Down with Variables

Alcino Cunha **Jorge Sousa Pinto** **José Proença**
{alcino,jsp,proenca}@di.uminho.pt

Techn. Report DI-PURe-05.06.01
2005, June

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-05.06.01
Down with Variables by Alcino Cunha and Jorge Sousa Pinto and José Proença

Abstract

The subject of this paper is *point-free* functional programming in Haskell. By this we mean writing programs using categorically-inspired combinators, algebraic data types defined as fixed points of functors, and impicit recursion through the use of type-parameterized recursion patterns. This style of programming is appropriate for *program calculation* (reasoning about programs equationally), but difficult to actually use in practice – most programmers use a mixture of the above elements with explicit recursion and manipulation of arguments. In this paper we present a mechanism that allows programmers to convert classic point-wise code into point-free style, and a Haskell library that enables the direct execution of the resulting code. Together, they make possible the use of point-free either as a direct programming style or as a domain into which programs can be transformed before being subject to further manipulation.

1 Introduction

The origins of the point-free style can be traced back to the ACM Turing Award Lecture given by John Backus in 1977 [1]. Instead of explicitly referring arguments, Backus recommended the use of *functional forms* (combinators) to build functions by combining simpler ones. The particular choice of combinators should be driven by the power of the associated algebraic laws; the desired result is an effective program calculus in which equational reasoning and program transformation can be performed.

Category theory has proven very successful in providing such a set of combinators. Function composition, the most fundamental combinator of point-free programming, is also the fundamental concept of this theory, and the universal characterization of types in a categorical setting is a good source for combinators and the associated laws. In particular, we will be interested in *almost bicartesian closed categories*, that is, categories with products, non-empty sums, exponentials, and terminal object. The set of combinators that characterizes these categories is nowadays standard among point-free programmers [23, 3, 11].

The point-free style of programming is usually combined with the use of *recursion patterns* to replace explicit recursion. These encapsulate standard shapes of recursion that, likewise to point-free combinators, are characterized by a rich set of equational laws. In the context of equational reasoning these laws replace fixpoint induction, and in program transformation they can be used to shortcut the classic fold/unfold transformation cycle [4]. For example, well known concepts like folding or fusion over lists were first introduced by Bird to derive accumulator-based implementations of inefficient specifications [2].

The drawback of using the point-free style is that, as the examples in this paper show, programs written without variables are not always easy to write or understand. In fact, it is virtually impossible to program without using variables here and there. The difficulties associated with the point-free style are our point of departure for the present paper.

This paper has two goals. The first is to present a mechanism for translating pointwise Haskell [14] code into point-free form. Equipped with such a tool, programmers may freely program in their favorite idiom and then have variables automatically eliminated, so that reasoning, calculation, and transformation can be performed. Interestingly, the need for such a mechanism was recently mentioned in the Haskell mailing list, in a thread concerning the advantages/disadvantages of the point-free style [17]:

[. . .] it is easier to reason equationally with point-free programs, even if the intended computation is often easier for mere mortals to see when named values are used. So point-free style helps when trying to apply program transformation techniques, and translation to make greater use of point-free idioms may be a useful precursor to transforming a program.

The second goal is to present a library for point-free programming in Haskell, that will be used to execute the code that results from the above translation. Of course, it can also be used to directly program in the point-free style with recursion patterns. Following the typical joke about point-free programming we have named this the Pointless Haskell library. It inherits a polytypic implementation of recursion patterns from PolyP [25], a generic programming library. With the help of extensions to the Haskell type system, we have implemented an implicit coercion mechanism that provides a limited form of structural equivalence between types. This has allowed us to embed in Haskell a syntax almost identical to the one used at the theoretical level for point-free terms.

Section 2 presents the basic point-free combinators and the equational laws that characterize them. It also describes how they are implemented in the Pointless library. Section 3 starts with the presentation of a simply-typed λ-calculus with products and sums, and then describes how it can be translated into the point-free style. Section 4 introduces recursion: data types are viewed as fixed points of functors, and recursive functions are declared implicitly by means of recursion patterns, namely *hylomorphisms*. Section 5 describes how Pointless Haskell incorporates generic recursion patterns and recursive types within the point-free style. Section 6 extends the pointwise to point-free translation to handle recursion: it defines two methods to replace fixpoint definitions by hylomorphisms. Section 7 shows how a limited form of pattern matching can be added to the core λ-calculus, making it easier to encode many typical Haskell definitions.

2 Point-free Programming

A category is a collection of objects and a collection of arrows between objects, to be used as denotations for types and functions, respectively. Two arrows $g : A \to B$ and $f : B \to C$ can be composed as $f \circ g : A \to C$, and for each object A there exists an identity arrow $\mathsf{id}_A : A \to A$. Composition and identity obey the following associative and naturality

laws, that are implicitly used in calculations.

$$f \circ (g \circ h) = (f \circ g) \circ h$$
$$\mathsf{id} \circ f = f \circ \mathsf{id} = f$$

Notice that typing subscripts in the basic constants (such as id) will be dropped when irrelevant or derivable from context. The same applies to the types of the meta-variables mentioned in laws.

Most research on point-free programming has been carried out in the context of total functions and total elements. Technically, this means that the concrete category is **Set** (of sets and total functions). Unfortunately, this category is not a good semantic model for most functional languages, namely Haskell, since it hardens the treatment of arbitrary recursive and partial definitions. Another problem is that finite and infinite data types are different things that cannot be combined, thus excluding, for example, functions defined by induction that work for both, and the hylomorphism recursion pattern to be presented later. We overcome these problems by moving to the category **CPO**, where objects are pointed complete partial orders and arrows are continuous functions, as described in [23]. In practice this means that every object A has a distinguished bottom element (denoted $\perp_A$), and some laws became polluted with strictness side-conditions (a function is strict if it preserves bottoms).

An object $\mathsf{1}$ is *terminal* if given any other object A there exists exactly one arrow from A to $\mathsf{1}$ denoted bang_A. The terminal object in **CPO** is the singleton set whose only element is $\perp_1$. bang is characterized by the following laws.

$$\mathsf{bang}_1 = \mathsf{id}_1 \qquad \text{Bang-Reflex}$$
$$\mathsf{bang} \circ f = \mathsf{bang} \qquad \text{Bang-Fusion}$$

Elements of a type A are represented categorically by arrows of type $1 \to A$, usually called *points*. Given an element $x \in A$, $\underline{x}$ denotes the corresponding point in the category. By composing points with bang it is possible to define constant morphisms, that ignore the argument and always return a specific value.

The product of two objects A and B is an object $A \times B$, together with a pair of projections $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$, such that for every object C, and arrows $g : C \to A$ and $h : C \to B$, there exists exactly one arrow from C to $A \times B$, denoted by $g \vartriangle h$, satisfying the following law.

$$f = g \vartriangle h \quad \Leftrightarrow \quad \mathsf{fst} \circ f = g \wedge \mathsf{snd} \circ f = h \qquad \text{Prod-Uniq}$$

In **CPO** products are implemented by cartesian products, with obvious choice for the projections, and the *split* combinator implemented as $(g \vartriangle h)\ x = (g\ x, h\ x)$. Due to Prod-Uniq, products are characterized by the following laws.

$$\mathsf{fst} \vartriangle \mathsf{snd} = \mathsf{id} \qquad \text{Prod-Reflex}$$

$$\mathsf{fst} \circ (f \vartriangle g) = f \ \wedge\ \mathsf{snd} \circ (f \vartriangle g) = g \qquad \text{Prod-Cancel}$$

$$(f \vartriangle g) \circ h = f \circ h \vartriangle g \circ h \qquad \text{Prod-Fusion}$$

It is also possible to define a product combinator and the absorption law that relates it to split. Notice that composition has higher priority than all the remaining combinators.

$$f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd} \qquad \text{Prod-Def}$$

$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \text{Prod-Absor}$$

As a first example of point-free programming and equational reasoning let us define the swap function

$$\begin{aligned} &\mathsf{swap}\ :\ A \times B \to B \times A \\ &\mathsf{swap} = \mathsf{snd} \vartriangle \mathsf{fst} \end{aligned}$$

and prove that it justifies the isomorphism $A \times B \simeq B \times A$.

$$\mathsf{swap} \circ \mathsf{swap} = \mathsf{id} \qquad \text{Swap-Iso}$$

$$\begin{aligned} &\mathsf{swap} \circ \mathsf{swap} \\ =\ &\quad \{\ \mathsf{swap}\ \text{definition}\ \} \\ &(\mathsf{snd} \vartriangle \mathsf{fst}) \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\ =\ &\quad \{\ \text{Prod-Fusion}\ \} \\ &\mathsf{snd} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \vartriangle \mathsf{fst} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\ =\ &\quad \{\ \text{Prod-Cancel}\ \} \\ &\mathsf{fst} \vartriangle \mathsf{snd} \\ =\ &\quad \{\ \text{Prod-Reflex}\ \} \\ &\mathsf{id} \end{aligned}$$

The coproduct of two objects A and B is an object $A + B$, together with a pair of injections $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$, such that for every object C, and arrows $g : A \to C$ and $h : B \to C$, there exists exactly one arrow from $A + B$ to C, denoted by $g \triangledown h$, satisfying the following law.

$$f = g \triangledown h \quad \Leftrightarrow \quad f \circ \mathsf{inl} = g \ \wedge\ f \circ \mathsf{inr} = h \qquad \text{Coprod-Uniq}$$

Unfortunately, it is well know that **CPO** does not have true coproduts. In this category, as in most lazy functional languages, this construction is usually approximated by the *separated sum* (henceforth denoted just by sum), where a new bottom element is added to the tagged union of the elements of both sets.

$$A + B = (\{0\} \times A) \cup (\{1\} \times B) \cup \{\bot_{A+B}\}$$

On sums it is possible to define the injections and the *either* combinator as follows.

$$\begin{array}{ll} \mathsf{inl}\ x = (0, x) & (g \triangledown h)\ \bot = \bot \\ \mathsf{inr}\ x = (1, x) & (g \triangledown h)\ (0, x) = g\ x \\ & (g \triangledown h)\ (1, x) = h\ x \end{array}$$

Although Coprod-Uniq is not verified by sums, they can still be given a unique characterization if attention is restricted to strict functions.

$$f = g \triangledown h \Leftrightarrow f \circ \mathsf{inl} = g \wedge f \circ \mathsf{inr} = h \wedge f \text{ strict} \qquad \text{Sum-Uniq}$$

Again, the following laws can be derived from Sum-Uniq. Notice that fusion can only be applied to strict functions.

$$\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id} \qquad \text{Sum-Reflex}$$

$$(f \triangledown g) \circ \mathsf{inl} = f \wedge (f \triangledown g) \circ \mathsf{inr} = g \qquad \text{Sum-Cancel}$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \quad \Leftarrow \quad f \text{ strict} \qquad \text{Sum-Fusion}$$

Dually to products it is possible to define the sum function combinator.

$$f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g \qquad \text{Sum-Def}$$

$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad \text{Sum-Absor}$$

Finally we also have exponentials. The exponentiation of an object B to A is an object B^A, together with an application function $\mathsf{ap} : B^A \times A \to B$, such that for every object C, and $g : C \times A \to B$, $\overline{g}$ is the unique arrow from C to B^A satisfying the following law.

$$f = \overline{g} \quad \Leftrightarrow \quad g = \mathsf{ap} \circ (f \times \mathsf{id}) \qquad \text{Exp-Uniq}$$

In **CPO** the object B^A contains all continuous functions with domain A and codomain B, whose least element is the function that always returns $\bot_B$. The application and *curry* combinators are defined as

$$\mathsf{ap}\ (f, x) = f\ x \qquad \overline{g}\ x\ y = g\ (x, y)$$

Some of the laws that characterize exponentiation are

$$\overline{\mathsf{ap}} = \mathsf{id} \qquad \text{Exp-Reflex}$$
$$\mathsf{ap} \circ (\overline{f} \times \mathsf{id}) = f \qquad \text{Exp-Cancel}$$
$$\overline{f} \circ g = \overline{f \circ (g \times \mathsf{id})} \qquad \text{Exp-Fusion}$$

and the exponentiation combinator can be defined as follows.

$$f^{\bullet} = \overline{f \circ \mathsf{ap}} \qquad \text{Exp-Def}$$
$$f^{\bullet} \circ \overline{g} = \overline{f \circ g} \qquad \text{Exp-Absor}$$

The following functions distribute the product over the sum and vice-versa.

$$\begin{aligned} &\mathsf{distr} \;:\; A \times (B + C) \to (A \times B) + (A \times C) \\ &\mathsf{distr} = (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ \mathsf{swap} \\ &\mathsf{undistr} \;:\; (A \times B) + (A \times C) \to A \times (B + C) \\ &\mathsf{undistr} = (\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr}) \end{aligned}$$

Unfortunately, due to the absence of coproducts, **CPO** is not a *distributive category*, which means that $A \times (B + C)$ is not isomorphic to $(A \times B) + (A \times C)$. In fact, there are more elements in the first type: distr maps a value $(x, \bot)$ into $\bot$, making impossible for undistr to recover the original x. Even so, it is still true that $\mathsf{distr} \circ \mathsf{undistr} = \mathsf{id}$.

A category that has products, exponentials, and terminal object is called *cartesian closed*. In these categories, the set of arrows from A to B can be represented by the object B^A. Formally, this means that an $f : A \to B$ can be internalized as a point $\underline{f} : 1 \to B^A$, and vice versa. Moreover, it is possible to define the conversions between a function and its point using the basic combinators previously defined. Given any arrow $f : A \to B$ we have [21]

$$\underline{f} = \overline{f \circ \mathsf{snd}} \qquad \text{To-Point}$$
$$f = \mathsf{ap} \circ (\underline{f} \circ \mathsf{bang} \vartriangle \mathsf{id}) \qquad \text{From-Point}$$

2.1 Implementation of the Basic Combinators

Most of the basic functions, combinators, and types presented above are already part of the Haskell 98 standard prelude [14]. However, there is a subtle difference in the definition of functions like id or fst. These were defined as families of monomorphic functions, indexed by the specific

type they operate on. Since most of the times this type can be easily inferred from the context, the indexes are usually omitted. The Haskell type system is polymorphic, which means that a single definition can be given for these function.

There is also a polymorphic bottom value predefined, namely `undefined`. Since it will be used often, it is convenient to define a shorter alias, with the advantage that it graphically resembles the mathematical notation.

```
_L :: a
_L = undefined
```

Using the standard Haskell 98 it is not possible to define a type that implements the terminal object, because any type declaration must have at least one constructor. The best approach would be to use the special predefined unit data type `()`. However, since it still has two elements, namely `()` and `_L`, it is not the terminal object of our semantic domain. The same discussion applies to any isomorphic data type with a single constructor without parameters. The problem can be solved by resorting to the use of Haskell extensions, since a data type without constructors can be declared.

```
data One
```

The only element of this data type is `_L` and as such it is indeed a terminal object. `bang` and the combinator $\underline{\cdot}$, that converts elements into points, can be implemented as follows. Due to Haskell limitations the syntactic notation must be compromised.

```
bang :: a -> One
bang _ = _L

pnt :: a -> One -> a
pnt x = \_ -> x
```

It is well known that Haskell semantics differs from the standard **CPO** denotational semantics, since all data types are by default pointed and lifted (every type has a distinct bottom element). This means that Haskell does not have true categorical products because `(_L,_L) /= _L`, nor true categorical exponentials because `(\_ -> _L) /= _L`. Concerning products, any function defined using pattern matching, such as `\(_,_) -> 0`, can distinguish between `(_L,_L)` and `_L`. For exponentials, the examples are more subtle and typically involve using the standard `seq` function. As discussed in [8], this fact complicates equational reasoning because the standard laws about products and functions no longer hold.

In the context of point-free programming this problem can be alleviated by prohibiting the use of `seq`. For products, since no pattern matching is used, values should be inspected using the predefined projections `fst` and `snd` (that cannot distinguish between `(_L,_L)` and `_L`), and thus we can "pretend" to have unlifted products. The same applies to exponentiation, since standard function application cannot distinghish between `\_ -> _L` and `_L`.

The infix split and product combinators can be defined as follows.

```
infix 6  /\
(/\) :: (a -> b) -> (a -> c) -> a -> (b,c)
(/\) f g x = (f x, g x)

infix 7  ><
(><) :: (a -> b) -> (c -> d) -> (a,c) -> (b,d)
f >< g = f . fst /\ g . snd
```

Unlike product, the separated sum is by definition lifted, so there is no problem in representing it by a Haskell data type. The predefined `Either` data type is used. The following new aliases are defined for the constructors.

```
inl :: a -> Either a b
inl = Left

inr :: b -> Either a b
inr = Right
```

The infix alias either and sum combinators are also defined.

```
infix 4 \/
(\/) :: (b -> a) -> (c -> a) -> Either b c -> a
(\/) f _ (Left x)  = f x
(\/) _ g (Right y) = g y

infix 5 -|-
(-|-) :: (a -> b) -> (c -> d) -> Either a c -> Either b d
f -|- g = inl . f \/ inr . g
```

Concerning exponentiation, the curry combinator is predefined but we need an uncurried version of the application function.

```
app :: (a -> b, a) -> b
app (f,x) = f x
```

An explicit exponentiation combinator is not defined because it just corresponds to the left-sectioning of the composition operator, with the additional advantage of a similar graphical notation.

Equipped with these definitions, point-free expressions can be directly translated to Haskell. The examples presented in the previous section can be encoded as follows.

```
swap :: (a,b) -> (b,a)
swap = snd /\ fst

distr :: (c, Either a b) -> Either (c,a) (c,b)
distr = (swap -|- swap) . distl . swap
    where distl = app . ((curry inl \/ curry inr) >< id)

undistr :: Either (c,a) (c,b) -> (c, Either a b)
undistr = (id >< inl) \/ (id >< inr)
```

3 From Pointwise to Point-free

The well-known equivalence between simply-typed λ-calculus (with pairs and terminal type) and cartesian closed categories was first stated by Lambek [18]. One half of this correspondence is testified by a translation from pointwise terms to categorical combinators, that was later used by Curien to define a new implementation technique for functional languages – the *categorical abstract machine* [7]. This translation is also the starting point for our point-free derivation mechanism. In this section we show how it can be extended to handle sums. Recursion will be handled in the next section.

3.1 A Simply Typed λ-calculus

We start by defining a point-wise language which is essentially a λ-calculus with product and sum types, as well as terminal type. The types and terms of the language are given by the following grammar. We reuse the keywords for denoting the projections and the injections.

$$\begin{aligned} A, B &::= 1 \mid \tau \mid A \to B \mid A \times B \mid A + B \\ M, N &::= * \mid x \mid c \mid M\ N \mid \lambda x : A.\ M \mid \\ &\quad \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \\ &\quad \mathsf{case}\ M\ \mathsf{of}\ x \to N;\ y \to N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \end{aligned}$$

τ ranges over base types; x stands for a variable and c for a constant of type $\Delta(c)$; $\mathsf{fst}\ M, \mathsf{snd}\ M$ are projections from a product type and $\mathsf{inl}\ M, \mathsf{inr}\ M$ are injections into a sum type. $\langle \cdot, \cdot \rangle$ is a pairing construct and case allows to perform case-analysis on sums. Finally, $*$ is the unique inhabitant of the terminal type. The usual notions of free and bound

variable are defined on terms. $\mathsf{FV}(M)$ denotes the set of free variables in M, and $M[N/x]$ the capture-avoiding substitution of N for the free occurrences of x in M.

As an example of using this λ-calculus, we define both swap and distr in the point-wise style.

$$\begin{aligned}\mathsf{swap} &: A \times B \to B \times A\\ \mathsf{swap} &= \lambda x.\langle \mathsf{snd}\ x, \mathsf{fst}\ x\rangle\end{aligned}$$

$$\begin{aligned}\mathsf{distr} &: A \times (B + C) \to (A \times B) + (A \times C)\\ \mathsf{distr} &= \lambda x.\mathsf{case}\ (\mathsf{snd}\ x)\ \mathsf{of}\ y \to \mathsf{inl}\ \langle \mathsf{fst}\ x, y\rangle;\\ &\qquad\qquad\qquad\qquad z \to \mathsf{inr}\ \langle \mathsf{fst}\ x, z\rangle\end{aligned}$$

3.2 The Translation

The translation from the typed λ-calculus (pointwise) to the internal language of a cartesian closed category (point-free) is rather ingenious. It is detailed in many text books on the subject, for instance [19, **?**]. The way variables are handled resembles the translation of the lambda calculus into the *de Bruijn notation* [9], where variables are represented by integers that measure the distance to the abstraction were they where bound. There are some suggestions about how it can be extended to cover sums in [5], but the translation is not fully detailed and its soundness is not proved. The approach presented here is slightly different.

The translation keeps track of variables by imposing some additional structure on the typing contexts. These will now be represented by left-nested pairs, defined by the grammar

$$\Gamma ::= \epsilon \mid \langle \Gamma, x : A\rangle$$

with x a variable and A a type. In point-free terms, each variable is replaced by the path to its position in the context tuple, given as follows

$$\mathsf{path}(\langle c, y\rangle, x) = \begin{cases}\mathsf{snd} & \text{if } x = y\\ \mathsf{path}(c, x) \circ \mathsf{fst} & \text{otherwise}\end{cases}$$

The translation function, denoted Φ, is overloaded and defined on types, typing contexts, and point-wise terms. It is assumed that each base type τ is represented by an object in the category, denoted by $\Phi(\tau)$. For the remaining types the translation into an object is defined as expected.

$$\begin{aligned}\Phi(1) &= 1\\ \Phi(A \times B) &= \Phi(A) \times \Phi(B)\\ \Phi(A + B) &= \Phi(A) + \Phi(B)\\ \Phi(A \to B) &= \Phi(B)^{\Phi(A)}\end{aligned}$$

Analogously, the object that represents a typing context is generated as follows.

$$\begin{aligned}\Phi(\epsilon) &= 1\\ \Phi(\langle \Gamma, x : A\rangle) &= \Phi(\Gamma) \times \Phi(A)\end{aligned}$$

The translation from λ-terms to arrows in the category operates on typing judgments. A judgment $\Gamma \vdash M : A$ will be translated as

$$\Phi(\Gamma \vdash M : A) : \Phi(\Gamma) \to \Phi(A)$$

according to the following rules. In order to simplify the presentation, type information is omitted.

$$\begin{aligned}
&\Phi(\Gamma \vdash \star) && = \mathsf{bang}\\
&\Phi(\Gamma \vdash c) && = \underline{c} \circ \mathsf{bang}\\
&\Phi(\Gamma \vdash x) && = \mathsf{path}(\Gamma, x)\\
&\Phi(\Gamma \vdash MN) && = \mathsf{ap} \circ (\Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N))\\
&\Phi(\Gamma \vdash \lambda x.M) && = \overline{\Phi(\langle \Gamma, x\rangle \vdash M)}\\
&\Phi(\Gamma \vdash \langle M, N\rangle) && = \Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N)\\
&\Phi(\Gamma \vdash \mathsf{fst}\ M) && = \mathsf{fst} \circ \Phi(\Gamma \vdash M)\\
&\Phi(\Gamma \vdash \mathsf{snd}\ M) && = \mathsf{snd} \circ \Phi(\Gamma \vdash M)\\
&\Phi(\Gamma \vdash \mathsf{inl}\ M) && = \mathsf{inl} \circ \Phi(\Gamma \vdash M)\\
&\Phi(\Gamma \vdash \mathsf{inr}\ M) && = \mathsf{inr} \circ \Phi(\Gamma \vdash M)\\
&\Phi(\Gamma \vdash \mathsf{case}\ L\ \mathsf{of}\ x \to M;\ y \to N) && =\\
&\qquad \mathsf{ap} \circ (\mathsf{either} \circ (\Phi(\Gamma \vdash \lambda x.M) \vartriangle \Phi(\Gamma \vdash \lambda y.N)) \vartriangle \Phi(\Gamma \vdash L))
\end{aligned}$$

We will now present some examples of translating terms in the cartesian closed fragment and delay the explanation of case translation for a moment.

The translation of a closed term $M : A \to B$ is the point that represents it in the category, that is, a morphism of type $1 \to \Phi(B)^{\Phi(A)}$. As seen in Section 2, since the category is cartesian closed, this point can be converted into the expected morphism of type $A \to B$ using From-Point. As such, for closed terms of functional type the translation is defined as follows.

$$\mathsf{ap} \circ (\Phi(\epsilon \vdash M) \circ \mathsf{bang} \vartriangle \mathsf{id})$$

For example, for the identity function we have

$$\Phi(\epsilon \vdash \lambda x : A.x) = \overline{\mathsf{snd}} : 1 \to A^A$$

Since this is a closed term of functional type, it is possible to convert the result to a morphism of type $A \to A$ and prove that it is indeed equivalent to id.

$$
\begin{array}{ll}
& \mathsf{ap} \circ (\overline{\mathsf{snd}} \circ \mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor} \,\} \\
& \mathsf{ap} \circ (\overline{\mathsf{snd}} \times \mathsf{id}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Exp\text{-}Cancel} \,\} \\
& \mathsf{snd} \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
& \mathsf{id}
\end{array}
$$

For the swap function we get the following translation.

$$\Phi(\epsilon \vdash \mathsf{swap}) = \overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} : 1 \to B \times A^{A \times B}$$

Again, since it is of functional type, some simple calculations show that it is equivalent to the expected definition.

$$
\begin{array}{ll}
& \mathsf{ap} \circ (\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} \circ \mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor} \,\} \\
& \mathsf{ap} \circ (\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} \times \mathsf{id}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Exp\text{-}Cancel} \,\} \\
& (\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
& \mathsf{snd} \circ \mathsf{snd} \circ (\mathsf{bang} \vartriangle \mathsf{id}) \vartriangle \mathsf{fst} \circ \mathsf{snd} \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
& \mathsf{snd} \vartriangle \mathsf{fst}
\end{array}
$$

Concerning the translation of the case construct, notice that, using a mixed pointwise and point-free style, $\mathsf{case}\ L\ \mathsf{of}\ x \to M;\ y \to N$ is equivalent to $(\lambda x.M \triangledown \lambda y.N)L$. This equivalence exposes the fact that a case is just an instance of application, and as such its translation exhibits the same top level structure:

$$\mathsf{ap} \circ \Phi(\Gamma \vdash \lambda x.M \triangledown \lambda y.N) \vartriangle \Phi(\Gamma \vdash L)$$

The question remains of how to combine $\Phi(\Gamma \vdash \lambda x.M) : \Gamma \to C^A$ and $\Phi(\Gamma \vdash \lambda y.N) : \Gamma \to C^B$ using either. Our solution is based on the internalization of the uncurried version of this combinator, that can be defined in point-free as follows.

$$
\begin{array}{l}
\mathsf{either}\ :\ C^A \times C^B \to C^{A+B} \\
\mathsf{either} = \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr}}
\end{array}
$$

In order to exemplify the translation of sums, consider its application to the pointwise definition of coswap.

$$
\begin{array}{l}
\mathsf{coswap}\ :\ A + B \to B + A \\
\mathsf{coswap} = \lambda x.\mathsf{case}\ x\ \mathsf{of}\ y \to \mathsf{inr}\ y;\ z \to \mathsf{inl}\ z
\end{array}
$$

The following result is obtained.

$$\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inr} \circ \mathsf{snd}} \vartriangle \overline{\mathsf{inl} \circ \mathsf{snd}}) \vartriangle \mathsf{snd})} : 1 \to B + A^{A+B}$$

To show that this expression corresponds to the expected definition we need the following result concerning either the proof of which can be found in [6].

$$\mathsf{either} \circ (\overline{f \circ \mathsf{snd}} \vartriangle \overline{g \circ \mathsf{snd}}) = \overline{(f \triangledown g) \circ \mathsf{snd}} \qquad \text{Either-Const}$$

The simplification can now be carried out as follows.

$$\begin{bmatrix} & \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inr} \circ \mathsf{snd}} \vartriangle \overline{\mathsf{inl} \circ \mathsf{snd}}) \vartriangle \mathsf{snd})} \circ \mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\, \text{Prod-Absor, Exp-Cancel} \,\} \\ & \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inr} \circ \mathsf{snd}} \vartriangle \overline{\mathsf{inl} \circ \mathsf{snd}}) \vartriangle \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\, \text{Either-Const} \,\} \\ & \mathsf{ap} \circ (\overline{(\mathsf{inr} \triangledown \mathsf{inl}) \circ \mathsf{snd}} \vartriangle \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\, \text{Prod-Absor, Exp-Cancel} \,\} \\ & (\mathsf{inr} \triangledown \mathsf{inl}) \circ \mathsf{snd} \circ (\mathsf{id} \vartriangle \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\, \text{Prod-Cancel} \,\} \\ & \mathsf{inr} \triangledown \mathsf{inl} \end{bmatrix}$$

A more substantial example is the translation of the point-wise version of distr presented in Section 3.1. The (very long) result together with the calculation that shows its equivalence to distr is shown in Figure 1.

It can be shown that the translation Φ is sound, i.e, all equivalences proved with an equational theory for the λ-calculus can also be proved in the categorical setting using the equations of Section 2. For the cartesian closed subset see for instance [7, 20]. The fundamental result is that the notion of substitution is replaced by composition. For the soundnesse of the sum translation see [6].

4 Programming With Recursion Patterns

Meijer, Fokkinga, and Paterson pioneered the study of recursion patterns in **CPO** [23]. Besides presenting traditional folds and unfolds, they also introduced a new pattern, called *hylomorphism*, that was later proved to be powerful enough to allow for the definition of any fixpoint [24]. Hylomorphisms will here be used to replace explicit recursion in pointwise definitions. Before presenting the translation of fixpoints, this section briefly reviews the categorical approach to recursive data types and hylomorphisms. In the next section we show how these concepts are incorporated in the Pointless Haskell library.

$$
\begin{array}{ll}
\lceil & \mathsf{ap} \circ (\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \circ \mathsf{fst} \triangle \mathsf{snd})} \triangle \overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \circ \mathsf{fst} \triangle \mathsf{snd})}) \triangle \mathsf{snd} \circ \mathsf{snd}) \circ \mathsf{bang} \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel},\ \mathsf{Prod\text{-}Def} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \times \mathsf{id})} \triangle \overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \times \mathsf{id})}) \triangle \mathsf{snd} \circ \mathsf{snd}) \circ (\mathsf{bang} \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion},\ \mathsf{Prod\text{-}Fusion} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl}} \triangle \overline{\mathsf{inr}}) \circ \mathsf{fst} \circ \mathsf{snd} \triangle \mathsf{snd} \circ \mathsf{snd}) \circ (\mathsf{bang} \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Cancel},\ \mathsf{Prod\text{-}Def} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl}} \triangle \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{either\ definition} \,\} \\
& \mathsf{ap} \circ ((\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ (\overline{\mathsf{inl}} \triangle \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \times \text{ functor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
& (\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ ((\overline{\mathsf{inl}} \triangle \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, + \text{ functor},\ \mathsf{distr}\ \text{natural} \,\} \\
& (\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ ((\overline{\mathsf{inl}} \triangle \overline{\mathsf{inr}}) \times \mathsf{id} + (\overline{\mathsf{inl}} \triangle \overline{\mathsf{inr}}) \times \mathsf{id}) \circ \mathsf{distr} \\
= & \quad \{\, + \text{ functor},\ \times \text{ functor},\ \mathsf{Prod\text{-}Cancel} \,\} \\
& (\mathsf{ap} \triangledown \mathsf{ap}) \circ (\overline{\mathsf{inl}} \times \mathsf{id} + \overline{\mathsf{inr}} \times \mathsf{id}) \circ \mathsf{distr} \\
= & \quad \{\, \mathsf{Sum\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
& (\mathsf{inl} \triangledown \mathsf{inr}) \circ \mathsf{distr} \\
= & \quad \{\, \mathsf{Sum\text{-}Reflex} \,\} \\
\lfloor & \mathsf{distr}
\end{array}
$$

Fig. 1. Translation of distr

4.1 Recursive Data Types Categorically

Consider the following Haskell declarations of natural numbers and polymorphic lists.

```
data Nat = Zero | Succ Nat
data List a = Nil | Cons a (List a)
```

In order to present a general theory for data types, it is first necessary to circumvent some "irregularities" in constructor declaration, namely, the fact that there may exist an arbitrary number of constructors, and that each may have an arbitrary number of arguments. This last problem is easily solved by treating constants as functions with domain 1, and by uncurrying constructors with more than one parameter. For naturals and lists this technique can be illustrated by the following declarations.

$$
\begin{array}{ll}
\mathsf{zero} : 1 \to \mathsf{Nat} & \mathsf{nil} : 1 \to \mathsf{List}\ A \\
\mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} & \mathsf{cons} : A \times \mathsf{List}\ A \to \mathsf{List}\ A
\end{array}
$$

All the constructors of a data type share the same target type. As such, the either combinator can be used to pack all of them in a single declaration, as in the following declaration for naturals.

$$\mathsf{zero} \triangledown \mathsf{succ} : 1 + \mathsf{Nat} \to \mathsf{Nat}$$

Since the domain is an expression involving the target type, the categorical concept of functor can be used to factor this type out. A *functor* F is a mapping between categories that preserves compositions and identities. *Endofunctors* in **CPO** will be used, mapping types to types, and functions to functions. The basic set of functors includes the identity functor Id, whose action on types is defined as $\mathsf{Id}\ A = A$, and on functions as $\mathsf{Id}\ f = f$. It also includes the constant functor: given a type A, the functor $\underline{A}$ is defined on types as $\underline{A}\ B = A$, and on functions as $\underline{A}\ f = \mathsf{id}_A$. Analogously, a *bifunctor* $\star$ maps pairs of types to types, and pairs of functions to functions. Given two monofunctors F and G and a bifunctor $\star$, a new monofunctor $F\ \hat{\star}\ G$ can be defined by *lifting* $\star$ as follows.

$$(F\ \hat{\star}\ G)\ A = (F\ A) \star (G\ A)$$
$$(F\ \hat{\star}\ G)\ f = (F\ f) \star (G\ f)$$

The packed representation of the constructors of a data type T will be denoted by in_T, and the *base functor* that captures its signature by F_T. Notice that with this approach, the type of in_T is always $\mathsf{F}_T\ T \to T$. For polymorphic data types the type variables will be omitted in subscripts in order to improve readability.

$$\mathsf{F}_{\mathsf{Nat}} = \underline{1}\ \hat{+}\ \mathsf{Id} \qquad \mathsf{F}_{\mathsf{List}} = \underline{1}\ \hat{+}\ \underline{A}\ \hat{\times}\ \mathsf{Id}$$
$$\mathsf{in}_{\mathsf{Nat}} = \mathsf{zero} \triangledown \mathsf{succ} \qquad \mathsf{in}_{\mathsf{List}} = \mathsf{nil} \triangledown \mathsf{cons}$$

A recursive data type T is then defined by taking the fixed point of its base functor F_T. Reynolds proved that in **CPO**, given a locally continuous and strictness-preserving base functor F, there exists a unique data type $T = \mu F$ and two unique strict functions $\mathsf{in}_T : F\ T \to T$ and $\mathsf{out}_T : T \to F\ T$ that are each other's inverse [27]. Fokkinga and Meijer [10] showed that all polynomial, and even all regular functors, are locally continuous and strictness-preserving. A *polynomial functor* is either the identity functor, a constant functor, a lifting of the sum and product bifunctors, or the composition of polynomial functors. A *regular functor* can also be built from type functors. This guarantees that, for example, all the above data types are well defined.

$$\mathsf{Nat} = \mu(\mathsf{F}_{\mathsf{Nat}}) \qquad \mathsf{List}\ A = \mu(\mathsf{F}_{\mathsf{List}})$$

4.2 Hylomorphisms

Given a functor F, a function $g : F\ B \to B$, and a function $h : A \to F\ A$, a hylomorphism is defined as the following recursive function, using the

fixpoint operator μ.

$$\begin{aligned}&[\![g, h]\!]_{\mu F} \;:\; A \to B\\ &[\![g, h]\!]_{\mu F} = \mu(\lambda f.g \circ F f \circ h)\end{aligned}$$

The main advantage of expressing recursive functions as hylomorphisms is that they have several interesting laws appropriate for program calculation and transformation. For example, by unfolding the fixpoint operator we immediately get the following cancellation law.

$$[\![g, h]\!]_{\mu F} = g \circ F\ [\![g, h]\!]_{\mu F} \circ h \qquad \textsf{Hylo-Cancel}$$

From this law, it is clear that the recursion pattern of the hylomorphism is characterized by the functor F. For example, if this functor is $\underline{1} \mathbin{\hat{+}} \mathsf{Id}$ then the resulting definition is necessarily linear recursive. To define a birecursive function a second degree polynomial functor, such as $\underline{1} \mathbin{\hat{+}} \mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id}$, must be used. In fact, the recursion tree of a function defined as a hylomorphism is modeled by μF.

Function h is responsible for all computations prior to recursion, namely, to compute the values passed to the recursive calls. Function g combines the results of the recursive calls in order to compute the final result. Notice that some values can be passed intact from h to g. This will be the case when a functor modeling a data type that stores some information in the nodes is used, like $\underline{1} \mathbin{\hat{+}} \underline{A} \mathbin{\hat{\times}} \mathsf{Id}$ for the case of lists.

Most of the fundamental laws about hylomorphisms follow directly from similar laws about fixpoints, or can be proved by fixpoint induction. That is the case of the following fusion law.

$$\begin{gathered}g \circ [\![h, i]\!]_{\mu F} \circ j = [\![k, l]\!]_{\mu F}\\ \Leftarrow\\ g \text{ strict} \;\wedge\; g \circ h = k \circ F\ g \;\wedge\; i \circ j = F\ j \circ l\end{gathered} \qquad \textsf{Hylo-Fusion}$$

Another important law about this recursion pattern is the shifting law, that can be used to change the shape of recursion.

$$[\![g \circ \eta, h]\!]_{\mu F} = [\![g, \eta \circ h]\!]_{\mu G} \quad \Leftarrow \quad \eta : F \mathrel{\dot{\to}} G \qquad \textsf{Hylo-Shift}$$

The side condition $\eta : F \mathrel{\dot{\to}} G$ requires that η is a *natural transformation* between F and G. That is the case if η is a function that assigns to each type A an arrow $\eta_A : F\ A \to G\ A$ such that, for any function $f : A \to B$ the following naturality condition holds.

$$G\ f \circ \eta_A = \eta_B \circ F\ f$$

A classic example of hylomorphism is the factorial function. It can be defined using a list of naturals as recursion tree. $\mathsf{one} = \mathsf{succ} \circ \mathsf{zero}$ and $\mathsf{mult} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ implements multiplication.

$$\begin{aligned}&\mathsf{fact} \;:\; \mathsf{Nat} \to \mathsf{Nat}\\ &\mathsf{fact} = [\![\mathsf{one} \triangledown \mathsf{mult}, (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{List\ Nat}}\end{aligned}$$

This definition can be manipulated as follows.

$$\begin{aligned}&\mathsf{fact} = [\![\mathsf{one} \triangledown \mathsf{mult}, (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}]\!]_{\mathsf{List\ Nat}}\\ \Leftrightarrow\quad &\{\,\mathsf{Hylo\text{-}Cancel}\,\}\\ &\mathsf{fact} = (\mathsf{one} \triangledown \mathsf{mult}) \circ (\mathsf{id} + \mathsf{id} \times \mathsf{fact}) \circ (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}\\ \Leftrightarrow\quad &\{\,\mathsf{out} \circ \mathsf{in} = \mathsf{id}\,\}\\ &\mathsf{fact} \circ \mathsf{in} = (\mathsf{one} \triangledown \mathsf{mult}) \circ (\mathsf{id} + \mathsf{id} \times \mathsf{fact}) \circ (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id})\\ \Leftrightarrow\quad &\{\,\mathsf{in}_{\mathsf{Nat}} = \mathsf{zero} \triangledown \mathsf{succ},\ \mathsf{Sum\text{-}Absor}\,\}\\ &\mathsf{fact} \circ (\mathsf{zero} \triangledown \mathsf{succ}) = \mathsf{one} \triangledown \mathsf{mult} \circ (\mathsf{id} \times \mathsf{fact}) \circ (\mathsf{succ} \vartriangle \mathsf{id})\\ \Leftrightarrow\quad &\{\,\mathsf{Sum\text{-}Fusion},\ \mathsf{fact}\ \text{strict},\ \mathsf{Prod\text{-}Absor}\,\}\\ &\mathsf{fact} \circ \mathsf{zero} \triangledown \mathsf{fact} \circ \mathsf{succ} = \mathsf{one} \triangledown \mathsf{mult} \circ (\mathsf{succ} \vartriangle \mathsf{fact})\end{aligned}$$

This calculation means that the above definition satisfies the following equations.

$$\mathsf{fact} \circ \mathsf{zero} = \mathsf{one} \;\wedge\; \mathsf{fact} \circ \mathsf{succ} = \mathsf{mult} \circ (\mathsf{succ} \vartriangle \mathsf{fact})$$

By applying the definitions of composition and split in order to recover variables, we can see that it corresponds to the expected Haskell definition.

```
fact :: Nat -> Nat
fact Zero     = Succ Zero
fact (Succ n) = mult (Succ n, fact n)
```

As said above, hylomorphisms are expressive enough to implement all the remaining typical recursion patterns. One of the fundamental patterns of recursion is iteration, where recursive data types are "consumed" by replacing their constructors by arbitrary functions. This recursion pattern is usually called fold or *catamorphism*. In Haskell this pattern is predefined for lists as the function `foldr`. Given a function of type $g : F\ A \to A$, the catamorphism operator which implements iteration over the data type μF can be generically defined as follows.

$$\begin{aligned}&(\!|g|\!)_{\mu F} \;:\; \mu F \to A\\ &(\!|g|\!)_{\mu F} = [\![g, \mathsf{out}_F]\!]_{\mu F}\end{aligned}$$

A well-know example of a catamorphism is the length function over lists.

$$\begin{aligned}&\mathsf{length} \;:\; \mathsf{List}\ A \to \mathsf{Nat}\\ &\mathsf{length} = (\!|\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd})|\!)_{\mathsf{List}\ A}\end{aligned}$$

A more advanced recursion pattern is the *paramorphism* [22]. It encodes primitive recursion, which means that both the recursive call on a substructure of the input and the substructure itself can be used to compute the result. The definition of a paramorphism as a hylomorphism is known at least since [23]. Unlike the definition of catamorphism, the intermediate data type will no longer be equal to the data type being consumed. Given an input of type μF, the functor that generates the intermediate data structure is $F \circ (\mathsf{Id} \mathbin{\hat{\times}} \underline{\mu F})$: every recursive occurrence of the original type is replaced by a new recursive occurrence and a copy of the older one that will be left intact when recursing. For example, a paramorphism over naturals will have as intermediate data structure an element of type $\mu(\underline{1} \mathbin{\hat{+}} \mathsf{Id} \mathbin{\hat{\times}} \underline{\mathsf{Nat}})$, which is isomorphic to a list of naturals.

Given a function $g : F\ (A \times \mu F) \to A$, a paramorphism parameterized by g can be generically defined using a hylomorphism as follows. Notice the use of the doubling combinator $(\mathsf{id} \vartriangle \mathsf{id})$ to replicate the substructures of the input value.

$$\begin{aligned} &(\!|g|\!)_{\mu F} \ :\ \mu F \to A \\ &(\!|g|\!)_{\mu F} = [\![g, F\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mu F}]\!]_{\mu(F \circ (\mathsf{Id} \hat{\times} \underline{\mu F}))} \end{aligned}$$

The most classic example of a paramorphism is the factorial function: for a nonzero parameter n, it uses both the recursive result $\mathsf{fact}\ (n-1)$ and the parameter n itself to compute the result.

$$\begin{aligned} &\mathsf{fact}\ :\ \mathsf{Nat} \to \mathsf{Nat} \\ &\mathsf{fact} = (\!|\mathsf{one} \triangledown \mathsf{mult} \circ (\mathsf{id} \times \mathsf{succ})|\!)_{\mathsf{Nat}} \end{aligned}$$

5 Pointless Haskell

At least since [24], it has been known how to implement generic versions of the recursion patterns in Haskell by defining data types explicitly as fixed points of functors. The implementation follows directly from the theoretical concepts presented in Section 4.1. This style of programming was used to implement generic recursion patterns by several authors [28, 11], and is also followed, with some improvements, in our library.

The explicit fixpoint operator can be defined at the type level using `newtype`.

```
newtype Functor f => Mu f = Mu {unMu :: f (Mu f)}
```

The context of the definition restricts the application of `Mu` to members of the `Functor` class. The use of `newtype` guarantees the strictness of `Mu`,

and thus enforces the isomorphism between `Mu f` and `f (Mu f)`. `inn` and `out` can be defined as aliases to the constructor and the destructor of `Mu`. Using `Mu` the data type of naturals can be defined in Haskell as follows.

```
newtype FNat x = FNat {unFNat :: Either One x}
instance Functor FNat
    where fmap f = FNat . (id -|- f) . unFNat
type Nat = Mu FNat

zero = inn . FNat . inl
succ = inn . FNat . inr
```

For parameterized data types, like lists, the base functor is obtained from a binary type constructor by treating the first type variable as a constant.

```
newtype FList a x = FList {unFList :: Either () (a,x)}
instance Functor (FList a)
    where fmap f = FList . (id -|- id >< f) . unFList
type List a = Mu (FList a)
```

Using this style of programming polytypism comes for free, since the fundamental recursion operators given in Section 4.2 can be generically defined as follows.

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo g h = g . fmap (hylo g h) . h

cata :: Functor f => (f a -> a) -> Mu f -> a
cata g = hylo g out
```

Given these operators, the length and factorial functions can be defined in a straightforward way.

```
len :: List a -> Nat
len = cata g
    where g = inn . FNat . (id -|- snd) . unFList

fact :: Nat -> Nat
fact = hylo g h
    where h = FList . (id -|- succ /\ id) . unFNat . out
          g = (succ . zero \/ mult) . unFList
```

In order to define paramorphisms as hylomorphisms, the concept of *functor transformer* can be used. A functor transformer is a type constructor with kind $(\star \to \star) \to (\star \to \star)$, that given a functor returns another functor, and will be used to capture the functor change that occurs in that definition. It has been used in a similar context in [28]. Recall the definition of paramorphisms: for a data type μF, the functor that

captures the shape of recursion in the hylomorphism that implements it is $F \circ (Id \hat{\times} \underline{\mu F})$. This specific functor change can be captured by the following transformer, defined in the pointwise style as a new data type.

```
newtype FPara f x = FPara {unFPara :: f (x, Mu f)}

instance Functor f => Functor (FPara f)
    where fmap f = FPara . fmap (f >< id) . unFPara
```

If the base functor of the input of a paramorphism is `f`, then the intermediate data structure of the hylomorphism that implements it is `Mu (FPara f)`. Using this transformer, paramorphisms can be defined according to the above definition.

```
para :: Functor f => (f (a, Mu f) -> a) -> Mu f -> a
para g = hylo (g . unFPara) h
    where h = FPara . fmap (id /\ id) . out
```

This approach has some disadvantages. First, since Haskell does not have structural type equivalence, coercing constructors and destructors are used often. Sometimes, this makes it difficult to translate a point-free definition to Haskell. To overcome this problem, one could define tailored instances of the recursion operators for each data type, as proposed in [11]. However, this would preclude polytypism, one of the main advantages of this approach. Second, it is impossible to use the recursion operators with the standard Haskell types, such as lists or integers. Finally, the `Functor` instances must be defined explicitly for every data type, when it is well known that the map function can be easily defined generically by induction on the structure of the type.

5.1 The PolyP Approach

Our solution to these problems is based on the generic programming library PolyP [25]. This library also views data types as fixed points of functors, but instead of using an explicit fixpoint operator, a multi-parameter type class [15] with a functional dependency [13] is used to relate a data type `d` with its base functor `f`. We remark that this is a non-standard Haskell feature provided as an extension. This class can be defined as follows.

```
class (Functor f) => FunctorOf f d | d -> f
  where inn' :: f d -> d out' :: d -> f d
```

The dependency means that different data types can have the same base functor, but one data type can have at most one base functor. The

main advantage of using `FunctorOf` is that predefined Haskell types can also be viewed as fixed points of functors. The use of the primes will be justified later.

We would like to stress that PolyP is not directly used in the implementation of Pointless Haskell. Some of its design choices would prevent the use of a syntax similar to the one described in the first section. As such, the relevant subset of PolyP was reimplemented according to our own design principles. For example, the `FunctorOf` class was simplified by restricting base functors to monofunctors (a parameterized type can still be defined using the left-sectioning of a bifunctor). The methods were reduced to the essential `in` and `out` functions.

Obviously, it is still possible to work with data types declared explicitly as fixed points of functors. For these, the instance of the `FunctorOf` class can be defined once and for all.

```
instance (Functor f) => FunctorOf f (Mu f)
    where inn' = Mu
          out' = unMu
```

To avoid the explicit definition of the map functions, functors are described using a fixed set of combinators instead of arbitrary data types. The combinators follow directly from the definition of regular functors: these include the identity and constant functors, the lifting of the sum and product bifunctors, and also the application of a functor to another functor. Infix constructors are used for these.

```
newtype Id x        = Id {unId :: x}
newtype Const t x   = Const {unConst :: t}
data (g :+: h) x    = Inl (g x) | Inr (h x)
data (g :*: h) x    = g x :*: h x
newtype (g :@: h) x = Comp {unComp :: g (h x)}
```

The `Functor` instances for these combinators are trivial and omitted here. Given this set of basic functors and functor combinators, there is no need to declare new functor data types to capture the recursive structure of a data type. Instead, they are declared using this basic set. For example, it is now possible to view the standard Haskell types for integers and lists as fixed points of functors.

```
instance FunctorOf (Const One :+: Id) Int
    where inn' (Inl (Const _)) = 0
          inn' (Inr (Id n))    = n+1
          out' 0     = Inl (Const _L)
          out' (n+1) = Inr (Id n)
```

```
instance FunctorOf (Const One :+: (Const a :*: Id)) [a]
    where inn' (Inl (Const _))           = []
          inn' (Inr (Const x :*: Id xs)) = x:xs
          out' []     = Inl (Const _L)
          out' (x:xs) = Inr (Const x :*: Id xs)
```

Unfortunately, this technique *per se* is not useful. The price to pay for not having to define the `Functor` instances is an enormous growth in the use of coercing constructors, rendering point-free programming almost impossible. That is the reason why the above instances are now defined in the pointwise style. This problem is solved by implementing a mechanism to perform implicit coercion between structurally equivalent data types, as described in the next section.

5.2 Implicit Coercion

To implement implicit coercion a multi-parameter type class is used.

```
class Rep a b | a -> b
    where to :: a -> b
          from :: b -> a
```

The first parameter should be a type declared using the basic set of functor combinators, and the second is the type that results after evaluating those combinators. The functional dependency imposes a unique result to evaluation. Unfortunately, a functional dependency from `b` to `a` does not exist because, for example, a type A can be the result of evaluating both $\mathsf{Id}\ A$ and $\underline{A}\ B$. The instances of `Rep` are also rather trivial. For example the identity and constant functors can be evaluated as follows.

```
instance Rep (Id a) a
    where to (Id x) = x
          from x = Id x

instance Rep (Const a b) a
    where to (Const x) = x
          from x = Const x
```

Given a bifunctor $\star$, the type that implements $(G \mathbin{\hat{\star}} H)\ A$ is $(G\ A) \star (H\ A)$. This means that, for the case of products and sums, the types that implement $G\ A$ and $H\ A$ should be computed prior to the resulting type. This evaluation order is guaranteed by using class constraints. The implementation for products is as follows.

```
instance (Rep (g a) b, Rep (h a) c)
             => Rep ((g :*: h) a) (b, c)
    where to (x :*: y) = (to x, to y)
          from (x, y) = from x :*: from y
```

To ensure that context reduction terminates, standard Haskell requires that the context of an instance declaration must be composed of simple type variables. In this example, although that condition is not verified, reduction necessarily terminates because contexts always get smaller. In order to force the compiler to accept these declarations, a non-standard type system extension must be activated with the option

```
-fallow-undecidable-instances
```

The implementations for the remaining functor combinators are similar. A possible interaction with a Haskell interpreter could now be

```
> to (Id 'a' :*: Const 'b')
('a','b')
> from ('a','b') :: (Id :*: Const Char) Char
Id 'a' :*: Const 'b'
```

Since the same standard Haskell type can represent different functor combinations, the expected result of the `from` function must be explicitly annotated. For example, another possible interaction could be

```
> from ('a','b') :: (Id :*: Id) Char
Id 'a' :*: Id 'b'
```

Since this type-checking problem would occur frequently, it was decided to annotate most of the polytypic functions with the functor to which they should be specialized. Types cannot be passed as arguments to functions, and so this is achieved indirectly through the use of a "dummy" argument and another non-standard Haskell feature, namely scoped type variables [16]. Since in Haskell only values of a concrete type (that is, of kind $\star$) can be passed as arguments, it is not possible to state directly the functor to which a function should be specialized. However, by using the type class `FunctorOf`, together with its functional dependency, it suffices to pass as argument a value of a data type that is the fixed point of the desired functor. Since recursive data types can still be defined explicitly using `Mu`, there is always a convenient choice for this parameter.

To start with, a polytypic map function is defined as follows.

```
pmap :: (FunctorOf f d, Rep (f a) fa, Rep (f b) fb) =>
        d -> (a -> b) -> (fa -> fb)
pmap (_::d) (f::a->b) =
    to . (fmap f :: FunctorOf f d => f a -> f b) . from
```

It is also useful to have the isomorphisms `in` and `out` with implicit coercion. In fact, this was the reason why the primes were used in the declaration of the `FunctorOf` class.

```
out :: (FunctorOf f d, Rep (f d) fd) => d -> fd
out = to . out'

inn :: (FunctorOf f d, Rep (f d) fd) => fd -> d
inn = inn' . from
```

A polytypic hylomorphism can be defined using `pmap`. Notice the use of bottom as the "dummy" argument to indicate the specific type to which a polytypic function should be instantiated.

```
hylo :: (FunctorOf f d, Rep (f b) fb, Rep (f a) fa) =>
        d -> (fb -> b) -> (a -> fa) -> a -> b
hylo mu g h = g . pmap mu (hylo mu g h) . h
```

This type annotation is essentially the same that was stated using a subscript in the theoretical notation. It is now possible to program with hylomorphisms in a truly point-free style. For example, the definition of factorial can now be transcribed directly to Haskell.

```
fact :: Int -> Int
fact = hylo (_L :: [Int]) f g
    where g = (id -|- succ /\ id) . out
          f = one \/ mult
```

The same applies to derived recursion patterns. Due to the ability to explicitly declare the intermediate data type as the fixed point of a functor it is no longer needed to define the functor transformers.

```
cata (_::d) f = hylo (_L::d) f out

para (_::d) f =
  hylo (_L::FunctorOf f d => Mu (f :@: (Id :*: Const d)))
       f (pmap (_L::d) (id /\ id) . out)
```

6 Translation of Recursive Definitions

We will now extend our λ-calculus with fixed points and show how they can be replaced by hylomorphisms. We assume that data types are defined using explicit fixed points of functors, as described in Section 4.1. The grammar is extended as expected.

$$M ::= \ldots \mid \mathsf{in}\ M \mid \mathsf{out}\ M \mid \mu\ M$$

For example, the length function can be defined as follows.

$$\begin{aligned}&\mathsf{length}\ :\ \mathsf{List}\ A \to \mathsf{Nat}\\ &\mathsf{length} = \mu(\lambda f.\lambda l.\mathsf{case}\ (\mathsf{out}\ l)\ \mathsf{of}\ x \to \mathsf{in}\ (\mathsf{inl}\ \star);\\ &\qquad\qquad\qquad\qquad\qquad\qquad y \to \mathsf{in}\ (\mathsf{inr}\ (f\ (\mathsf{snd}\ y))))\end{aligned}$$

The translation of the constructors and destructors to point-free is trivial.

$$\begin{aligned}\Phi(\Gamma \vdash \mathsf{in}\ M) &= \mathsf{in} \circ \Phi(\Gamma \vdash M)\\ \Phi(\Gamma \vdash \mathsf{out}\ M) &= \mathsf{out} \circ \Phi(\Gamma \vdash M)\end{aligned}$$

6.1 Direct Translation of Fixpoints

The first method for translating recursive definitions into hylomorphisms uses the encoding of the fixpoint operator a an hylomorphism, first proposed in [24]. The insight to this result is to notice that $\mu\ f$ is determined by the infinite application $f\ (f\ (f\ \ldots))$, whose recursion tree is a stream of functions f, subsequently consumed by application. Streams can be defined as

$$\mathsf{Stream}\ A = \mu(\underline{A} \mathbin{\hat{\times}} \mathsf{Id})$$

with a single constructor to insert an element at the head.

$$\mathsf{in} : A \times \mathsf{Stream}\ A \to \mathsf{Stream}\ A$$

Given a function f, the following hylomorphism encodes the fixpoint operator. It builds the recursion tree $\mathsf{in}\ (f, \mathsf{in}\ (f, \mathsf{in}\ (f, \ldots)))$, and then just replaces in by ap.

$$\begin{aligned}&\mathsf{fix}\ :\ A^A \to A\\ &\mathsf{fix} = [\![\mathsf{ap}, \mathsf{id} \vartriangle \mathsf{id}]\!]_{\mathsf{Stream}\ A^A}\end{aligned}$$

Using the Pointless library we get the following Haskell definition.

```
fix :: (a -> a) -> a
fix = hylo (_L::Mu(Const (a->a) :*: Id)) app (id /\ id)
```

Given this point-free definition, the translation of the fixpoint operator is now trivial.

$$\Phi(\Gamma \vdash \mu M) = \mathsf{fix} \circ \Phi(\Gamma \vdash M)$$

We now give a simple example of translating a recursive function defined over a recursive data type. Consider the function that given an element generates an infinite stream with copies of that value. In pointwise it can be defined as follows.

$$\begin{aligned}&\mathsf{repeat}\ :\ A \to \mathsf{Stream}\ A\\ &\mathsf{repeat} = \lambda x.\mu(\lambda y.\mathsf{in}\ \langle x, y\rangle)\end{aligned}$$

By applying the translation rules to this definition we get the following point-free expression. In this case, the intermediate data structure

of the hylomorphism that encodes fix is a stream of functions of type $\mathsf{Stream}\ A \to \mathsf{Stream}\ A$.

$$\overline{\mathsf{fix} \circ \overline{\mathsf{in} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{snd})}} : 1 \to (\mathsf{Stream}\ A)^A$$

It can be shown by calculation (details omitted here) that this expression is equivalent to the expected point-free hylomorphism.

$$\left[\begin{array}{l} \mathsf{ap} \circ (\overline{\mathsf{fix} \circ \overline{\mathsf{in} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{snd})}} \circ \mathsf{bang} \vartriangle \mathsf{id}) \\ = \quad \{\, \text{Calculations} \,\} \\ [\![\mathsf{in}, \mathsf{id} \vartriangle \mathsf{id}]\!]_{\mathsf{Stream}\ A} \end{array}\right.$$

6.2 Deriving Hylomorphisms

This last example was very contrived in order to make possible the calculation that shows its equivalence to the expected hylomorphism. However, with normal recursive definitions such calculations can be quite difficult. This is largely due to the fact that hylomorphisms are introduced only to encode the fixpoint operator, yielding definitions very different from those one would get if making the implementation ourselves. Ideally, one would like the resulting hylomorphisms to be more informative about the original function definition, in the sense that the intermediate data structure should model its recursion tree.

Hu, Iwasaki, and Takeichi have defined an algorithm that derives such hylomorphisms from explicitly recursive definitions [12]. This algorithm was developed to be used in the fusion system HYLO [26], and although it has several limitations (in particular it can not handle mutual or nested recursion), it covers most of the useful function definitions. In the present context, the idea is to use this algorithm in a stage prior to the point-free translation defined in Section 3. First, a pointwise hylomorphism is derived, and then the translation is applied to the functions that parameterize it. The main difference between the presentation given in [12] and the one given here lies in the underlying λ-calculus. While the original formulation allowed for user-defined types *a la* Haskell and general pattern matching, in our λ-calculus data types are declared as fixed points, and pattern matching is restricted to sums.

The hylomorphism derivation can be summarized as follows. Given a single-parameter recursive function defined using fixpoint

$$\mathsf{fix}(\lambda f.\lambda x.L) : A \to B$$

three transformations are defined: one to derive the functor that generates the recursion tree of the hylomorphism ($\mathcal{F}$), a second one to derive the

function that is invoked after recursion ($\mathcal{A}$), and another one for the function that is invoked prior to recursion ($\mathcal{C}$). The above function will be translated into the following hylomorphism.

$$[\![\lambda x.\mathcal{A}(L), \lambda x.\mathcal{C}(L)]\!]_{\mu(\mathcal{F}(L))} : A \to B$$

Some restrictions must be imposed on the syntax used to define recursive functions. The first is that the definition must be a closed expression, that is $\mathsf{FV}(L) = \{f, x\}$. This restriction guarantees that the parameters of the hylomorphism are also closed. If that was not the case, it would be necessary to propagate the typing context inside the hylomorphism. This can only be achieved by changing the intermediate data structure, and the translation would produce hylomorphisms as unmanageable as the ones obtained by directly encoding the fixpoint operator. The second restriction is that the body of the function to be translated must be defined as a kind of decision tree, implemented by case analysis on the input, whose leaves are the different possible outputs. The main syntactic restrictions are the absence of abstractions, which prevents the definition of some higher-order functions like accumulations, and the obligation that f always appears applied. For example the definition of repeat presented in the previous section is not covered by this algorithm. However, the following more natural definition of the same function or the length function defined above are examples of functions to which this algorithm can be applied.

$$\begin{array}{l}\mathsf{repeat}\ :\ A \to \mathsf{Stream}\ A\\ \mathsf{repeat} = \mathsf{fix}(\lambda f.\lambda x.\mathsf{in}\ \langle x, f\ x\rangle)\end{array}$$

The formal definition of the three transformations are presented in [6] and omitted here. Transformation $\mathcal{F}$ yields a summand for each path along the decision tree. Each summand signals the presence of a recursive invocation using the identity functor. It also has place-holders for the input-dependent information that should not be affected by recursion: since abstractions are not allowed, it suffices to introduce a constant functor of appropriate type for every variable outside a recursive invocation. Transformation $\mathcal{C}$ modifies the function body in order to put all variables and parameters of recursive calls in the appropriate place-holders. Finally, transformation $\mathcal{A}$ just replaces these by the appropriate paths to the recursion tree of the hylomorphism.

For example, using this algorithm the function repeat can be transformed into the point-wise hylomorphism

$$\mathsf{repeat} = [\![\lambda x.\mathsf{in}\ \langle \mathsf{fst}\ x, \mathsf{snd}\ x\rangle, \lambda x.\langle x, x\rangle]\!]_{\mu(\underline{A}\hat{\times}\mathsf{Id})}$$

$$
\begin{array}{ll}
 & [\![\mathsf{ap} \circ (\overline{\mathsf{in} \circ (\mathsf{fst} \circ \mathsf{snd} \vartriangle \mathsf{snd} \circ \mathsf{snd})} \circ \mathsf{bang} \vartriangle \mathsf{id}), \mathsf{ap} \circ (\overline{\mathsf{snd} \vartriangle \mathsf{snd}} \circ \mathsf{bang} \vartriangle \mathsf{id})]\!] \\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
 & [\![\mathsf{in} \circ (\mathsf{fst} \circ \mathsf{snd} \vartriangle \mathsf{snd} \circ \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id}), (\mathsf{snd} \vartriangle \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id})]\!] \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Cancel} \,\} \\
 & [\![\mathsf{in} \circ (\mathsf{fst} \vartriangle \mathsf{snd}), \mathsf{id} \vartriangle \mathsf{id}]\!] \\
= & \quad \{\, \mathsf{Prod\text{-}Reflex} \,\} \\
 & [\![\mathsf{in}, \mathsf{id} \vartriangle \mathsf{id}]\!]
\end{array}
$$

Fig. 2. repeat as a point-free hylomorphism

and length into the hylomorphism

$$
\begin{aligned}
\mathsf{length} = [\![& \lambda z.\mathsf{case}\ z\ \mathsf{of}\ x \to \mathsf{in}\ (\mathsf{inl}\ \star);\ y \to \mathsf{in}\ (\mathsf{inr}\ y), \\
 & \lambda l.\mathsf{case}\ (\mathsf{out}\ l)\ \mathsf{of}\ x \to \mathsf{inl}\ \star;\ y \to \mathsf{inr}\ (\mathsf{snd}\ y)]\!]_{\mu(\underline{1} \hat{+} \mathsf{Id})}
\end{aligned}
$$

The result of translating repeat into the point-free style, and the proof that it correspond to the expected definition, are presented in Figure 2.

7 Pattern Matching

Although the λ-calculus defined so far can be used to define most typical recursive functions, it is still much less convenient than the usual style in which Haskell functions are defined. In this section we briefly overview how to accommodate a limited form of pattern matching, over user-defined data types.

Concerning data types, it is well-known how to implement an algorithm for defining `FunctorOf` instances for almost any user-defined data type [25]. This means that it is possible to replace constructors by their equivalent fixpoint definitions, and thus it suffices to have pattern matching over the generic constructor in, sums, pairs, and the constant $\star$. We will introduce a new construct that implements such a mechanism, but with some limitations: there can be no repeated variables in the patterns, no overlapping, and the patterns must be exhaustive. It matches an expression against a set of patterns, binds all the variables in the matching one, and returns the respective right-hand side.

$$
M, N ::= \ldots \mid \mathsf{match}\ M\ \mathsf{with}\ \{P \to N;\ \ldots; P \to N\}
$$

The syntax of patterns is determined by the following grammar.

$$
P ::= \star \mid x \mid \langle P, P \rangle \mid \mathsf{in}\ P \mid \mathsf{inl}\ P \mid \mathsf{inr}\ P
$$

Using this construct, it is now possible to define functions using a syntax more similar to that of Haskell. For example, the swap and distr functions can be defined as follows.

$$\begin{aligned}&\mathsf{swap} \;:\; A \times B \to B \times A\\ &\mathsf{swap} = \lambda x.\mathsf{match}\ x\ \mathsf{with}\ \{\langle y, z\rangle \to \langle z, y\rangle\}\end{aligned}$$

$$\begin{aligned}\mathsf{distr} \;:\; & A \times (B + C) \to (A \times B) + (A \times C)\\ \mathsf{distr} = \lambda x.\mathsf{match}\ x\ \mathsf{with}\ \{&\langle y, \mathsf{inl}\ z\rangle \to \mathsf{inl}\ \langle y, z\rangle;\\ &\langle y, \mathsf{inr}\ z\rangle \to \mathsf{inr}\ \langle y, z\rangle\}\end{aligned}$$

Given the standard Haskell definition of length, it is now possible to have a almost direct translation into our λ-calculus by replacing `[]` and `(:)` by their fixpoint equivalent: $\mathsf{in}\ (\mathsf{inl}\ \star)$ and $\lambda xy.\mathsf{in}\ (\mathsf{inr}\ \langle x, y\rangle)$, respectively.

$$\begin{aligned}&\mathsf{length} \;:\; \mathsf{List}\ A \to \mathsf{Nat}\\ &\mathsf{length} = \mathsf{fix}(\lambda f.\lambda l.\mathsf{match}\ l\ \{&&\mathsf{in}\ (\mathsf{inl}\ \star) &&\to \mathsf{in}\ (\mathsf{inl}\ \star);\\ &&&\mathsf{in}\ (\mathsf{inr}\ \langle h, t\rangle) &&\to \mathsf{in}\ (\mathsf{inr}\ (f\ t))\})\end{aligned}$$

Instead of directly translating this new construct to point-free, a rewriting system is defined that eliminates generalized pattern-matching, and simplifies expressions back into the core λ-calculus previously defined. The rewriting system is presented in Figure 3. Matching over $\star$ succeeds trivially. Matching over a variable binds the variable and triggers a substitution in the right-hand side. In both cases there can only be one pattern in the set due to the non-overlapping constraint. For pairs, components are matched in turns. The chosen order is irrelevant, but after matching one projection with a specific pattern, the other one must only be matched against the pairing patterns. Some care must be taken in renaming variables in patterns in order to avoid variable capture. To match over a sum type case analysis is used. Due to the exhaustiveness requirement, the set of patterns can be partitioned into two disjoint sets, containing terms whose outermost constructor is inl and inr, respectively. Finally, when matching a value of a recursive type, the out function is used in order to expose its top level structure. Notice that this rewrite relation is guaranteed to terminate because the patterns always get smaller. Using this technique all the above examples are translated back into the respective definitions in the core λ-calculus.

Since Haskell does not have true products, this rewrite relation can sometimes produce expressions whose semantic behavior is different from the original. Consider the Haskell function `\(x,y) -> 0` that distinguishes

$$
\begin{array}{lcl}
\mathsf{match}\ M\ \mathsf{with}\ \{\star \to N\} & \rightsquigarrow & N \\
\mathsf{match}\ M\ \mathsf{with}\ \{x \to N\} & \rightsquigarrow & N[M/x] \\
\mathsf{match}\ M\ \mathsf{with}\ \{\mathsf{inl}\ P_{1,1} \to N_{1,1}; & & \mathsf{case}\ M\ \mathsf{of}\dot{x} \to \mathsf{match}\ x\ \mathsf{with}\{\ P_{1,1} \to N_{1,1}; \\
\quad \ldots & & \qquad \ldots \\
\quad \mathsf{inl}\ P_{1,i} \to N_{1,i}; & & \qquad P_{1,i} \to N_{1,i}\}; \\
\quad \mathsf{inr}\ P_{2,1} \to N_{2,1}; & \rightsquigarrow & \qquad \dot{y} \to \mathsf{match}\ y\ \mathsf{with}\ \{P_{2,1} \to N_{2,1}; \\
\quad \ldots & & \qquad \ldots \\
\quad \mathsf{inr}\ P_{2,j} \to N_{2,j}\} & & \qquad P_{2,j} \to N_{2,j}\} \\
\mathsf{match}\ M\ \mathsf{with}\ \{\mathsf{in}\ P_1 \to N_1; & & \mathsf{match}\ (\mathsf{out}\ M)\ \mathsf{with}\ \{P_1 \to N_1; \\
\quad \ldots & \rightsquigarrow & \qquad \ldots \\
\quad \mathsf{in}\ P_i \to N_i\} & & \qquad P_i \to N_i\} \\
 & & \mathsf{match}\ (\mathsf{fst}\ M)\ \mathsf{with}\ \{P_1 \to \mathsf{match}\ (\mathsf{snd}\ M) \\
\mathsf{match}\ M\ \mathsf{with}\ \{\langle P_1, Q_{1,1}\rangle \to N_{1,1}; & & \qquad \mathsf{with}\ \{Q_{1,1} \to N_{1,1}; \\
\quad \ldots & & \qquad \ldots \\
\quad \langle P_1, Q_{1,j}\rangle \to N_{1,j}; & & \qquad Q_{1,j} \to N_{1,j}\}; \\
\quad \ldots & \rightsquigarrow & \qquad \ldots \\
\quad \langle P_i, Q_{i,1}\rangle \to N_{i,1}; & & \qquad P_i \to \mathsf{match}\ (\mathsf{snd}\ M) \\
\quad \ldots & & \qquad \mathsf{with}\ \{Q_{i,1} \to N_{i,1}; \\
\quad \langle P_i, Q_{i,k}\rangle \to N_{i,k}\} & & \qquad \ldots \\
 & & \qquad Q_{i,k} \to N_{i,k}\}\}
\end{array}
$$

Fig. 3. Pattern matching elimination

`_L` from `(_L,_L)`. This function can be directly encoded using match and translated into the core λ-calculus using the following rewrite sequence.

$$
\begin{array}{ll}
 & \lambda z.\mathsf{match}\ z\ \mathsf{with}\ \{\langle x, y\rangle \to \mathsf{in}\ (\mathsf{inl}\ \star)\} \\
\rightsquigarrow & \lambda z.\mathsf{match}\ (\mathsf{fst}\ z)\ \mathsf{with}\ \{x \to \mathsf{match}\ (\mathsf{snd}\ z) \\
 & \qquad\qquad\qquad\qquad\qquad \mathsf{with}\ \{y \to \mathsf{in}\ (\mathsf{inl}\ \star)\}\} \\
\rightsquigarrow & \lambda z.\mathsf{match}\ (\mathsf{fst}\ z)\ \mathsf{with}\ \{x \to \mathsf{in}\ (\mathsf{inl}\ \star)\} \\
\rightsquigarrow & \lambda z.\mathsf{in}\ (\mathsf{inl}\ \star)
\end{array}
$$

Since it no longer has pattern matching, the resulting function is different from the original since it no longer distinguishes a bottom from a pair of bottoms. Apart from this problem, with this pattern matching construct it is now possible to translate into the point-free style many typical Haskell functions, such as the ones presented in Appendix A.

8 Conclusions and Future Work

In this paper we have presented a mechanism to translate a function defined in a core functional programming language into the point-free programming style. Although none of its components is completely new,

we believe it is the first time they are put together in order to build a complete translation. Starting from the standard translation of the simply typed λ-calculus into cartesian closed categories, we have shown how to enrich it with case analysis over sums, and generalized recursion. We have also shown how to adapt the hylomorphism derivation algorithm first presented in [12] to our λ-calculus. This algorithm enables the derivation of more tractable hylomorphisms, provided that the functions are defined with a special restricted syntax. When combined with pattern matching, this syntax corresponds to the one typically used to define most recursive functions in languages like Haskell.

We have also presented the Pointless Haskell library in which the resulting expressions can be directly executed. The implementation of polytypic abilities is similar to that used in the PolyP library. To enable a truly point-free style, we defined an implicit coercion mechanism that encodes a limited form of structural equivalence between types. The implementation required some extensions to the standard Haskell type system. If used without care, these extensions can make type-checking undecidable. By introducing type annotations similar to the ones used in the theoretical notation this problem was avoided. The main disadvantage of using this library is that, due to the heavy use of extensions, the error messages displayed by the compiler are of limited help for the programmer.

Although the resulting expressions are quite verbose, and sometimes quite intricate, they can be simplified by calculation. In fact, we have already implemented a prototype rewriting system that can automate some of these calculations, and that can automatically simplify the expressions resulting from the translation [6]. In the future we intend to integrate this system in the point-free derivation mechanism.

References

1. John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
2. Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.
3. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
4. R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–76, January 1977.
5. Roy Crole. *Categories for Types*. Cambridge University Press, 1993.
6. Alcino Cunha. *Point-free Program Calculation*. PhD thesis, Departamento de Informática, Universidade do Minho, 2005. To appear.

7. Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhäuser, 2nd edition, 1993.
8. Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction (MPC'04)*, volume 3125 of *LNCS*. Springer-Verlag, 2004.
9. Nicolaas de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
10. Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.
11. Jeremy Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer-Verlag, 2002.
12. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.
13. Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
14. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
15. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.
16. Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. To be submitted to The Journal of Functional Programming, March 2002.
17. Graham Klyne. Re: [haskell-cafe] point-free style (was: Things to avoid). Message sent to the Haskell-Cafe mailing list, February 2005.
18. Joachim Lambek. From lambda calculus to cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic*, pages 375–402. Academic Press, 1980.
19. Joachim Lambek and Philip Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Series in Advanced Mathematics*. Cambridge University Press, 1986.
20. Alfio Martini. Category theory and the simply-typed lambda calculus. Technical Report 7, Technische Universitaet Berlin, Informatik, 1996.
21. Colin McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides*. Oxford University Press, 1995.
22. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
23. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.
24. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, 1995.

25. Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Draft proceedings of the 15th International Workshop on the Implementation of Functional Languages (IFL'03)*, 2003.
26. Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997.
27. J.C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, July 1977.
28. Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–390, 2001.

A Examples of Translating Haskell to Point-free

We now present some examples of Haskell functions that can be translated into the point-free style using the prototype tool that implements the mechanism described in this paper. Both the Pointless library and a tool that translates a (very limited) subset of Haskell into the point-free style can be found in the first author's web page. The resulting Pointless Haskell code is presented in Figure 4. Notice that the `FunctorOf` instance for lists is predefined in the library.

```
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)

distr :: (a,Either b c) -> Either (a,b) (a,c)
distr (z, Left  y) = Left  (z, y)
distr (z, Right w) = Right (z, w)

data Nat = Zero | Succ Nat

plus :: (Nat,Nat) -> Nat
plus (Zero,   y) = y
plus (Succ x, y) = Succ (plus (x,y))

len :: [a] -> Nat
len []    = Zero
len (h:t) = Succ (len t)

fib :: Nat -> Nat
fib Zero            = Succ Zero
fib (Succ Zero)     = Succ Zero
fib (Succ (Succ x)) = plus (fib x, fib (Succ x))
```

```
swap :: (a, b) -> (b, a)
swap = (app . (((curry ((snd . snd) /\ (fst . snd))) . bang) /\ id))

distr :: (a, Either b c) -> Either (a, b) (a, c)
distr = (app . (((curry (app . ((eithr . ((curry (inl . ((fst . (snd . fst)) /\ snd))) /\ (curry (inr . ((fst . (snd . fst)) /\ snd))))) /\
            (snd . snd)))) . bang) /\ id))

data Nat = Zero | Succ Nat

instance FunctorOf (Const One :+: Id) Nat
    where inn' (Inl (Const _L)) = Zero
          inn' (Inr (Id n)) = Succ n
          out' Zero     = Inl (Const _L)
          out' (Succ n) = Inr (Id n)

plus :: (Nat, Nat) -> Nat
plus = hylo (_L :: Mu (Const a0 :+: Id))
       (app . (((curry (app . ((eithr . ((curry (snd . snd)) /\  (curry (inn . (inr . snd))))) /\ snd))) . bang) /\ id))
       (app . (((curry (app . ((eithr . ((curry (inl . (snd . fst))) /\ (curry (inr . (snd /\ (snd . (snd . fst)))))))) /\ (out . (fst . snd))))) . bang) /\ id))

len :: [a] -> Nat
len = hylo (_L :: Mu (Const One :+: Id))
      (app . (((curry (app . ((eithr . ((curry (inn . (inl . bang))) /\ (curry (inn . (inr . snd))))) /\ snd))) . bang) /\ id))
      (app . (((curry (app . ((eithr . ((curry (inl . bang)) /\ (curry (inr . (snd . snd))))) /\ (out . snd)))) . bang) /\ id))

fib :: Nat -> Nat
fib = hylo (_L :: Mu (Const One :+: (Const One :+: (Id :*: Id))))
      (app . (((curry (app . ((eithr . ((curry (inn . (inr . (inn . (inl . bang))))) /\ (curry (app . ((eithr . ((curry (inn . (inr . (inn . (inl . bang))))) /\
          (curry (app . (((((pnt plus) . fst) . fst) . fst) /\ ((fst . snd) /\ (snd . snd))))))) /\ snd))))) /\ snd))) . bang) /\ id))
      (app . (((curry (app . ((eithr . ((curry (inl . bang)) /\ (curry (inr . (app . ((eithr . ((curry (inl . bang)) /\ (curry (inr . (snd /\
          (inn . (inr . snd)))))))) /\ (out . snd)))))))) /\ (out . snd)))) . bang) /\ id))
```

Fig. 4. Pointless Haskell Examples