7.1).

6 TRANSPOSITION ALGORITHMS

We implemented transposition as a two-stage process. In a first step a new program skeleton, consisting only of the type signatures, is computed from the given program and the (co)data type T chosen to be transposed. In the new program skeleton, the chosen data type becomes a codata type, or vice versa, with its constructor or destructor signatures collected from the original program, and there are certain changes to the function signatures. The reason to have this stage separate is that it allows us to formulate the statement that typechecking is preserved under transposition. In the second step the new function bodies are computed from the old program. For this, we use constructorization and destructorization functions for expressions in a given program.

In subsection 6.1 we present the running example for the presentation of the algorithm. In subsections 6.2 and 6.3 we present the first and second stage of the algorithm, respectively.

6.1 Example

We will use the example of Figure 12 to illustrate the transposition algorithm. Its formal representation can be found in Figure 13. Constructorizing the codata type `Light` of the program P on the left yields the program P' on the right. Inversely, destructorizing the data type `Light` of P' yields P.

6.2 Stage 1: Computing the New (Co-)Data Types and Function Signatures

The new program skeleton consists of data types, codata types, and the signatures of consumer functions, generator functions, and regular functions, which we obtain from the original program by the changes described in Figure 14. Signatures of regular functions always remain unchanged. The function LOCALCASES(T) will return the signatures of all local matches for type T in a program, i.e. their names, each together with a list of the types in the bindings list and the return type. LOCALCOCASES does the same for local comatches.

[7] The Coq formalization uses standard congruence rules instead of evaluation contexts because we found congruence rules to be easier to formalize.
[8]The technicalities of the proper recursive definition of substitution are part of the Coq formalization.

This means that we remove `Light` from $\textsc{CoDt}$ and add it to $\textsc{Dt}$ and we set $\textsc{Dtor}(\texttt{Light}) = \textsc{Gfun}(\texttt{Light}) = \emptyset$. Furthermore, with $\textsc{localCocases}(\texttt{Light}) = \{\texttt{\_BlueRed}()\}$, we obtain

$$
\begin{aligned}
\textsc{Dt}' &= \textsc{Dt} \cup \{\texttt{Light}\} = \{\texttt{Color}, \texttt{Light}\} \\
\textsc{CoDt}' &= \textsc{CoDt} \setminus \{\texttt{Light}\} = \emptyset \\
\textsc{Ctor}'(\texttt{Light}) &= \textsc{Gfun}(\texttt{Light}) \cup \textsc{localCocases}(\texttt{Light}) \\
&= \{\texttt{Const}(\texttt{Color}), \texttt{RedBlue}()\} \cup \{\texttt{\_BlueRed}()\} \\
\textsc{Cfun}'(\texttt{Light}) &= \textsc{Dtor}(\texttt{Light}) = \{\texttt{color}() : \texttt{Color},\ \texttt{next}() : \texttt{Light}\}.
\end{aligned}
$$

Observe that the two transformations are entirely symmetric in this first stage; i.e. one can exchange constructorization and destructorization, data type and codata type, consumer and generator, as well as match and comatch, and the description remains the same.

6.3 Stage 2: Computing the New Function Bodies

We obtain the new function bodies from the original bodies by the transformations shown in Figure 15. We refer to the result of constructorization of a program P as $\mathfrak{C}[P]$, and to the destructorization result as $\mathfrak{D}[P]$. We also use $\mathfrak{C}$ and $\mathfrak{D}$ for the constructorization and destructorization of expressions, respectively, which we will define in the next paragraph. We denote the collection of cocases for destructor d from all over the original program (i.e. from all generator functions and all local comatches) as cocases(d), and similarly the collection of cases for constructor c as cases(c). It is this collection step that makes the transformation a whole-program transformation, as it requires searching through the entire program for the relevant (co)case bodies. In our running constructorization example, new consumer functions `color` and `next` are added. For instance, the cases of `color` are the following collected cocases(`color`):

```
color() => c
color() => red()
color() => blue()
```

which stem from the generator functions `const`, `RedBlue` and the local comatch `_BlueRed`, respectively.

`Light` is a codata type

```
data Color where
  Red()
  Blue()
codata Light where
  color() : Color
  next()  : Light
gfun Const(c : Color) : Light :=
  color() => c
  next()  => Const(c)
gfun RedBlue() : Light :=
  color() => Red()
  next()  =>
    comatch _BlueRed on Light where
      color() => Blue()
      next()  => RedBlue()
```

`Light` is a data type

```
data Color where
  Red()
  Blue()
data Light where
  Const(Color)
  RedBlue()
  _BlueRed()
cfun Light:color() : Color :=
  Const(c)      => c
  RedBlue()     => Red()
  _BlueRed()    => Blue()
cfun Light:next() : Light :=
  Const(c)      => Const(c)
  RedBlue()     => _BlueRed()
  _BlueRed()    => RedBlue()
```

Fig. 12. Programs P (left) and P' (right).

`Light` is a codata type

$$
\begin{aligned}
\textsc{Dt} &= \{\texttt{Color}\}\\
\textsc{CoDt} &= \{\texttt{Light}\}\\
\textsc{Ctor}(\texttt{Color}) &= \{\texttt{Red}(), \texttt{Blue}()\}\\
\textsc{Dtor}(\texttt{Light}) &= \{\texttt{color}() : \texttt{Color},\\
&\qquad \texttt{next}() : \texttt{Light}\}\\
\textsc{GFun}(\texttt{Light}) &= \{\texttt{Const}(\texttt{Color}),\\
&\qquad \texttt{RedBlue}()\}\\
\textsc{CoCases}(\texttt{Const}) &= \{\texttt{color}() \texttt{ => c},\\
&\qquad \texttt{next}() \texttt{ => Const(c)}\}\\
\textsc{CoCases}(\texttt{RedBlue}) &= \{\texttt{color}() \texttt{ => Red()},\\
&\qquad \texttt{next}() \texttt{ => } [\ldots]^{*}\}
\end{aligned}
$$

Omitted code above ($*$):

```
comatch _BlueRed on Light where
  color() => Blue()
  next()  => RedBlue()
```

`Light` is a data type

$$
\begin{aligned}
\textsc{Dt}' &= \{\texttt{Color}, \texttt{Light}\}\\
\textsc{CoDt}' &= \emptyset\\
\textsc{Ctor}'(\texttt{Color}) &= \{\texttt{Red}(), \texttt{Blue}()\}\\
\textsc{Ctor}'(\texttt{Light}) &= \{\texttt{Const}(\texttt{Color}),\\
&\qquad \texttt{RedBlue}(),\\
&\qquad \texttt{\_BlueRed}()\}\\
\textsc{CFun}'(\texttt{Light}) &= \{\texttt{color}() : \texttt{Color},\\
&\qquad \texttt{next}() : \texttt{Light}\}\\
\textsc{Cases}'(\texttt{color}) &= \{\texttt{Const(c)=> c}\\
&\qquad \texttt{RedBlue()=> Red()}\\
&\qquad \texttt{\_BlueRed()=> Blue()}\}\\
\textsc{Cases}'(\texttt{next}) &= \{\texttt{Const(c)=> Const(c)}\\
&\qquad \texttt{RedBlue()=> \_BlueRed()}\\
&\qquad \texttt{\_BlueRed()=> RedBlue()}\}
\end{aligned}
$$

Fig. 13. The formal representations of P (left) and P' (right).

Constructorization

$$
\begin{aligned}
\textsc{Dt}' &:= \textsc{Dt} \cup \{T\}\\
\textsc{CoDt}' &:= \textsc{CoDt} \setminus \{T\}\\
\textsc{Ctor}(S)' &:= \begin{cases} \textsc{Ctor}(S) & S \neq T \\ \textsc{localCoCases}(T) \\ \ \cup \textsc{GFun}(T) & S = T \end{cases}\\
\textsc{Dtor}(S)' &:= \begin{cases} \textsc{Dtor}(S) & S \neq T \\ \emptyset & S = T \end{cases}\\
\textsc{GFun}(S)' &:= \begin{cases} \textsc{GFun}(S) & S \neq T \\ \emptyset & S = T \end{cases}\\
\textsc{CFun}(S)' &:= \begin{cases} \textsc{CFun}(S) & S \neq T \\ \left\{(d(\overline{T'}) : T'') \in \textsc{Dtor}(T) \,\middle|\, \text{isGlobal}(d)\right\} & S = T \end{cases}\\
\textsc{Fun}' &:= \textsc{Fun}
\end{aligned}
$$

Destructorization

$$
\begin{aligned}
\textsc{Dt}' &:= \textsc{Dt} \setminus \{T\}\\
\textsc{CoDt}' &:= \textsc{CoDt} \cup \{T\}\\
\textsc{Ctor}(S)' &:= \begin{cases} \textsc{Ctor}(S) & S \neq T \\ \emptyset & S = T \end{cases}\\
\textsc{Dtor}(S)' &:= \begin{cases} \textsc{Dtor}(S) & S \neq T \\ \textsc{localCases}(T) \\ \ \cup \textsc{CFun}(T) & S = T \end{cases}\\
\textsc{GFun}(S)' &:= \begin{cases} \textsc{GFun}(S) & S \neq T \\ \left\{C(\overline{T'}) \in \textsc{Ctor}(T) \,\middle|\, \text{isGlobal}(C)\right\} & S = T \end{cases}\\
\textsc{CFun}(S)' &:= \begin{cases} \textsc{CFun}(S) & S \neq T \\ \emptyset & S = T \end{cases}\\
\textsc{Fun}' &:= \textsc{Fun}
\end{aligned}
$$

Fig. 14. The algorithm for computing the new typing information, i.e. the new *skeleton*, when transposing with type T. In this and the following figures, the isGlobal and isLocal predicates check whether a name is global or local, respectively.

Constructorization/Destructorization of Expressions. Constructorization $\mathfrak{C}$ of an expression in a given program and with respect to a type T is shown in Figure 16. For constructorization, the interesting cases are some of those expressions that are related to the type T to be constructorized, specifically:

- Comatches generating T, which become local constructor calls.

Constructorization $\mathfrak{C}$

$$\textsc{Cocases}(f)' := \begin{cases} \mathfrak{C}[\textsc{Cocases}(f)] & f \notin \textsc{Gfun}(T) \\ \emptyset & f \in \textsc{Gfun}(T) \end{cases}$$

$$\textsc{Cases}(f)' := \begin{cases} \mathfrak{C}[\textsc{Cases}(f)] & f \notin \textsc{Cfun}(T)' \\ \left\{ \mathfrak{C}[\text{cocases}(d)] \;\middle|\; (d(\overline{T_p}) : T_r) \in \textsc{Dtor}(T), \text{isGlobal}(d) \right\} & f \in \textsc{Cfun}(T)' \end{cases}$$

$$\textsc{Body}(f)' := \mathfrak{C}[\textsc{Body}(f)] \quad \text{for all } f$$

Destructorization $\mathfrak{D}$

$$\textsc{Cocases}(f)' := \begin{cases} \mathfrak{D}[\textsc{Cocases}(f)] & f \notin \textsc{Gfun}(T)' \\ \left\{ \mathfrak{D}[\text{cases}(C)] \;\middle|\; C(\overline{T_p}) \in \textsc{Ctor}(T), \text{isGlobal}(C) \right\} & f \in \textsc{Gfun}(T)' \end{cases}$$

$$\textsc{Cases}(f)' := \begin{cases} \mathfrak{D}[\textsc{Cases}(f)] & f \notin \textsc{Cfun}(T) \\ \emptyset & f \in \textsc{Cfun}(T) \end{cases}$$

$$\textsc{Body}(f)' := \mathfrak{D}[\textsc{Body}(f)] \quad \text{for all } f$$

Fig. 15. The algorithm for computing the new function bodies when transposing with type T. Here $\textsc{Cfun}(T)$ (resp. $\textsc{Gfun}(T)$ is the set of consumer functions (resp. generator functions) of the *old* program, while $\textsc{Cfun}(T)'$ (resp. $\textsc{Gfun}(T)'$) is the set of consumer functions (generator functions) in the *new* program.

- Generator function calls for generator functions generating T, which become global constructor calls.
- Global destructor calls to destructors of T, which become consumer function calls.

But the most important case is that for *local* destructor calls $e.d(\overline{e})$ (for a destructor of T). Such a local destructor call is translated to a local match. The cases for that match are collected from the comatches and generator functions generating T, fetching all the cocases cocases(d) for the destructor d under consideration. This is the same algorithm as for the collection of the cases of the new consumer functions described above.

The cases for destructors $e.d(\overline{e})$ and generator function calls $C(\overline{e})$ of T look like congruence cases; we list them above the horizontal line because the meaning of those expression changes: The destructor call $e.d(\ldots)$ is turned into a consumer function call $\mathfrak{C}[e].d(\ldots)$, which happens to have the same syntax (because it allows a more economical presentation of the language). Similarly, the generator function call $C(\ldots)$ is turned into a constructor call $C(\ldots)$ that happens to have the same syntax.

Destructorization $\mathfrak{D}$ of expressions (Figure 17) works analogously, with matches turned into local destructor calls, consumer function calls turned into global destructor calls, global constructor calls turned into generator function calls, and local constructor calls turned into comatches with the cocases collected among the matches and consumer functions. There is one slight technical complication in the destructorization part: To check which case of the function definition applies to a `match` expression, we need to know the type of the expression on which we match. We use the notation $e : T$ to refer to that type, which would in an actual implementation be stored and remembered during typechecking. The alternative would have been to make $\mathfrak{D}$ type-directed instead of syntax-directed, but we considered this alternative to be more readable.

Due to local destructor or local constructor calls where the algorithm needs to collect the (co)cases from all over the original program, constructorization/destructorization of expressions is a whole-program transformation itself. Especially, this means that it must take as inputs not only the expression and the type to be transposed, but also the program with respect to which the

$$
\begin{array}{lll r}
\mathfrak{C}[e.d(\overline{e})] & := & \mathfrak{C}[e].d(\overline{\mathfrak{C}[e]}),\ \text{if } d \in \textsc{Dtor}(T) \text{ and isGlobal}(d) & \textit{Destructor} \\
\mathfrak{C}[e.d(\overline{e})] & := & \textbf{match } d \textbf{ on } \mathfrak{C}[e] \textbf{ using } \overline{\mathfrak{C}[e]} \textbf{ with } \mathfrak{C}[\text{cocases}(d)], & \\
 & & \quad \text{if } d \in \textsc{Dtor}(T) \text{ and isLocal}(d) & \\
\mathfrak{C}[C(\overline{e})] & := & C(\overline{\mathfrak{C}[e]}),\ \text{if } C \in \textsc{Gfun}(T) & \textit{Generator function} \\
\multicolumn{3}{l}{\mathfrak{C}[\textbf{comatch } C \textbf{ on } T \textbf{ using } \overline{e : T} \textbf{ with } \ldots] := C(\overline{\mathfrak{C}[e]})} & \textit{Comatch} \\
\hline
\mathfrak{C}[x] & := & x & \textit{Variable} \\
\mathfrak{C}[C(\overline{e})] & := & C(\overline{\mathfrak{C}[e]}) & \textit{Constructor} \\
\mathfrak{C}[e.d(\overline{e})] & := & \mathfrak{C}[e].d(\overline{\mathfrak{C}[e]}),\ \text{if } d \notin \textsc{Dtor}(T) & \textit{Destructor} \\
\mathfrak{C}[f(\overline{e})] & := & f(\overline{\mathfrak{C}[e]}) & \textit{Function} \\
\mathfrak{C}[C(\overline{e})] & := & C(\overline{\mathfrak{C}[e]}),\ \text{if } C \notin \textsc{Gfun}(T) & \textit{Generator function} \\
\mathfrak{C}[e.d(\overline{e})] & := & \mathfrak{C}[e].d(\overline{\mathfrak{C}[e]}) & \textit{Consumer function} \\
\multicolumn{3}{l}{\mathfrak{C}\left[\textbf{match } d \textbf{ on } e \textbf{ using } \overline{e} \textbf{ with } \overline{C \Rightarrow e}\right] :=} & \textit{Match} \\
\multicolumn{3}{l}{\qquad\qquad \textbf{match } d \textbf{ on } \mathfrak{C}[e] \textbf{ using } \overline{\mathfrak{C}[e]} \textbf{ with } \overline{C \Rightarrow \mathfrak{C}[e]}} & \\
\multicolumn{3}{l}{\mathfrak{C}\left[\textbf{comatch } C \textbf{ on } S \textbf{ using } \overline{e} \textbf{ with } \overline{d \Rightarrow e}\right] :=} & \textit{Comatch} \\
\multicolumn{3}{l}{\qquad\qquad \textbf{comatch } C \textbf{ on } S \textbf{ using } \overline{\mathfrak{C}[e]} \textbf{ with } \overline{d \Rightarrow \mathfrak{C}[e]}, \text{for } S \neq T} & \\
\mathfrak{C}[\textbf{let } e_1 \textbf{ in } e_2] & := & \textbf{let } \mathfrak{C}[e_1] \textbf{ in } \mathfrak{C}[e_2] & \textit{Let}
\end{array}
$$

Fig. 16. Constructorization of expressions in a given program w.r.t. a codata type T. Interesting cases are above the horizontal line, congruence cases below.

$$
\begin{array}{lll r}
\mathfrak{D}[C(\overline{e})] & := & C(\overline{\mathfrak{D}[e]}),\ \text{if } C \in \textsc{Ctor}(T) \text{ and isGlobal}(C) & \textit{Constructor} \\
\mathfrak{D}[C(\overline{e})] & := & \textbf{comatch } C \textbf{ on } T \textbf{ using } \overline{\mathfrak{D}[e]} \textbf{ with } \overline{d \Rightarrow X}, & \\
 & & \quad \text{if } C \in \textsc{Ctor}(T) \text{ and isLocal}(C) & \\
\mathfrak{D}[e.d(\overline{e})] & := & \mathfrak{D}[e].d(\overline{\mathfrak{D}[e]}),\ \text{if } d \in \textsc{Cfun}(T) & \textit{Consumer function} \\
\multicolumn{3}{l}{\mathfrak{D}[\textbf{match } d \textbf{ on } e \textbf{ using } \overline{e} \textbf{ with } \overline{C \Rightarrow e}] := \mathfrak{D}[e].d(\overline{\mathfrak{D}[e]}),\ \text{if } e : T} & \textit{Match} \\
\hline
\mathfrak{D}[x] & := & x & \textit{Variable} \\
\mathfrak{D}[C(\overline{e})] & := & C(\overline{\mathfrak{D}[e]}),\ \text{if } C \notin \textsc{Ctor}(T) & \textit{Constructor} \\
\mathfrak{D}[e.d(\overline{e})] & := & \mathfrak{D}[e].d(\overline{\mathfrak{D}[e]}) & \textit{Destructor} \\
\mathfrak{D}[f(\overline{e})] & := & f(\overline{\mathfrak{D}[e]}) & \textit{Function} \\
\mathfrak{D}[C(\overline{e})] & := & C(\overline{\mathfrak{D}[e]}) & \textit{Generator function} \\
\mathfrak{D}[e.d.\overline{e}(]) & := & \mathfrak{D}[e].d(\overline{\mathfrak{D}[e]}),\ \text{if } d \notin \textsc{Cfun}(T) & \textit{Consumer function} \\
\multicolumn{3}{l}{\mathfrak{D}\left[\textbf{match } d \textbf{ on } e \textbf{ using } \overline{e} \textbf{ with } \overline{C \Rightarrow e}\right] :=} & \textit{Match} \\
\multicolumn{3}{l}{\qquad\qquad \textbf{match } d \textbf{ on } \mathfrak{D}[e] \textbf{ using } \overline{\mathfrak{D}[e]} \textbf{ with } \overline{C \Rightarrow \mathfrak{D}[e]} \text{ if } e : S \text{ and } S \neq T} & \\
\multicolumn{3}{l}{\mathfrak{D}\left[\textbf{comatch } C \textbf{ on } S \textbf{ using } \overline{e} \textbf{ with } \overline{d \Rightarrow e}\right] :=} & \textit{Comatch} \\
\multicolumn{3}{l}{\qquad\qquad \textbf{comatch } C \textbf{ on } S \textbf{ using } \overline{\mathfrak{D}[e]} \textbf{ with } \overline{d \Rightarrow \mathfrak{D}[e]}} & \\
\multicolumn{3}{l}{\mathfrak{D}[\textbf{let } e_1 \textbf{ in } e_2] := \textbf{let } \mathfrak{D}[e_1] \textbf{ in } \mathfrak{D}[e_2]} & \textit{Let}
\end{array}
$$

Fig. 17. Destructorization of expressions in a given program w.r.t. a data type T. Interesting cases are above the horizontal line, congruence cases below.

transformation happens, i.e. the program that contains the term. To simplify the development of our Coq implementation, we have therefore split it into three parts: 1.) lifting of (co)matches to top-level functions, 2.) actual program transposition, where we produce top-level generator/consumer functions potentially marked as local (i.e., to be inlined), and 3.) inlining of these marked-as-local functions as (co)matches. This way we have an improved separation of concerns and can implement core constructorization/destructorization of expression functions that do not require the full program as input and are simple folds over the given expression, replacing local xtor calls by local function calls to be inlined later as (co)matches.

7 RESULTS

We have proven the theorems listed in this section, the first part concerning the soundness of our language and the second concerning the correctness of our transformations. The essential, difficult, parts have been mechanically verified in Coq. We did not mechanize proofs of some of the straightforward, but tedious, well-formedness properties of the resulting program which do not relate to well-typedness or totality proper.[9]

7.1 Type Soundness

We have proven the type soundness of our language in Coq by the usual preservation and progress theorems.

Theorem 7.1 (Preservation). *For all expressions e, if $\Gamma \vdash e : T$ in some program P with P OK and $e \rightarrow e'$, then $\Gamma \vdash e' : T$.*

Theorem 7.2 (Progress). *For all programs P with P OK and expressions e, if $\Gamma \vdash e : T$, then either $e \rightarrow e'$, or e is a value.*

Furthermore, we have implemented algorithmic functions implementing the typing and evaluation relations and proven their correctness and completeness with respect to the inductive relations given here. In particular, this shows that the typing and small-step reduction relations are decidable.

7.2 Correctness of the Transformations

For the following, we fix a type T which is used for the transposition. We furthermore assume that the inputs are suitable for the transformations, i.e. that T is a data type if we perform destructorization and a codata type if we perform constructorization.

Theorem 7.3 (Transposition preserves typing). *If e is any expression in the original program P (with P OK) such that $\Gamma \vdash e : T$ holds (in P), then $\Gamma \vdash \mathfrak{C}[e] : T$ holds in $\mathfrak{C}[P]$. Similarly, $\Gamma \vdash \mathfrak{D}[e] : T$ holds in $\mathfrak{D}[P]$.*

Theorem 7.4 (Transposition is total). *If a program P is well-formed (P OK), then transposition will result in a well-formed program P' (P' OK).*

Theorem 7.5 (Transposition preserves reduction relation). *If expression e which is a part of a program P (with P OK) reduces to e' (in P), then $\mathfrak{C}[e]$ reduces to $\mathfrak{C}[e']$ in $\mathfrak{C}[P]$. Likewise $\mathfrak{D}[e]$ reduces to $\mathfrak{D}[e']$ in $\mathfrak{D}[P]$.*

Theorem 7.6 (Transpositions are mutual inverses). *If P OK, then constructorization and destructorization are mutual inverses on all (co)data types defined in P (up to the ordering of signatures, function definitions, or (co)cases).*

[9]All Coq results are summarized in the file `Results.v` in the supplementary material, where we also give explanations or non-mechanized proofs for these tedious admitted parts.

Programs in our language can be naturally thought of as consisting of sets of these signatures, definitions, and co(cases), thus their order does not matter for typechecking or the reduction relation.[10] We have to point out that the mutual inverses property depends on our de Bruijn representation of variables; for a language with ordinary variable names, the property holds only modulo α-equivalence.

8 OUTLOOK

We propose several avenues for follow-up research. We intend to improve the practical applicability of our current work, contribute to the design process of programming languages in a fundamental way, and to bolster the theoretical underpinnings of our language. Regarding the latter, we are interested in (a) generalizing our approach to languages with more elementary features, like first-class continuations, from which we may obtain our present language features as specializations, and (b) to bring our approach to the type level and beyond.

Symmetric Programming Environment. On the practical side, we have developed a prototype of a visual programming environment which allows programmers to switch between the data and codata sides, and to do so with an important feature present that is usually expected from a programming language, namely block structure i.e. local definitions. Our prototype already has some of the features expected of a modern IDE, like (semi-)automatic refactorings, and we intend to develop these features further to make it more useful in practice. Once there is a more general language including e.g. first-class continuations, the catalogue of interesting refactorings may further increase.

Functional vs Object-Oriented Languages. We believe that our work can contribute to reconciling the tension between object-oriented and functional languages. Previous attempts to combine these paradigms have led to non-orthogonal language designs with overlap between language features. Programming with codata is, arguably, the essence of object-oriented programming [Cook 2009]; we believe that our language can be seen as the first truly symmetric and orthogonal language design that combines the ideas of both paradigms; total and inverse transposition is the constructive proof of the symmetry.

Program Matrices and PL Design. The matrix formalism approach to de/refunctionalization was first explored by Rendel et al. [2015] (based on the relation to the expression problem drawn there, and thus also in the tradition of earlier matrix representations [Cook 1990]). Our work did not explicitly employ this kind of formalism, but still, our mental image of the transformations is strongly influenced by the matrix idea.[11] We think it is possible to formalize our work in terms of matrices, but a key difference to prior work is that not everything has global scope anymore: The matrices would need to be enhanced with additional constraints that represent the possible locality restrictions of constructors or destructors. The matrix formalism is close to how we envision the symmetric programming environment. Further, as was hinted at in the introduction, we hope to exploit dualities for PL design, and such an explicit two-dimensional representation of programs can potentially help us better understand the design space of programming languages. We hope that this way one can avoid design mistakes, such as the above mentioned non-orthogonal design found in previous attempts at combining the functional and object-oriented paradigms. To bring

[10] All (co)pattern matches are exhaustive and non-overlapping, therefore such reordering cannot affect any property relating to them.

[11] For instance, our Coq proof of the program transpositions being mutual inverses amounts to reducing this problem to matrix transpose being involutive.

this idea to fruition it might also be instructive to try to combine it with the more general language approach and/or more powerful type systems as outlined in the next paragraphs.

A Deeper Symmetry. Throughout this paper, we emphasized the symmetry inherent in our language and the transformations defined on it. However, if "two-for-the-price-of-one economy" [Wadler 2003] is what one hopes to gain from such symmetries, one could conceivably be far more economical. Observe how constructors take in a certain sense the place of destructors, and vice versa, when switching between the data and the codata sides. Now, compare the structure of constructor signatures $con(\overline{T_c^a}) : T_c$ and of destructor signatures $T_d.des(\overline{T_d^a}) : T_o$: On one side, we have the constructed type T_c, and on the opposite side, the destructed type T_d, which form a dual pair, and on both sides we have lists of arguments ($\overline{T_c^a}$ and $\overline{T_d^a}$, respectively). But the *output* type of destructors T_o lacks a counterpart on its opposite (the constructor) side. It is this lack of a better symmetry that at all forced us to give two accounts, one for constructorization and one for destructorization, or one for the data and one for the codata fragment, at least when explicitly formalizing them in Coq. Even if the sides only differ slightly, the missing symmetry and consequent structural difference is still glaringly obvious. Going back to the major theoretical background for Abel et al.'s seminal work on copatterns [Abel et al. 2013], we can find a solution: Zeilberger [2008] showed how we can learn from polarized logic (and more to the point, the two meaning theories of logic, verificationist and pragmatist) to obtain a deeper symmetry. We believe that our two language fragments can be obtained as specializations of Zeilberger's positive and negative fragments, and the same goes for our combined language and his Calculus of Unity. The thus generalized versions of the constructor/destructor and match/comatch pairs are indeed structurally identical, namely Zeilberger's *values* and *covalues* (which specialize to constructors and destructors), and *continuations* and *expressions* (which specialize to matches and comatches). Going back to the specific issue we mentioned, covalues and continuations do not produce an output, other than destructors and matches. Rather, they expect the remaining computation(s) to be passed as (an) argument(s). Thus, Zeilberger's choice of the name "continuation" here is no accident: This general setting could allow us to better understand the relationship between transposition-related symmetries and continuation-passing style versus direct-style programs (cf. e.g. [Danvy and Nielsen 2001] [Danvy and Millikin 2009]). Related work on the symmetry between greatest and least fixed point in linear logic [Baelde 2012] may also prove to be useful for exploring the symmetry. For this it might be instructive to investigate how to port relevant ideas from linear logic to polarized logic, similarly to what Zeilberger has done.

To the Type Level and Beyond. The data and codata languages and the transformations on them as defined by Rendel et al. [2015] have already been extended to cover parametric polymorphism [Ostermann and Jabs 2018]. One small step ahead could thus be the combination of this and our present work to obtain a language with greater practical applicability. In the future, the approach could be extended to even stronger type systems, especially ones that have a copy of the data fragment/codata fragment duality at the type level and the appropriate transpose transformation. More speculatively, it may also be interesting to study how such a kind of system could be realized in a dependently typed setting, or more generally how one could devise something like the lambda cube [Barendregt 1991] for it, or yet more generally, what the duality means for pure type systems.

Separate Compilation. Finally, transposition of programs offers a new perspective on separate compilation. In most common module systems, extensibility is restricted to either constructors or destructors. However, with our approach it seems possible to import a module and specify for each type which is exported by this module whether it should be imported as data or as codata. In the example from section 3, we could decide if we wanted to import `Int2Int` as a data type with two constructors (`Plus` and `Mult`) and one cfun (`apply`) or as a codata type with one destructor and two gfuns. The first option would allow us to add a second cfun, e.g. `isPlus`, the second option would permit extending with an additional new constructor, e.g. `abs`. If we add facilities to add constructors or destructors to imported types, performing transposition of an imported type also seems possible, e.g. we could import `Int2Int` as a data type, then add the `isPlus` cfun and destructorize afterwards, which would result in a codata type with two cfuns as previously, but with an added destructor, which is specified in the program which imports the type. Unfortunately, the situation is less clear in the presence of local xtors and xmatches. We can identify two avenues for an approach: We can choose to either export local xtors and xmatches or hide them. We consider the first option to be undesirable, since this would expose local xtors, which may only be used once. This means that in most cases, they may not be used in the importing program at all, thus rendering them useless. Hiding local xtors results in a problem, however: If we assume that the type T was exported as a data type with a (hidden) local constructor C and we added a cfun d to T, what would the result of $C.d()$ be? We suggest that it might be possible that this problem can be avoided by specifying translation functions which can be used when a local xtor is encountered. A detailed analysis of such an approach is topic for further research.

9 RELATED WORK

We mainly focus on related work not already discussed in the previous sections.

Defunctionalization. Defunctionalization as a technique to eliminate higher-order functions to make control flow (combined with CPS transformation) explicit goes back to Reynolds's classic essay [Reynolds 1972]. Danvy and colleagues have shown how it can be more widely applied [Danvy and Nielsen 2001], and in particular how it can be usefully combined with CPS transformations to derive semantic artifacts. They also introduced the partial (!) inverse to defunctionalization, refunctionalization [Danvy and Millikin 2009], and showed a similar relation to direct-style transformation. Our case study in section 4 is inspired by their showcase of all these transformations [Danvy et al. 2011].

Coinduction, Codata, and Copatterns. Coinduction and coinductive types are directly supported in some languages such as Coq [Giménez 1996]. Coinductive types still define a data type in terms of its constructors; the main difference to inductive data types is that the semantics changes (greatest fixed point instead of least fixed points, guarded corecursion instead of structural recursion). The modularity/extensibility of the program is not affected by the use of coinductive types. Codata types were first introduced by [Hagino 1989] as an extension of ML. Since then, objects and classes have been described coalgebraically [Jacobs 1995] and codata has been described as the essence of object-oriented programming [Cook 2009]. Abel et al. [2013] proposed a language with both data types/pattern matching and codata/copattern matching, which has inspired an implementation in Agda. Abel et al.'s language is not symmetric in the sense we analyzed in this work because it mixes two forms of codata: codata defined in codata types, and first-class functions. These two forms of codata crucially depend on each other (a destructor with arguments is modeled as a no-argument destructor that resolves to a function that expects the argument). Due to the interplay between functions and codata, it is not obvious what destructorization and constructorization should mean in this language.

Expression Problem. There is a long string of works on programming techniques and language designs that allow simultaneous extensibility in the constructor and destructor dimension (see related work section in [Oliveira and Cook 2012], for instance), usually by enabling a kind of micro-modularity, where the implementation for each constructor/destructor combination can be a separate module that can be freely composed with other such modules. The aim of these works is different from this one. Their goal is to enable extensibility and composability of both constructors and destructors. Our goal is to provide a symmetric language where the extensibility dimension can be switched with our transposition algorithms. There have also been some works that discuss the relation between the data/codata duality and the expression problem. Lämmel and Rypacek [2008] discusses a category-theoretic formulation of the duality between data and codata in terms of the expression problem; however, that work is about semantic methods and not programming language design. The most closely related work to ours is the one by Rendel et al. [2015], whose relation to this work has been discussed in detail at the end of section 2.

Data/Codata Transformations. Downen et al. [2019] compile a language with data types into one with codata types and vice versa, but their transformations are very different from ours. They map data to codata via the visitor pattern, i.e., the result has a codata type with one destructor per constructor in the original data type, and an additional codata type for the visitor. To translate codata to data, they use what they refer to as *tabulation*: the resulting data type represents a table of potential answers to the destructor observations. Laforgue and Régis-Gianas [2017] propose a macro to support codata in OCaml, in which codata operations are reified as a data type and codata types are encoded as dispatch functions on reified codata operations. The essential difference of both works to ours is that the transformations of Downen et al. and Laforgue and Régis-Gianas are compositional and hence do not change the extensibility of the program. Their aim is a compositional encoding, not a change in the decomposition of the whole program.

Defunctionalization in Compilers. Defunctionalization is used as a compiler technique to achieve different goals. In many cases, they remain opaque to the programmer since they do not add new functionality, but are rather used for low-level optimizations. Examples of these kinds of usage include Boquist and Johnsson [1996]'s work on optimizations for lazy functional languages. Sometimes, they might also be employed to provide additional functionality, mainly first-class functions, to a language. One such instance is explored by Grust et al. [2013]. Contrary to this, our approach intends to make different decompositions of a program *accessible* to the programmer and thus provide such transformations as a means to manipulate programs. Moreover, in this work the transpositions are mainly a means to an end, i.e. providing a multi-faceted view of a program to programmers.

10 CONCLUSIONS

We have presented the first full programming language with symmetric support for data and codata and transposition algorithms that flip constructor-centric and destructor-centric program decompositions. We have formalized the language and the transformations and have proven (with most proofs mechanized in Coq) that the language is type-safe and that the transformations are type-preserving, behavior-preserving, total, and inverses of each other. Our case study illustrates how our transposition algorithms can be used, beyond their implications for modularity, to automatically interderive algorithms and data structures that would otherwise require manual work/proof to establish their equivalence. We have validated the language with an implementation extracted from the formalization (which is hence provably correct), and have implemented an IDE which offers the transformations proposed here as a refactoring tool for the programmer.

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming infinite structures by observations. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 27–38.

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 197–208.

David Baelde. 2012. Least and Greatest Fixed Points in Linear Logic. *ACM Trans. Comput. Logic* 13, 1, Article 2 (Jan. 2012), 2:1–2:44 pages.

Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of functional programming* 1, 2 (1991), 125–154.

Urban Boquist and Thomas Johnsson. 1996. The GRIN project: A highly optimising back end for lazy functional languages. In *Symposium on Implementation and Application of Functional Languages*. Springer, 58–84.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2007. Finally tagless, partially evaluated. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer LNCS 4807, 222–238. https://doi.org/10.1007/978-3-540-76637-7_15

William R. Cook. 1990. Object-oriented programming versus abstract data types. In *Proceedings of the REX Workshop / School on the Foundations of Object-Oriented Languages*. Springer-Verlag, 151–178.

William R. Cook. 2009. On understanding data abstraction, revisited. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 557–572.

Olivier Danvy, Jacob Johannsen, and Ian Zerny. 2011. A walk in the semantic park. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*. ACM, New York, NY, USA, 1–12.

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at work. *Science of Computer Programming* 74, 8 (2009), 534–549.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*. 162–174.

Paul Downen, Zachary Sullivan, Zena M Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *European Symposium on Programming*. Springer, 119–146.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., Boston, Massachusetts, USA.

Eduardo Giménez. 1996. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer Berlin Heidelberg.

Torsten Grust, Nils Schweinsberg, and Alexander Ulrich. 2013. Functions are data too: defunctionalization for PL/SQL. *Proceedings of the VLDB Endowment* 6, 12 (2013), 1214–1217.

Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6 (1989), 629–650.

Bart Jacobs. 1995. Objects and classes, coalgebraically. In *Object Orientation with Parallelism and Persistence*. Springer-Verlag, 83–103.

Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203.

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. 1998. Synthesizing Object-Oriented and Functional Design to Promote Re-Use. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECCOP '98)*. Springer-Verlag, Berlin, Heidelberg, 91–113. http://dl.acm.org/citation.cfm?id=646155.679709

Paul Laforgue and Yann Régis-Gianas. 2017. Copattern Matching and First-class Observations in OCaml, with a Macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP '17)*. ACM, New York, NY, USA.

Ralf Lämmel and Ondrej Rypacek. 2008. The Expression Lemma. In *Proceedings of the Conference on Mathematics of Program Construction*. Springer LNCS 5133.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 2–27.

Klaus Ostermann and Julian Jabs. 2018. Dualizing Generalized Algebraic Data Types by Matrix Transposition. In *European Symposium on Programming*. Springer, 60–85.

Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann. 2015. 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 (ICFP 2015)*. ACM, New York, NY, USA, 269–279.

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*. ACM, 717–740.

John C. Reynolds. 1975. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages 1975*, Stephen Schuman (Ed.). IFIP Working Group 2.1 on Algol, INRIA, Rocquencourt, France, 157–168.

Philip Wadler. 1998. The Expression Problem. (Nov. 1998). Note to Java Genericity mailing list.

Philip Wadler. 2003. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 189–201.

Noam Zeilberger. 2008. On the unity of duality. *Annals of Pure and Applied Logic* 153, 1-3 (2008), 66–96.