Automatic Refunctionalization to a Language with Copattern Matching

With Applications to the Expression Problem

Tillmann Rendel   Julia Trieflinger   Klaus Ostermann
University of Tübingen, Germany

Abstract

Defunctionalization and refunctionalization establish a correspondence between first-class functions and pattern matching, but the correspondence is not symmetric: Not all uses of pattern matching can be automatically refunctionalized to uses of higher-order functions. To remedy this asymmetry, we generalize from first-class functions to arbitrary codata. This leads us to full defunctionalization and refunctionalization between a codata language based on copattern matching and a data language based on pattern matching.

We observe how programs can be written as matrices so that they are modularly extensible in one dimension but not the other. In this representation, defunctionalization and refunctionalization correspond to matrix transposition which effectively changes the dimension of extensibility a program supports. This suggests applications to the expression problem.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Defunctionalization, Refunctionalization, Codata, Copattern Matching, Uroboro, Expression Problem

1. Introduction

Defunctionalization transforms programs with higher-order functions into first-order programs with pattern matching (Reynolds 1972; Danvy and Nielsen 2001). Specifically, each function type is replaced by an algebraic data type with one variant for each location in the program where a function of that type is created. The components of each variant represent the values of the free variables in the function body. Application of a function of that type is replaced by a call to an apply function, which dispatches by pattern matching on the algebraic data type. For instance, the program

$$\begin{aligned}&\text{mult}\ n\ y = y * n\\&\text{add}\ n\ y = y + n\\&\text{both}\ (f, (a, b)) = (f\ a, f\ b)\\&\text{example}\ (n, x) = \text{both}\ (\text{mult}\ n, \text{both}\ (\text{add}\ n, x))\end{aligned}$$

looks as follows after defunctionalization:

$$\begin{aligned}&\textbf{data}\ \text{IntToInt} = \text{Mult Int} \mid \text{Add Int}\\&\text{apply}\ (\text{Mult}\ n, y) = y * n\\&\text{apply}\ (\text{Add}\ n,\ y) = y + n\\&\text{both}\ (f, (a, b)) = (\text{apply}\ (f, a), \text{apply}\ (f, b))\\&\text{example}\ (n, x) = \text{both}\ (\text{Mult}\ n, \text{both}\ (\text{Add}\ n, x))\end{aligned}$$

Refunctionalization is the left-inverse of defunctionalization (Danvy and Millikin 2009). It works on programs that are in the image of defunctionalization, that is, there must only be one function that pattern-matches on the algebraic data type. In that case, we can replace calls to apply by function application and constructor applications by abstractions based on the apply function and then remove the algebraic data type and the apply function. Hence we are back at the original program.

Unfortunately, refunctionalization no longer works when more than one function pattern-matches on the algebraic data type. For instance, in the defunctionalized version of the program, we can find out whether a function from Int to Int is the addition function:

$$\begin{aligned}&\text{isAdd}\ (\text{Add}\ \_) = \text{True}\\&\text{isAdd}\ (\text{Mult}\ \_) = \text{False}\end{aligned}$$

This program can no longer be refunctionalized, because there is no way to analyze a function beyond applying it to a value.

The goal of this paper is to remedy this asymmetry between defunctionalization and refunctionalization. Our main insight is that symmetry can be restored by generalizing first-class functions to codata, that is, objects defined by multiple observations (whereas functions are objects defined by just one observation, namely function application). The contributions of this paper are as follows:

- We present Uroboro, a language with pattern and copattern matching (following Abel et al. 2013), and the defunctionalization and refunctionalization between its data and codata fragments (Section 2).
- We formalize the data and codata fragments and show that the total and inverse defunctionalization and refunctionalization preserve typing and behavior (Section 3).
- We observe that the two transformations can be considered a form of matrix transposition (Section 4).
- We relate to the expression problem (Wadler 1998; Reynolds 1975; Cook 1990) by showing that the transformations switch the dimension of extensibility of the program.

Section 5 contains an extension of Reynolds's (1972) original example to demonstrate the utility of defunctionalization and unrestricted refunctionalization. We discuss our results and their relation to previous work in Section 6 and conclude in Section 7.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...
http://dx.doi.org/10.1145/2784731.2784763

```
data Nat where
  zero() : Nat
  succ(Nat) : Nat
function sub(Nat, Nat) : Nat where
  sub(zero(), k) = zero()
  sub(n, zero()) = n
  sub(succ(n), succ(k)) = sub(n, k)
```

Figure 1. Natural numbers and truncated subtraction.

```
codata List where
  List.index(Nat) : Nat
function nil() : List where
  nil().index(n) = error()
function cons(Nat, List) : List where
  cons(head, tail).index(zero()) = head
  cons(head, tail).index(succ(n)) = tail.index(n)
```

Figure 2. Lists of natural numbers represented as codata type.

codata type	$\rightleftharpoons$	data type
copattern-matching function	$\rightleftharpoons$	constructor
destructor	$\rightleftharpoons$	pattern-matching function

Figure 3. How defunctionalization (read left to right) and refunctionalization (read right to left) change the entities in a program.

2. Symmetric Data and Codata in Uroboro

We introduce the language Uroboro with its symmetric support for defunctionalization and refunctionalization and their connection to the expression problem with a series of small examples.

2.1 Natural Numbers as Data Type

Uroboro supports the definition of algebraic data types and the implementation of first-order functions with pattern matching. For example, Figure 1 shows how to define the algebraic data types of natural numbers and a function that takes two numbers. The data type definition for Nat declares the signatures of the two constructors zero (without arguments) and succ (with one argument written in parentheses that is itself of type Nat). After the colon, all constructors of a data type have to use the corresponding data type as return type, in this case, Nat.

Semantically, calling constructors is the only way to create a value of the data type. This means that consumers of a data type can be defined by pattern matching on the constructors. If a consumer handles all constructors, it is sure to support all values of the data type. For example, the function sub is defined by pattern matching on natural numbers. After the **function** keyword, we first give the signature of sub. The function takes two arguments of type Nat and returns Nat. The implementation of sub is given by pattern matching, using the constructor names zero and succ declared with the data type above. The equations deal with all the different ways the arguments could have been created by calling the constructors. The first equation says that subtracting from 0 gives 0 (because of *truncated* subtraction), the second equation says that subtracting 0 doesn't change a number, and the third equation recurses in the case that both arguments to sub are greater than one.

This example is interesting because sub uses some features of pattern matching that are not obvious to refunctionalize: It pattern matches on both function arguments and it uses catch-all patterns instead of enumerating all constructors. We will come back to these aspects in Section 3.2.

We chose the syntax for constructor and function signatures to mimic the syntax of constructor and function calls to highlight the fact that in Uroboro, functions and constructors are second-class entities that can only be used in calls, but cannot form expressions on their own. Consequently, Uroboro does not support higher-order functions directly. Instead, higher-order functions are supported by encoding them as codata types with a singly apply destructor.

2.2 Lists as Codata Type

Uroboro supports the definition of codata types and the implementation of first-order functions with copattern matching. For example, a list of natural numbers can be represented as a partial function from indices to the list element at that index. With first-class functions, we could implement this idea by defining a type synonym $\text{List} = \text{Nat} \to \text{Nat}$. In Uroboro, we define the codata type in Figure 2 instead. The codata type definition for List declares the signatures of the destructor index (with one argument of type Nat). All destructors of a codata type have to use the corresponding codata type as receiver type before the dot, in this case, List.

Semantically, calling destructors is the only way to consume a value of the codata type. This means that creators of a codata type can be defined by copattern matching on the destructors. If a creator supports all destructors, it is sure to provide enough information for all consumers of the codata type. For example, the functions nil and cons are implemented by copattern matching, using the destructor name index declared with the codata type above. The equations deal with all the different ways the result could be consumed by calling the destructor. The equation for nil says that indexing into an empty list is an error. The first equation of cons says that calling index with 0 returns the head of the list. And the second equation delegates a call to index with a higher number to the tail of the list.

Again, the syntax for destructor signatures mimics the syntax for destructor calls. We choose the syntax with the dot after the receiver (as well as the word receiver itself) to resemble the usual presentation of method calls or record accesses. However, note that codata types differ from both object types and record types as they are found in most languages. Unlike usual record types, the components of codata types are evaluated only when a destructor is called. Unlike object types, codata types don't support subtyping, inheritance or implicit self recursion.

2.3 Defunctionalizing Lists

A problem with the implementation of lists in Figure 2 is that it is not immediately apparent how the lists are stored on the heap. We can make the in-memory structure of a codata type more apparent by defunctionalizing[1] it to a data type. With defunctionalization, all functions that copattern match on the original codata type become constructors of a new data type, and all destructors of the original codata type become functions that pattern match on the new data type (see Figure 3, reading left to right).

For example, Figure 4a shows the result of defunctionalizing the list representation from Figure 2. The functions nil and cons become constructors without changing their signature. The destructor index becomes a function, with the receiver moved to the first argument position. The equations are rewritten accordingly and associated to the new function.

The data representation of lists makes it easier to see that the lists are stored as single-linked lists on the heap, and that indexing into a list takes linear time. Since defunctionalization preserves the operational behavior of programs, these insights into the behavior of lists carry over to the codata representation. Carrying over such

[1] We keep the name *defunctionalization* although technically we do not have higher-order functions anymore. Pottier and Gauthier (2006) proposed the more general term *concretization* for translations of introduction forms into injections and elimination forms into case analysis.

```
data List where
  nil() : List
  cons(Nat, List) : List
function index(List, Nat) : Nat where
  index(nil(), n) = error()
  index(cons(head, tail), zero()) = head
  index(cons(head, tail), succ(n)) = index(tail, n)
```

(a) Lists of natural numbers, represented as data type.

```
function null(List) : Bool where
  null(nil()) = true()
  null(cons(head, tail)) = false()
```

(b) Detecting empty lists, modular in the data representation of lists.

Figure 4. Data representation of lists.

```
codata List where
  List.index(Nat) : Nat
  List.null() : Bool
function nil() : List where
  nil().index(n) = error()
  nil().null() = true()
function cons(Nat, List) : List where
  cons(head, tail).index(zero()) = head
  cons(head, tail).index(succ(n)) = tail.index(n)
  cons(head, tail).null() = false()
```

(a) Detecting empty lists, scattered in the codata representation of lists.

```
function repeat(Nat) : List where
  repeat(head).index(n) = head
  repeat(head).null() = false()
```

(b) Creating infinite lists, modular in the codata representation of lists.

Figure 5. Codata representation of lists.

insights was the original goal of defunctionalization in Reynolds's (1972) work on definitional interpreters: To understand a higher-order program (the metacircular interpreter) by studying an operationally equivalent program (the defunctionalized interpreter). In fact, the environment representations in Reynold's interpreters are very similar to the representations of lists in this paper.

It turns out that the data representation of lists make it also easy to add additional consumers of lists, for example, a function that checks whether a list is empty. Since additional consumers are separate functions, they can be added in a modular way in the sense that adding them doesn't require to change any old code, or to intersperse new code with old code. In a practical implementation, they could live in separate files. In this paper, they can live in separate figures, in this example Figure 4b.

2.4 Refunctionalizing Lists

We can undo the defunctionalization by refunctionalizing the program in Figure 4a back to the program in Figure 2. With refunctionalization, all functions that pattern match on the original data type become destructors of a new codata type, and all constructors of the original data type become functions that copattern match on the new codata type (see Figure 3, reading right to left).

Defunctionalization and refunctionalization are both whole-program transformation, because they require to transform *all* functions that copattern match respectively pattern match on a type, not just the functions in any particular module or part of a program. For example, if we want to support the null operation on the codata representation of lists, we have to refunctionalize the code in Figures 4a and 4b together. The result is shown in Figure 5a.

Since we now refunctionalize a data type wich two functions pattern match on, we get a codata type with two destructors. This would not be supported by the usual refunctionalization to a program with higher-order functions. In Uroboro, however, the support for data types and for codata types is more balanced which leads to more symmetric defunctionalization and refunctionalization.

2.5 Modular Extensibility

In Uroboro, like in many languages, it is more modular to add functions than to add constructors or destructors. Functions can be added in a separate part of the program, without changing existing code. But to add constructors or destructors, one has to change the existing data respectively codata type, as well as all existing functions that pattern respectively copattern match on that type. Since defunctionalization and refunctionalization change which aspects of a program are encoded as functions, they also change which dimensions of extensibility are supported in a modular way.

For example, with the codata representation of lists with null in Figure 5a, the three lines of code from the null extension in Figure 4b are scattered to three different locations: The signature of null moves to the declaration of the codata type, the equation for empty lists moves to the definition of the nil function, and the equation for non-empty lists move to the definition of the cons function. This scattering shows how the dimension of extensibility "add consumers" is better supported in the data representation than in the codata representation.

Conversely, the dimension of extensibility "add creators" is better supported in the codata representation than in the data representation, for example, a function that creates a list which infinitely repeats the same element. Since additional creators are separate functions, they can be added modularly without changing any old code or interspersing new code with old code. Again, in a practical implementation, they could live in separate files as here they can live in separate figures, in this example Figure 5b.

We could defunctionalize the code in Figures 5a and 5b to study how the same extension looks like in the data representation of lists. The three lines of code in Figure 5b would be scattered to three different locations the defunctionalized program: The signature of repeat would move to the data type declaration as additional constructor, and the equations would move to the index and null functions, respectively. This scattering shows how codata representation supports the dimension of extensibility "add creators" better.

This trade-off between two dimensions of extensibility relates to Wadler's (1998) expression problem. Traditionally, it is phrased as a trade-off between a functional and an object-oriented decomposition. In the light of the present work, we would rather phrase it as a trade-off between a data-oriented and codata-oriented decomposition, with defunctionalization and refunctionalization as transformations between the two decompositions.

3. Formalization

We formally define the two language fragments that are related by defunctionalization and refunctionalization: One language with data types and pattern matching, the other with codata types and copattern matching. These languages are simply-typed and enjoy type soundness, proven via the usual progress and preservation theorems. The formalization of these languages allows us to clearly define defunctionalization and refunctionalization and meaningfully talk about the property of these transformations. We show that they preserve typing and behavior and are inverse to each other.

$$\begin{array}{lcl}
\sigma, \tau & = & \text{data type names} \\
\mathit{con} & = & \text{constructor names} \\
\mathit{des} & = & \text{destructor names} \\
\mathit{fun} & = & \text{function names} \\
x, y & = & \text{variable names} \\
\mathit{prg} & ::= & \mathit{def}^*
\end{array}$$

(a) Common syntax.

$$\begin{array}{lcl}
\mathit{def} & ::= & \mathbf{data}\ \tau\ \mathbf{where}\ \mathit{sig}^* \\
 & \mid & \mathbf{function}\ \mathit{fun}(\sigma, \tau^*) : \tau\ \mathbf{where}\ \mathit{eqn}^* \\
\mathit{sig} & ::= & \mathit{con}(\tau^*) : \tau \\
\mathit{eqn} & ::= & \mathit{fun}(\mathit{con}(x^*), y^*) = t \\
s, t & ::= & x \mid \mathit{fun}(s, t^*) \mid \mathit{con}(t^*) \\
u, v & ::= & \mathit{con}(v^*) \\
\mathcal{E} & ::= & [] \mid \mathit{fun}(v^*, \mathcal{E}, t^*) \mid \mathit{con}(v^*, \mathcal{E}, t^*)
\end{array}$$

(b) Syntax of the data fragment.

$$\begin{array}{lcl}
\mathit{def} & ::= & \mathbf{codata}\ \tau\ \mathbf{where}\ \mathit{sig}^* \\
 & \mid & \mathbf{function}\ \mathit{fun}(\tau^*) : \tau\ \mathbf{where}\ \mathit{eqn}^* \\
\mathit{sig} & ::= & \sigma.\mathit{des}(\tau^*) : \tau \\
\mathit{eqn} & ::= & \mathit{fun}(x^*).\mathit{des}(y^*) = t \\
s, t & ::= & x \mid \mathit{fun}(t^*) \mid s.\mathit{des}(t^*) \\
u, v & ::= & \mathit{fun}(v^*) \\
\mathcal{E} & ::= & [] \mid \mathit{fun}(v^*, \mathcal{E}, t^*) \mid \mathcal{E}.\mathit{des}(t^*) \mid v.\mathit{des}(v^*, \mathcal{E}, t^*)
\end{array}$$

(c) Syntax of the codata fragment.

Figure 6. Syntactic structure of definitions, terms, values and evaluation contexts.

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

(a) Avoiding the catch-all pattern x in sub's second equation in Figure 1.

```
function sub(Nat, Nat) : Nat where
  sub(zero(), x) = zero()
  sub(succ(x), m) = aux(m, x)
function aux(Nat, Nat) : Nat where
  aux(zero(), x) = succ(x)
  aux(succ(y), x) = sub(x, y)
```

(b) Avoiding to match on the second argument of sub in (a).

```
codata Nat where
  Nat.sub(Nat) : Nat
  Nat.aux(Nat) : Nat
function zero() : Nat where
  zero().sub(x) = zero()
  zero().aux(x) = succ(x)
function succ(Nat) : Nat where
  succ(x).sub(m) = m.aux(x)
  succ(y).aux(x) = x.sub(y)
```

(c) Refunctionalized version of sub in (b).

Figure 7. Subtraction in the data and codata fragments.

3.1 The Data Fragment

The two language fragments have some parts in common that are specified in Figure 6a. In both language fragments, a program *prg* is a list of top-level definitions *def*. The fragments differ in what definitions are allowed in *def*.

The data fragment supports the definition of data types and of functions that pattern match on their first argument. Hence a function is defined by one equation per constructor that could have been used to create the function's first argument. The exact syntax of this fragment is given in Figure 6a and Figure 6b. The restriction to functions that pattern match on their first argument is visible throughout the grammar: The syntax of function signatures $\mathit{fun}\ (\sigma, \tau^*)$ requires that we declare the type of at least one argument, the syntax of function calls $\mathit{fun}\ (s, t^*)$ requires that we provide at least one actual argument, and the syntax of equations $\mathit{fun}\ (\mathit{con}\ (x^*), y^*) = t$ hard-codes the fact that we are pattern matching exactly on the top-level structure of the first argument. To avoid nondeterminism or ambiguities, we assume that all equations of a function match against different constructor names.

The definition of evaluation contexts $\mathcal{E}$ and values v make it clear that the semantics of the data fragment uses call-by-value for function calls, evaluates arguments left-to-right, and has strict data constructors. In this regard, the data fragment is entirely standard.

3.2 Restricted Pattern Matching

The data fragment is unusual in that it supports only the very restricted form of pattern matching on the top-level structure of the first function argument. More liberal languages with pattern matching define a grammar of potentially nested patterns and allow arbitrary patterns for all function arguments in the left-hand sides of equations. The problem with this more liberal treatment of pattern matching is that it is not clear what nested pattern matching or pattern matching on multiple arguments should refunctionalize to. Fortunately, the restriction to top-level pattern matching on the first argument does not restrict the expressivity of our language, because we can desugar nested pattern matching or pattern matching on multiple arguments by introducing helper functions.

For example, the function sub from Figure 1 pattern matches on both function arguments. This can be desugared to the formally defined data fragment by introducing a helper function which performs the second pattern match as shown in Figures 7a and 7b. The basic idea is to bind the value that we want to pattern match on to a variable (here m) and then call the helper function (here aux) with that variable as first argument. The helper function can then perform the pattern match. The right-hand sides of the original equations are copied to the right-hand sides of the helper function. In this case, the original function was recursive, so the desugared function and the helper function will be mutually recursive.

Under the name "disentanglement", this desugaring is performed manually in many works on interderiving semantic artifacts (for example, see Ager et al. 2003). Setzer et al. (2014) call it unnesting and show how to extract an unnesting algorithm from a coverage checker for their dependently-typed language with pattern matching and copattern matching. A similar transformation is performed when a compiler transforms a pattern match into a decision tree.

3.3 The Codata Fragment

The codata fragment is in many ways dual to the data fragment. It supports the definition of codata types and of functions that copattern match on the function result. Hence a function is defined by one equation per destructor that could be used to destruct the function's result. The exact syntax of this fragment is given in Figure 6a and Figure 6c. As with the data fragment, we hard-code the restriction to function definition by copattern matching into the syntax of equations, and we assume that all equations of a function match against different destructor names.

The definition of evaluation contexts $\mathcal{E}$ makes it clear that exactly like with the data fragment, the semantics of the codata fragment uses call-by-value for function calls and evaluates arguments left-to-right. Destructor calls are also evaluated according to the call-by-value strategy. In this regard, the codata fragment doesn't differ from the data fragment. This might come as a surprise:

$$
\begin{array}{rll}
x \rightleftharpoons x & & \\
\mathit{fun}(s, t_1, ..., t_n) \rightleftharpoons s'.\mathit{des}(t'_1, ..., t'_n) & \text{if } \mathit{fun} \rightleftharpoons \mathit{des}, s \rightleftharpoons s', t_1 \rightleftharpoons t'_1, ..., \text{and } t_n \rightleftharpoons t'_n \\
\mathit{con}(t_1, ..., t_n) \rightleftharpoons \mathit{fun}(t'_1, ..., t'_n) & \text{if } \mathit{con} \rightleftharpoons \mathit{fun}, t_1 \rightleftharpoons t'_1, ..., \text{and } t_n \rightleftharpoons t'_n
\end{array}
$$

(a) Rules for the relation $t \rightleftharpoons t'$ between defunctionalized and refunctionalized terms.

$$
\begin{aligned}
\langle x \rangle^{\mathrm{r}} &= x \\
\langle \mathit{fun}(s, t_1, ..., t_n) \rangle^{\mathrm{r}} &= \langle s \rangle^{\mathrm{r}}.\langle \mathit{fun} \rangle^{\mathrm{r}}(\langle t_1 \rangle^{\mathrm{r}}, ..., \langle t_n \rangle^{\mathrm{r}}) \\
\langle \mathit{con}(t_1, ..., t_n) \rangle^{\mathrm{r}} &= \langle \mathit{con} \rangle^{\mathrm{r}}(\langle t_1 \rangle^{\mathrm{r}}, ..., \langle t_n \rangle^{\mathrm{r}})
\end{aligned}
$$

(b) Refunctionalisation $\langle t \rangle^{\mathrm{r}}$ of terms.

$$
\begin{aligned}
\langle x \rangle^{\mathrm{d}} &= x \\
\langle s.\mathit{des}(t_1, ..., t_n) \rangle^{\mathrm{d}} &= \langle \mathit{des} \rangle^{\mathrm{d}}(\langle s \rangle^{\mathrm{d}}, \langle t_1 \rangle^{\mathrm{d}}, ..., \langle t_n \rangle^{\mathrm{d}}) \\
\langle \mathit{fun}(t_1, ..., t_n) \rangle^{\mathrm{d}} &= \langle \mathit{fun} \rangle^{\mathrm{d}}(\langle t_1 \rangle^{\mathrm{d}}, ..., \langle t_n \rangle^{\mathrm{d}})
\end{aligned}
$$

(c) Defunctionalization $\langle t \rangle^{\mathrm{d}}$ of terms.

Figure 8. Defunctionalization and refunctionalization of terms.

$$
\begin{array}{rcll}
\text{“}\mathit{fun}(\sigma, \tau_1, ..., \tau_n)\text{”} \in \mathit{prg} & \Longleftrightarrow & \text{“}\sigma.\mathit{des}'(\tau_1, ..., \tau_n)\text{”} \in \mathit{prg}' & \text{if } \mathit{fun} \rightleftharpoons \mathit{des}' \text{ and } \mathit{prg} \rightleftharpoons \mathit{prg}' \\
\text{“}\mathit{con}(\tau_1, ..., \tau_n)\text{”} \in \mathit{prg} & \Longleftrightarrow & \text{“}\mathit{fun}'(\tau_1, ..., \tau_n)\text{”} \in \mathit{prg}' & \text{if } \mathit{con} \rightleftharpoons \mathit{fun}' \text{ and } \mathit{prg} \rightleftharpoons \mathit{prg}' \\
\text{“}\mathit{fun}(\mathit{con}(x_1, ..., x_n), y_1, ..., y_k) = t\text{”} \in \mathit{prg} & \Longleftrightarrow & \text{“}\mathit{fun}'(x_1, ..., x_n).\mathit{des}'(y_1, ..., y_k) = t'\text{”} \in \mathit{prg}' & \\
 & & & \text{if } \mathit{fun} \rightleftharpoons \mathit{des}', \mathit{con} \rightleftharpoons \mathit{fun}', t \rightleftharpoons t' \text{ and } \mathit{prg} \rightleftharpoons \mathit{prg}'
\end{array}
$$

(a) Intended consequences of the relation $\mathit{prg} \rightleftharpoons \mathit{prg}'$ between defunctionalized and refunctionalized programs.

$$
\begin{aligned}
\langle \mathit{prg} \rangle^{\mathrm{r}} = \{\, & \textbf{codata } \sigma \textbf{ where} \\
& \quad \{\, \sigma.\langle \mathit{fun} \rangle^{\mathrm{r}}(\tau_1, ..., \tau_n) : \tau \\
& \quad \mid \text{“}\mathit{fun}(\sigma, \tau_1, ..., \tau_n) : \tau\text{”} \in \mathit{prg} \,\} \\
& \mid \text{“ } \textbf{data } \sigma \text{ ...”} \in \mathit{prg} \,\} \\
\cup \ \{\, & \textbf{function } \langle \mathit{con} \rangle^{\mathrm{r}}(\tau_1, ..., \tau_n) \textbf{ where} \\
& \quad \{\, \langle \mathit{con} \rangle^{\mathrm{r}}(x_1, ..., x_n).\langle \mathit{fun} \rangle^{\mathrm{r}}(y_1, ..., y_k) = \langle t \rangle^{\mathrm{r}} \\
& \quad \mid \text{“}\mathit{fun}(\mathit{con}(x_1, ..., x_n), y_1, ..., y_k) = t\text{”} \in \mathit{prg} \,\} \\
& \mid \text{“}\mathit{con}(\tau_1, ..., \tau_n) : \tau\text{”} \in \mathit{prg} \,\}
\end{aligned}
$$

(b) Refunctionalisation $\langle \mathit{prg} \rangle^{\mathrm{r}}$ of programs.

$$
\begin{aligned}
\langle \mathit{prg} \rangle^{\mathrm{d}} = \{\, & \textbf{data } \sigma \textbf{ where} \\
& \quad \{\, \langle \mathit{fun} \rangle^{\mathrm{d}}(\tau_1, ..., \tau_n) : \tau \\
& \quad \mid \text{“}\mathit{fun}(\tau_1, ..., \tau_n) : \tau\text{”} \in \mathit{prg} \,\} \\
& \mid \text{“}\textbf{codata } \sigma \text{ ...”} \in \mathit{prg} \,\} \\
\cup \ \{\, & \textbf{function } \langle \mathit{des} \rangle^{\mathrm{d}}(\sigma, \tau_1, ..., \tau_n) \textbf{ where} \\
& \quad \{\, \langle \mathit{des} \rangle^{\mathrm{d}}(\langle \mathit{fun} \rangle^{\mathrm{d}}(x_1, ..., x_n), y_1, ..., y_k) = \langle t \rangle^{\mathrm{d}} \\
& \quad \mid \text{“}\mathit{fun}(x_1, ..., x_n).\mathit{des}(y_1, ..., y_k) = t\text{”} \in \mathit{prg} \,\} \\
& \mid \text{“}\sigma.\mathit{des}(\tau_1, ..., \tau_n) : \tau\text{”} \in \mathit{prg} \,\}
\end{aligned}
$$

(c) Defunctionalization $\langle \mathit{prg} \rangle^{\mathrm{d}}$ of programs.

Figure 9. Defunctionalization and refunctionalization of programs.

Shouldn't the codata fragment use some form of call-by-name evaluation to support infinite values? Indeed, there is support for infinite values, hidden in the definition of values v. In the codata fragment, function calls are values. This means that function calls are not performed until a destructor is called on the function's result.

3.4 Defunctionalization and Refunctionalization

To specify defunctionalization and refunctionalization, we can now define a one-to-one relationship between programs in the data and in the codata fragments. Both directions of this relationship can be implemented as transformations, that is, we can mechanically defunctionalize a program in the codata fragment to the related program in the data fragment; and we can mechanically refunctionalize a program in the data fragment to the corresponding program in the codata fragment. Since the relation is one-to-one, defunctionalization and refunctionalization are inverse to each other.

We assume that the transformations do not change variable names or type names, and that one-to-one relations $\mathit{fun} \rightleftharpoons \mathit{des}$ and $\mathit{con} \rightleftharpoons \mathit{fun}$ are set up to map function and constructor names in the data fragment to destructor and function names in the codata fragment, respectively. In all examples, we simply use the same names for related entities in the two fragments. In this formalization, we still make the potential renaming explicit to clarify the difference between function and constructor names which is not readily apparent from their uses in function respectively constructor calls.

We first define which terms t in the data language are related to which terms t' in the codata language, written $t \rightleftharpoons t'$. The relation $\rightleftharpoons$ on terms is defined inductively on the syntax of terms by the rules in Figure 8a. The rules specify how function application in the data fragment relates to destructor application in the codata fragment, and how constructor application in the data fragment relates to function application in the data fragment. This is the same relationship as informally introduced in Figure 3.

Reading the rules for the $\rightleftharpoons$ relation on terms left-to-right, we can extract the recursive transformation from data terms to codata terms in Figure 8b. And reading the rules right-to-left, we can extract the recursive transformation from codata terms to data terms in Figure 8c. By construction, these transformations are inverse to each other. We also observe that $\rightleftharpoons$ relates data values with codata values. This allows us to lift $\rightleftharpoons$ as well as the transformations $\langle \cdot \rangle^{\mathrm{d}}$ and $\langle \cdot \rangle^{\mathrm{r}}$ to evaluation contexts by pointwise application.

We cannot specify the relation $\rightleftharpoons$ on programs in such a syntactic way because defunctionalization and refunctionalization operate on whole programs. In particular, they collect all function definitions and put them in a single data respectively codata type. We therefore specify the relation $\rightleftharpoons$ up to reordering of top-level definitions in terms of the containment of equations and function, constructor and destructor signatures in programs.

This specification is shown in Figure 9a. It describes how all parts of one program show up in the related program, just at different places. The first two lines describe how function, constructor and destructor signatures relate in the two fragments. Necessarily, these relationships mimic the relationships of function, constructor and destructor calls from Figure 8a. The last line of the specification describes that related programs basically have the same equations, but are written differently.

We can implement transformations between related programs by loops over the input program. The set comprehensions in

$$\frac{\begin{array}{l}\text{“}\mathit{fun}(\tau_1,...,\tau_n):\tau\text{”}\in\Sigma\\ \quad\Gamma\vdash_\Sigma t_1:\tau_1\\ \qquad\ddots\\ \qquad\Gamma\vdash_\Sigma t_n:\tau_n\end{array}}{\Gamma\vdash_\Sigma \mathit{fun}(t_1,...,t_n):\tau}
\qquad
\frac{\begin{array}{l}\text{“}\mathit{con}(\tau_1,...,\tau_n):\tau\text{”}\in\Sigma\\ \quad\Gamma\vdash_\Sigma t_1:\tau_1\\ \qquad\ddots\\ \qquad\Gamma\vdash_\Sigma t_n:\tau_n\end{array}}{\Gamma\vdash_\Sigma \mathit{con}(t_1,...,t_n):\tau}
\qquad
\frac{\begin{array}{l}\text{“}\sigma.\mathit{des}(\tau_1,...,\tau_n):\tau\text{”}\in\Sigma\\ \quad\Gamma\vdash_\Sigma s:\sigma\\ \qquad\Gamma\vdash_\Sigma t_1:\tau_1\\ \qquad\quad\ddots\\ \qquad\quad\Gamma\vdash_\Sigma t_n:\tau_n\end{array}}{\Gamma\vdash_\Sigma s.\mathit{des}(t_1,...,t_n):\tau}$$

$$\frac{\text{“}x:\tau\text{”}\in\Gamma}{\Gamma\vdash_\Sigma x:\tau}
\qquad
\frac{\begin{array}{l}\text{“}\mathit{con}(\sigma_1,...,\sigma_n):\sigma\text{”}\in\Sigma\\ \quad\text{“}\mathit{fun}(\sigma,\tau_1,...,\tau_k):\tau\text{”}\in\Sigma\\ \quad\quad x_1:\sigma_1,...,x_n:\sigma_n,\\ \quad\quad y_1:\tau_1,...,y_k:\tau_k\vdash_\Sigma t:\tau\end{array}}{\Sigma\vdash \mathit{fun}(\mathit{con}(x_1,...,x_n),y_1,...,y_k)=t\text{ ok}}
\qquad
\frac{\begin{array}{l}\text{“}\mathit{fun}(\sigma_1,...,\sigma_n):\sigma\text{”}\in\Sigma\\ \quad\text{“}\sigma.\mathit{des}(\tau_1,...,\tau_k):\tau\text{”}\in\Sigma\\ \quad\quad x_1:\sigma_1,...,x_n:\sigma_n,\\ \quad\quad y_1:\tau_1,...,y_k:\tau_k\vdash_\Sigma t:\tau\end{array}}{\Sigma\vdash \mathit{fun}(x_1,...,x_n).\mathit{des}(y_1,...,y_k)=t\text{ ok}}$$

(a) Common typing rules. (b) Additional typing rules for data fragment. (c) Additional typing rules for codata fragment.

$$\frac{\mathit{prg}\vdash t\rightsquigarrow t'}{\begin{array}{l}\mathit{prg}\vdash\mathcal{E}[t]\rightsquigarrow\\ \qquad\mathcal{E}[t']\end{array}}
\qquad
\frac{\text{“}\mathit{fun}(\mathit{con}(x_1,...,x_n),y_1,...,y_k)=t\text{”}\in\mathit{prg}}{\begin{array}{l}\mathit{prg}\vdash \mathit{fun}(\mathit{con}(u_1,...,u_n),v_1,...,v_k)\rightsquigarrow\\ \qquad t[x_1\mapsto u_1,...,x_n\mapsto u_n,y_1\mapsto v_1,...,y_k\mapsto v_k]\end{array}}
\qquad
\frac{\text{“}\mathit{fun}(x_1,...,x_n).\mathit{des}(y_1,...,y_n)=t\text{”}\in\mathit{prg}}{\begin{array}{l}\mathit{prg}\vdash \mathit{fun}(u_1,...,u_n).\mathit{des}(v_1,...,v_k)\rightsquigarrow\\ \qquad t[x_1\mapsto u_1,...,x_n\mapsto u_n,y_1\mapsto v_1,...,y_k\mapsto v_k]\end{array}}$$

(d) Congruence rule. (e) Contraction rule for data fragment. (f) Contraction rule for codata fragment.

Figure 10. Static and dynamic semantics.

$$\begin{array}{l}\mathrm{sub(s(s(z())),s(z()))}\rightsquigarrow\mathrm{aux(s(z()),s(z()))}\rightsquigarrow\mathrm{sub(s(z()),z())}\rightsquigarrow\mathrm{aux(z(),z())}\rightsquigarrow\mathrm{s(z())}\\ \mathrm{s(s(z())).sub(s(z()))}\rightsquigarrow\mathrm{s(z()).aux(s(z()))}\rightsquigarrow\mathrm{s(z()).sub(z())}\rightsquigarrow\mathrm{z().aux(z())}\rightsquigarrow\mathrm{s(z())}\end{array}$$

Figure 11. Reduction sequences for computing $2-1=1$ using the programs in Figure 7b (upper sequence) and 7c (lower sequence). The identifiers succ and zero are abbreviated as s and z.

Figure 9b describe the steps necessary for refunctionalization of programs: We first loop over all data types in the original program and transform them to codata types. In the inner loop, for each data type σ, we collect all function signatures from the original program that have σ as first argument and transform them into destructor signatures for the newly created codata type. Then we loop over all constructor signatures in the original program and transform them to functions. In the inner loop, for each constructor *con*, we loop over all equations in the original program that pattern match on *con* and transform them into equations for the newly created function. Defunctionalization is defined similarly by the set comprehensions in Figure 9c. It is easy to see that these transformations implement the specification from Figure 9a and that they are inverse of each other, up to ordering of program elements.

For example, a refunctionalized version of the program in Figure 7b is shown in Figure 7c. These two programs are related by $\rightleftharpoons$. It is interesting to see how the helper function aux introduced in Section 3.2 gets refunctionalized to a helper destructor. In object-oriented programming, this corresponds to a typical approach of simulating double dispatch.

3.5 Typing

We now define a static type system for the data and codata fragments to show that defunctionalization and refunctionalization preserve typing. The well-typedness of expressions is defined with respect to a signature Σ which contains all function, constructor and destructor signatures that occur in a program (but not the equations), and a context Γ which contains type assignments for variables. The typing rules are formally defined in Figure 10 using rules for the following judgments: $\Gamma\vdash_\Sigma t:\tau$ means that expression t has type τ under signature Σ and context Γ, and $\Sigma\vdash\mathit{eqn}$ ok means that equation *eqn* is well-typed under signature Σ.

The rules in Figure 10a are common to both language fragments. Note that in the data fragment, functions need to have at least one argument, so n in rule FUN cannot be 0 if the rule is used to check an expression in the data fragment, but everywhere else in Figure 10, n or k can be 0. Figure 10b and Figure 10c list the typing rules for the data and codata fragments, respectively. Since the language fragments are both first-order and don't support local variable binding, typing is very simple. In particular, all binding occurrences of variables are in the left-hand sides of equations, and all bound occurrences are in the right-hand sides of equations, so the two rules for the $\Sigma\vdash\mathit{eqn}$ ok judgment are the only rules that need to manipulate the typing context. A program is well-typed if all its equations are well-typed.

A program *prg* is complete with respect to a signature Σ if the equations in the program uniquely cover all combinations of functions and constructors induced by Σ. For the data fragment, this means that for all "$\mathit{con}\ (\sigma_1,...,\sigma_n):\sigma$" $\in\Sigma$ and "$\mathit{fun}\ (\sigma_1,\tau_1,...,\tau_k):\tau$" $\in\Sigma$, there is a unique t so that "$\mathit{fun}\ (\mathit{con}\ (x_1,...,x_n),y_1,...,y_k)=t$" $\in\mathit{prg}$. And for the codata fragment, it means that for all "$\mathit{fun}\ (\sigma_1,...,\sigma_n):\sigma$" $\in\Sigma$ and "$\sigma_1.\mathit{dst}\ (\tau_1,...,\tau_k):\tau$" $\in\Sigma$, there is a unique t so that "$\mathit{fun}(x_1,...,x_n).\mathit{des}(y_1,...,y_k)=t$" $\in\mathit{prg}$.

We can apply $\rightleftharpoons$, $\langle\cdot\rangle^{\mathrm{r}}$ and $\langle\cdot\rangle^{\mathrm{d}}$ to signatures Σ analogously to their definition on programs *prg*. We see that given $\Sigma\rightleftharpoons\Sigma'$ and $\mathit{prg}\rightleftharpoons\mathit{prg}'$, the program *prg* is complete with respect to Σ if and only if the program prg' is complete with respect to Σ'. We can use the same typing context Γ for related programs in the data and codata fragments because the names of variables and types remain unchanged during defunctionalization and refunctionalization. This allows us to state the following lemmas about the fact that $\rightleftharpoons$ preserves typing for terms and equations:

Lemma 1. Given $\Sigma\rightleftharpoons\Sigma'$, $t\rightleftharpoons t'$, and $\tau\rightleftharpoons\tau'$, we can derive $\Gamma\vdash_\Sigma t:\tau$ if and only if we can derive $\Gamma\vdash_{\Sigma'} t':\tau'$.

Proof. We prove each direction by induction on the typing derivation we are given. If it is the rule for variables, we are done, because the typing context is unchanged. If it is one of the other rules, we use the induction hypotheses and one of the first two properties in Figure 9a to construct the corresponding derivation. ☐

Lemma 2. Given $\Sigma \rightleftharpoons \Sigma'$ and $prg \rightleftharpoons prg'$, then all equations in prg are well-typed if and only if all equations in prg' are well-typed.

Proof. For each equation having to show its well-typedness, we pick the corresponding relation in the transformed program from which we know that it is already well-typed (by the third property in Figure 9a). We observe that the equations construct the same variables and finish the prove with Lemma 1. ☐

3.6 Semantics

As mentioned before the function equations are used for rewriting, so we perform reduction steps until no rewriting rule can be used furthermore. In case of our data-language this means constructor calls and in case of our codata-language this means function calls.

We formally specify the dynamic semantics of the two language fragments by rules for judgments of the form $prg \vdash t \rightsquigarrow t'$. This judgment states that given the equations in program prg, the term t reduces to t' in one step. The language fragments share the standard congruence rule in Figure 10d, albeit each fragment defines their own notion of evaluation context $\mathcal{E}$ (in Figures 6b and 6c). We need only one additional reduction rule per language fragment to specify its semantics, because their equations have restricted shape.

For the data fragment, the rule in Figure 10e specifies how to execute a function call if the first argument has already been evaluated to a constructor application. In this case, the function call is replaced by the right-hand side of the equation for that particular function-constructor combination, with the free variables appropriately substituted. And for the codata fragment, the rule in Figure 10f specifies how to execute a destructor call if the first argument has already been evaluated to a function application. In this case, the destructor call is replaced by the right-hand side of the equation for that particular function-destructor combination, with the free variables appropriately substituted. For example, Figure 11 shows the reduction sequences that arise from computing $3 - 2$ using the definitions of sub from Figures 7b or 7c, respectively.

Lemma 3. If in either language fragment, Σ is the signature of prg, all equations in prg are well-typed, $\Gamma \vdash_\Sigma t : \tau$, and $prg \vdash t \rightsquigarrow t'$ then $\Gamma \vdash_\Sigma t' : \tau$.

Proof. We prove this by induction on the derivation of $prg \vdash t \rightsquigarrow t'$. For the congruence rule, we use induction on the evaluation context $\mathcal{E}$ to construct the typing derivation which is necessary to use the induction hypothesis. For the other rules, we use the fact that all equations are well-typed and a standard substitution lemma (proven by induction on the structure of terms). ☐

Lemma 4. If in either language fragment, prg is complete with respect to Σ, all equations in prg are well-typed, and $\vdash_\Sigma t : \tau$, then either t is v, or there exists t' so that $prg \vdash t \rightsquigarrow t'$.

Proof. We prove this by induction on the derivation of $\Gamma \vdash_\Sigma t : \tau$. The case for variables is impossible because the context is empty. In the other cases, we apply the induction hypotheses for all sub-derivations, from left to right, until we find the first subexpression of t that is reducible. If we find a reducible subexpression, we construct a reduction derivation with the corresponding evaluation context. If all subexpressions of t are values, we find that either t is a value or t is reducible. In the latter case, since prg is complete with respect to Σ, we know that the equation, necessary to construct a reduction derivation for t, is available in prg.

We don't need a lemma for canonical forms, because in each of the fragments, there is only one form of values. ☐

Lemma 5. Given $prg \rightleftharpoons prg'$, $s \rightleftharpoons s'$, and $t \rightleftharpoons t'$, then $prg \vdash s \rightsquigarrow t$ if and only if $prg' \vdash s' \rightsquigarrow t'$.

Proof. We prove both directions by induction on the reduction derivation we are given. For the congruence rule, we use induction on the evaluation context $\mathcal{E}$ to construct the reduction derivation necessary to use the induction hypothesis. This uses the similarity of the evaluation contexts for the two language fragments. For the other rules, we use the last property from Figure 9a and a lemma that states how $\rightleftharpoons$ interacts with substitution (proven by induction on the structure of terms). ☐

The previous lemma shows that de- and refunctionalization preserve the operational semantics of terms in the strong sense that evaluation of related terms proceeds in lockstep. This allows us to study the operational behavior of a term by studying the operational behavior after de- or refunctionalizing it. In the case of definitional interpreters, this use case is particularly important, because studying the operational behavior of an interpreter corresponds to studying the operational behavior of the interpreted language.

Writing $\rightsquigarrow^*$ for the reflexive, transitive closure of $\rightsquigarrow$, we can also state a weaker result about the result of the reduction of normalizing terms.

Lemma 6. Given $prg \rightleftharpoons prg'$, $s \rightleftharpoons s'$, and $v \rightleftharpoons v'$, then $prg \vdash s \rightsquigarrow^* v$ if and only if $prg' \vdash s' \rightsquigarrow^* v'$.

Proof. We prove both directions by induction on the length of the reduction sequence, constructing a corresponding reduction sequence of equal length using Lemma 5 for every step. ☐

With the last lemma it is clear that the transformation of a program evaluates to the same result and this with the same number of evaluation steps.

4. Transformations as Matrix Transpositions

We now want to highlight another way to view the two languages and their relation through de- and refunctionalization, namely as matrices and matrix transposition, respectively. Figure 12 shows how we can arrange the two programs from Figure 7 in matrices so that defunctionalization and refunctionalization correspond to matrix transposition.

If we unify the syntax of function and destructor calls (e.g., $\mathrm{aux}\,(m, x)$ vs $m.\mathrm{aux}\,(x)$) to only use the former variant, and do the same with declarations (e.g., write $\mathrm{sub}\,(\mathrm{Nat}, \mathrm{Nat}) : \mathrm{Nat}$ for both $\mathrm{sub}(\mathrm{Nat}, \mathrm{Nat}) : \mathrm{Nat}$ and $\mathrm{Nat}.\mathrm{sub}(\mathrm{Nat}) : \mathrm{Nat}$) we can also show both versions of the program in a single matrix:

	zero() : Nat	succ(Nat) : Nat
sub(Nat,Nat) : Nat	zero()	aux(m,x)
aux(Nat,Nat) : Nat	succ(x)	sub(x,y)

A row-by-row reading of the matrix corresponds to the program variant in Figure 12a, whereas a column-by-column reading corresponds to Figure 12b. This means that both defunctionalization and refunctionalization can be understood as matrix transpositions.

Adding a new row to the matrix means to extend the program with a new consumer, which could be done in a modular way in the data language to not have to scatter code and to avoid the expression problem. Analogously adding a new column means to extend the program with a new creater structure which could be done modularly in the codata language. If someone wants to add both, rows and columns, this can be achieved by repeatedly transposing the matrix using defunctionalization and refunctionalization.

$$\begin{array}{rcc}
\textbf{data}\ \mathrm{Nat}\ \textbf{where} & \mathrm{zero}()\ :\ \mathrm{Nat} & \mathrm{succ}(\mathrm{Nat})\ :\ \mathrm{Nat} \\
\textbf{function}\ \mathrm{sub}(\mathrm{Nat},\mathrm{Nat}) : \mathrm{Nat}\ \textbf{where} & \mathrm{sub}(\mathrm{zero}(), x) = \mathrm{zero}() & \mathrm{sub}(\mathrm{succ}(x), m) = \mathrm{aux}(m, x) \\
\textbf{function}\ \mathrm{aux}(\mathrm{Nat},\mathrm{Nat}) : \mathrm{Nat}\ \textbf{where} & \mathrm{aux}(\mathrm{zero}(), x) = \mathrm{succ}(x) & \mathrm{aux}(\mathrm{succ}(y), x) = \mathrm{sub}(x, y)
\end{array}$$

(a) The program from Figure 7b arranged as a matrix.

$$\begin{array}{rcc}
\textbf{codata}\ \mathrm{Nat}\ \textbf{where} & \mathrm{Nat.sub}(\mathrm{Nat})\ :\ \mathrm{Nat} & \mathrm{Nat.aux}(\mathrm{Nat})\ :\ \mathrm{Nat} \\
\textbf{function}\ \mathrm{zero}() : \mathrm{Nat}\ \textbf{where} & \mathrm{zero}().\mathrm{sub}(x) = \mathrm{zero}() & \mathrm{zero}().\mathrm{aux}(x) = \mathrm{succ}(x) \\
\textbf{function}\ \mathrm{succ}(\mathrm{Nat}) : \mathrm{Nat}\ \textbf{where} & \mathrm{succ}(x).\mathrm{sub}(m) = m.\mathrm{aux}(x) & \mathrm{succ}(y).\mathrm{aux}(x) = x.\mathrm{sub}(y)
\end{array}$$

(b) The program from Figure 7c arranged as a matrix.

Figure 12. If we write programs as matrices, defunctionalization and refunctionalization correspond to matrix transposition.

If one organizes the matrix a bit differently, then it is not just a projection of the programs but one can reconstruct the programs from the matrix by a row-by-row or column-by-column, respectively, reading. For the data type reading, we can reconstruct the data declarations by assembling all constructors with the same return type to a data type declaration. The entries in the first column give us the signatures of the functions. Similarly, for the codata reading, we can assemble all destructors with the same first argument into a codata declaration. The first row gives us the signatures of the functions. Since we can organize the matrix such that all declarations that belong together are adjacent to each other, one linear pass through the matrix without book-keeping is sufficient to reconstruct all declarations.

But we cannot yet reconstruct the full function definitions because the binding positions of the variables are not specified. However, if we fix the names of variables to be, from left to right, *x0*, *x1* etc., we can avoid the issue. This works because there is only one way to write the left hand side of an equation (see Section 3.1). In this version, the matrix for the example looks like this:

	zero() : Nat	succ(Nat) : Nat
sub(Nat,Nat) : Nat	zero()	aux(x1,x0)
aux(Nat,Nat) : Nat	succ(x0)	sub(x1,x0)

We have implemented a prototype of Uroboro in which the full program is stored as a matrix as above and in which we use a standard matrix transposition function to perform defunctionalization and refunctionalization.

We believe that the idea to represent programs as matrices rather than as trees is interesting on its own. Furthermore, a matrix-like depiction of programs is standard in presentations about the expression problem. We have formalized this graphical metaphor.

5. Case Study

To illustrate the power of defunctionalization with full refunctionalization, we follow Reynolds (1972, Sec. 5) and consider a metacircular interpreter for the untyped lambda calculus, written in a language with higher-order functions. We will first present a meta interpreter where closures are represented by closures, then defunctionalize it to a syntactic interpreter, then extend the interpreter with normalization-by-evaluation, and then refunctionalize it back to the codata language.

5.1 The Object Language

We focus on the pure untyped lambda calculus, that is, application, lambda abstraction and variable occurrences, only adding a term err which leads to an immediate error when executing. We represent bound variables as de Bruijn indices. These changes will be useful later in this section, when we add reification of values to terms in order to achieve normalization by evaluation. In the data fragment of Uroboro, we can express this abstract syntax as a data structure Exp shown in Figure 14.

```
codata Exp where
  Exp.eval(Env) : Val
function var(Nat) : Exp where
  var(name).eval(env) = env.index(name)
function app(Exp, Exp) : Exp where
  app(fun, arg).eval(env) =
    fun.eval(env).apply(arg.eval(env))
function fun(Exp) : Exp where
  fun(body).eval(env) = closure(body, env)
function err() : Exp where
  err().eval(env) = error()
codata Val where
  Val.apply(Val) : Val
function closure(Exp, Env) : Val where
  closure(body, env).apply(arg) = body.eval(cons(arg, env))
function error() : Val where
  error().apply(arg) = error()
function interpret(Exp) : Val where
  interpret(e) = e.eval(nil())
```

Figure 13. Meta Interpreter

The data type Exp supports variables (var), lambda expressions (fun) and application (app). Note that our use of de Bruijn indices means that we don't store binding variable occurrences in the fun nodes. Instead, bound variable occurrences count how many lambdas we have to jump over before we find the lambda with the binding occurrence. For example, the term *double* $\equiv \lambda f.\lambda x.f(fx) \equiv \lambda\lambda 1(1\ 0)$ is represented as as follows:

```
fun(fun(app(var(succ(zero())),
            app(var(succ(zero())),
                var(zero())))))
```

5.2 Metacircular Interpretation

Using codata types to encode higher-order functions, we can closely follow Reynolds's (1972) metacircular interpreter as shown in Figure 13. Unlike Reynolds, we also represent Exp as refunctionalized data structure using a codata type because we want to focus on the codata fragment of Uroboro for the meta interpreter.

Both environments and values are represented by codata types. Since environments for de Bruijn indices are, apart from the element type, identical to lists as defined in Figure 2, we do not repeat its definition here and assume that they are defined as in Section 2

```
data Exp where
  var(Nat) : Exp
  app(Exp, Exp) : Exp
  fun(Exp) : Exp
  err() : Exp
data Val where
  closure(Exp, Env) : Val
  error() : Val
function eval(Exp, Env) : Val where
  eval(var(x), env) = index(env, x)
  eval(app(e_1, e_2), env) =
    apply(eval(e_1, env), eval(e_2, env))
  eval(fun(body), env) = closure(body, env)
  eval(err(), env) = error()
function apply(Val, Val) : Val where
  apply(closure(body, env), arg) =
    eval(body, cons(arg, env))
  apply(error(), arg) = error()
function interpret(Exp) : Val where
  interpret(e) = eval(e, nil())
```

Figure 14. Defunctionalization of the interpreter in Figure 13 yields this more syntactic interpreter.

with the following superficial changes: The type is called Env instead of List, and the type of list elements is Val instead of Nat. The helper function closure creates values. The main entry point is interpret which calls eval with an initial environment. Since we don't provide any built-in operations, we can use the empty environment here. Finally, the code of eval is distributed among the functions var, app, fun, and err, similar to a pure embedding(Hudak 1998) of the lambda calculus.

5.3 Defunctionalization to a Syntactic Interpreter

Defunctionalization of the codata types Val, Env and Exp yields the more syntactic interpreter shown in Figure 14. We only show the result of defunctionalizing Val and Exp; the defunctionalization of Env is as in Figure 4a.

A well-known benefit of defunctionalization is that it is usually easier to understand the memory layout of algebraic data types than to understand the memory layout of first-class functions. In this case, the defunctionalization makes it clear that values are stored as closures, and environments are stored as lists.

Defunctionalization also collects all cases of eval together into a function that is defined by pattern matching on the syntax of expressions. This suggests that defunctionalization and refunctionalization can describe the relationship between shallow and deep embedding of a language.

5.4 Reification

Another benefit of defunctionalization is that once we have a representation of a function (or rather, codata) space as algebraic data type, we can add more functions that pattern match on values of that type. In this case, let us add a function reify that takes a value and returns an expression in normal form which would evaluate to that value. In other words, let us implement normalization by evaluation (Berger and Schwichtenberg 1991).

For example, if *double* is the representation of $\lambda f.\lambda x.f\ (f\ \ x)$ from above, then the normal form of *double* applied to itself is the representation of $\lambda f.\lambda x.f\ (f\ (f\ (f\ \ x)))$. Using our reify function, we can compute this representation as follows:

$$\text{reify}(\text{eval}(\text{app}(\mathit{double}, \mathit{double}), \text{nil}()), \text{zero}())$$

```
data Val where
  closure(Exp, Env) : Val
  error() : Val
    -- step 4:
  resVar(Nat) : Val
    -- step 7:
  resApp(Val, Val) : Val
function apply(Val, Val) : Val where
  apply(closure(body, env), arg) =
    eval(body, cons(arg, env))
  apply(error(), arg) = error()
    -- step 6:
  apply(resVar(level), arg) =
    resApp(resVar(level), arg)
    -- step 9:
  apply(resApp(v_1, v_2), v_3) =
    resApp(resApp(v_1, v_2), v_3)
  -- step 1:
function reify(Val, Nat) : Exp where
    -- step 2:
  reify(error(), level) = err()
    -- step 3:
  reify(closure(body, env), level) =
    fun(reify(eval(body, cons(resVar(succ(level)), env)),
             succ(level)))
    -- step 5:
  reify(resVar(outer), inner) =
    var(sub(inner, outer))
    -- step 8:
  reify(resApp(v_1, v_2), level) =
    app(reify(v_1, level), reify(v_2, level))
```

Figure 15. Extending the syntactic interpreter from Figure 14 to implement normalization-by-evaluation.

Figure 15 shows all necessary changes to the vanilla syntactic interpreter in Figure 14. In order to understand how to come up with this implementation, we go through the necessary changes in an order they could have been done in.

1. Our goal is to write reify(Val, Nat) : Exp so that it transforms a value back into a term in normal form. The additional Nat argument is the de Bruijn level of the first variable to be bound inside the returned expression. We need this information in order to compute de Bruijn indices for variables bound outside but used inside the returned expression.
2. The case for error() is easy because we took care to add err() to the set of expressions.
3. In the case for closure(*body*, *env*), we would like to return a lambda expression with a body in normal form. To normalize the body, we want to evaluate and then reify the body from the closure, treating the freshly bound variable as a residual term that is already in normal form. Assuming a constructor resVar(Nat) : Val which creates such a residual variable (at a given de Bruijn level), we can complete the case.
4. Now we have to actually add the new resVar constructor for residual variables.

5. Since we added a constructor for Val, we have to implement reification for it. A residual variable bound at de Bruijn level $outer$ and used at de Bruijn level $inner$ is reified to a variable with de Bruijn index $inner - outer$. We use sub as a helper function to subtract natural numbers as necessary for the computation of de Bruijn indices from de Bruijn levels.
6. We also have to extend the apply function to deal with the new resVar constructor for residual variables. Applying a residual variable to a value creates a residual application, so to implement it, we have to assume the addition of yet another constructor resApp for the algebraic data type of values.
7. Next, we actually add the new resApp constructor.
8. For the new constructor, we have to extend reify again. We reify a residual application by reifying the operator and operand, and then constructing an application.
9. Finally, we also have to reify apply for the new resApp constructor. Luckily, applying a residual application just creates another residual application, so we don't have to add any more constructors, and this completes our implementation of normalization-by-evaluation.

We learn two lessons from this experiment: On the one hand, it was possible to extend the defunctionalized form of the interpreter because we could just add additional functions that pattern match on the defunctionalized codata space. But on the other hand, we had to change many parts of the program when we added new constructors to the defunctionalized codata space. Changing already existing parts of a program is not good from a modularity and maintainability perspective.

We recognize this as an instance of the expression problem: The two dimensions of extensibility are the addition of functions that consume values and the addition of kinds of values. In the defunctionalized form, the former is well-supported, and the latter is ill-supported in a modular way.

5.5 There and Back Again

At this point, we want to undo the defunctionalization, that is, we would like to refunctionalize the interpreter back to use codata to see a different trade off between the two dimensions of extensibility. In a conventional functional language we would be stuck at this point, because after the addition of normalization-by-evaluation, our program is no longer in the image of defunctionalization, because there is more than one function that pattern matches on Val.

In Uroboro, however, codata is not restricted to a single observation, hence we can simply add another destructor $\text{Val.reify}(\text{Nat}) : \text{Exp}$ to the Val codata type, as shown in Figure 16. The comments in this figure also highlight the changes necessary to add normalization-by-evaluation, with the same step numbers as in Figure 15.

Thinking about the expression problem again, we observe the the relationship between dimensions of extensibility and support for modular changes summarized in Figure 17. We see again that in the defunctionalized form, adding consumers of values is well-supported but adding ways to constructor values is ill-supported in a modular way. And conversely, we see that in the refunctionalized form, adding consumers is ill-supported and adding ways to construct is well-supported in a modular way. This is a typical situation with respect to the expression problem: Two complementary ways to encode information support different dimensions of extensibility. This case study confirms the authors' intuition that defunctionalization and refunctionalization could be a theoretical foundation for thinking about the expression problem, as well as for describing the various solutions for the expression problem that are based on carefully combining data and codata types.

```
codata Val where
  Val.apply(Val) : Val
    -- step 1:
  Val.reify(Nat) : Exp
function closure(Exp, Env) : Val where
  closure(body, env).apply(arg) =
    body.eval(cons(arg, env))
    -- step 3:
  closure(body, env).reify(level) =
    fun(body.eval(cons(resVar(succ(level)), env))
            .reify(succ(level)))
  -- step 4, 5, 6:
function resVar(Nat) : Val where
  resVar(outer).reify(inner) =
    var(sub(inner, outer))
  resVar(level).apply(arg) =
    resApp(resVar(level), arg)
  -- step 7, 8, 9:
function resApp(Val, Val) : Val where
  resApp(v_1, v_2).reify(level) =
    app(v_1.reify(level), v_2.reify(level))
  resApp(v_1, v_2).apply(v_3) =
    resApp(resApp(v_1, v_2), v_3)
function error() : Val where
  error().apply(arg) = error()
    -- step 2:
  error().reify(level) = err()
```

Figure 16. Refunctionalization of the interpreter in Figure 15 yields this more metacircular implementation of normalization by evaluation.

Step	Dimension	Defunct.	Refunct.
1	add consumer	modular	nonmodular
2	add consumer	modular	nonmodular
3	add consumer	modular	nonmodular
4	add constructor	nonmodular	modular
5	add constructor	nonmodular	modular
6	add helper	modular	modular
7	add constructor	nonmodular	modular
8	add constructor	nonmodular	modular
9	add constructor	nonmodular	modular
10	add constructor	nonmodular	modular

Figure 17. Modular support for different changes in the defunctionalized and refunctionalized variants of the interpreter.

6. Related and Future Work

Danvy and his collaborators have developed a long-standing program to interrelate semantic artifacts (such as big-step semantics, small-step semantics, abstract machines) through systematic transformations, such as CPS transformation, closure conversion, refocusing (for example, Danvy and Millikin 2009; Ager et al. 2003; Danvy and Nielsen 2001). Defunctionalization and refunctionalization are two key components in this program. We believe that a better correspondence between these two transformations can have a positive influence on the whole program.

Cook discussed the relation between object-oriented programming and abstract data types (Cook 2009). We believe that our work can be seen as a formalization of the relation as described by Cook.

While our language does not support abstract data types through a type system, a data type definition together with all functions that operate on it can be seen as an abstract data type, and programmers could, by disciplined usage, ensure that representation independence holds. Also, the variant of objects described by Cook fits well to our support of codata and copattern matching. Ignoring the missing enforcement of representation independence, defunctionalization and refunctionalization as described in this paper hence correspond to the relation between ADTs and objects as described in Sec. 4.2 and 4.3 of Cook's paper.

Janzen and de Volder (2004) discuss a programming system in which one can both view and edit a program in two different decompositions, namely a decomposition into object-oriented classes and a decomposition into "modules", which collect all implementations of a method into one module. Our approach can be seen as a semantic justification for their approach. Integrating the transformations proposed in this paper into an IDE to switch between these two "views" and edit the program in the one that fits best to the task at hand would also be a straightforward application of this paper.

Lämmel and Rypacek (2008) also investigate the duality between data and codata and their relation to the expression problem. They focus on semantic methods and a theoretical description of the duality, using category theory, whereas we focus on syntactic methods and the design of a practical language, using basic programming language methodology. It would be interesting to understand the exact relationship between their results and our transformations.

Among others, Carette et al. (2009) propose to implement domain-specific languages (EDSLs) by a form of Church encoding. This requires to specify every semantics of an EDSL in a compositional way. It appears as if disentanglement (Section 3.2) followed by refunctionalization could be used to automatically transform a non-compositional function on an initial embedding (data types and pattern matching) to a compositional semantics that is suitable for use with the final embedding. As expected, the compositional semantics would include some extra information that is only needed to achieve compositionality, in the form of additional destructors.

Our use of copattern matching derives from Abel et al.'s (2013) work. To achieve symmetry with our first-order, simply-typed data fragment, we leave out polymorphic and dependent types, and merge Abel's application copattern and destructor copatterns into a form of destructor copatterns that also support arguments. In our future work, we want to consider more powerful type systems for Uroboro. As a first step, we want to consider polymorphism. It is known that defunctionalization of polymorphic functions requires generalized algebraic data types (GADTs) (Pottier and Gauthier 2006). We expect that for refunctionalization of polymorphic functions we need to invent something like generalized codata types.

7. Conclusions

We have shown that defunctionalization and refunctionalization can be made symmetric by generalizing higher order functions to codata. We believe that this result is significant both from a theoretical and from a pragmatic point of view. It provides a strong justification for programming languages with codata and copattern matching and may as such inform the design of new functional programming languages. The transformations can also be used as programming techniques, either in the design of automated tools (or even IDEs) or simply as another tool in the programmer's toolbox of powerful program transformations. We have seen that the two languages we defined also shed new light on the expression problem, since they correspond to the two forms of extensibility that are in the focus of the expression problem. Finally, the organization of programs as matrices and the transformations as transpositions of these matrices suggests a novel view of programs as two-dimensional (rather than tree-structured) entities, which we believe to be interesting to explore in future work.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. Paolo G. Giarrusso pointed out the possible connection between disentangling and automatic compositionalization. Olivier Danvy provided feedback that helped us in the preparation of the final version. Tobias Weber implemented a typechecker for Uroboro which we used to type check the code examples in the figures.

References

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 27–38. ACM, 2013.

M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM, 2003.

U. B. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In *Proceedings of the Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society, 1991.

J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, Sept. 2009.

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

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

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

O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*, pages 162–174, 2001.

P. Hudak. Modular domain specific languages and tools. In *Proceedings of the Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

D. Janzen and K. de Volder. Programming with crosscutting effective views. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 195–218. Springer LNCS 3086, 2004.

R. Lämmel and O. Rypacek. The Expression Lemma. In *Proceedings of the Conference on Mathematics of Program Construction*. Springer LNCS 5133, July 2008.

F. Pottier and N. Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, Mar. 2006.

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

J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages 1975*, pages 157–168. IFIP Working Group 2.1 on Algol, INRIA, 1975.

A. Setzer, A. Abel, B. Pientka, and D. Thibodeau. Unnesting of copatterns. In *Proceedings of the Joint Conference on Rewriting Techniques and Applications and Typed Lambda Calculi and Applications*, pages 31–45. Springer LNCS 8560, 2014.

P. Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.