A Framework for Point-free Program Transformation*

Alcino Cunha, Jorge Sousa Pinto, and José Proença

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
{alcino,jsp,jproenca}@di.uminho.pt

Abstract. The subject of this paper is functional program transformation in the so-called *point-free* style. By this we mean first translating programs to a form consisting only of categorically-inspired combinators, algebraic data types defined as fixed points of functors, and impicit recursion through the use of type-parameterized recursion patterns. This form is appropriate for reasoning about programs equationally, but difficult to actually use in practice for programming. In this paper we present a collection of libraries and tools developed at Minho with the aim of supporting the automatic conversion of programs to *point-free* (embedded in Haskell), their manipulation, and their rule-driven simplification. The latter tool also supports the limited automatic application of *fusion* for program transformation.

1 Introduction

Functional Programming has always been known to be appropriate for activities involving manipulation of programs, such as program transformation. This is due to the strong theoretical basis that underlies the programming languages: the semantics of functional programs are easier to formalize.

As with any programming paradigm, different functional programmers use different styles of programming; it is however true that most advanced programmers resort to some concise form where functions are written as combinations of other functions, rather than programming by explicit manipulation of the arguments and explicit recursion. For instance a function that sums the squares of the elements in a list can be written in Haskell as

```
sum_squares = (foldr (+) 0) . (map sq)  where sq x = x*x
```

A radical style of programming is the so-called *point-free* style, which totally dispenses with variables. For instance the function `sq` above can be written as `sq = mult . (id /\ id)`, where the infix operator `/\` corresponds to the *split* combinator that applies two functions to an argument, producing a pair, and `mult` is the uncurried product.

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.

In the modern incarnation of these ideas, the combinators correspond to morphisms in a category (where the denotational semantics of the language are constructed) and the desired laws follow directly from universal properties of this category. What is more, this approach extends smoothly to the treatment of recursion in what is known as the *datatype-generic* approach to programming [8, 14]. This theory allows one to reason equationally about functions obtained by applying *recursion patterns* that encapsulate standard shapes of recursion, thus replacing the use of fixpoint induction. The generic aspect of this approach comes from the fact that all the constructions are parameterized by the recursive data types involved in the computations.

In the field of program transformation, well-known concepts like folding or fusion over lists were first introduced by Bird to derive accumulator-based implementations from inefficient specifications [2]. It is now widely accepted that the style described above is a good choice for reasoning about programs equationally and generically, as well for program transformation [4].

* This work was partially supported by FCT project POSI/CHS/44304/2002.

As a simple example of the kind of transformation we mean, in the function `map_squares` above the fold can be equationally fused with the map to give the following one-pass function, where `plus` is uncurried sum.

```
sum_squares' = foldr aux 0  where aux = curry (plus . (sq.fst /\ snd))
```

The drawback of using this radical 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. Pointwise vs. point-free is a lively discussion subject in Haskell forums; we would like to emphasize at this point that we do not advocate the use of point-free style directly for programming: the goal of the present paper is to present a set of libraries and tools that support point-free program transformation, but this includes the automatic translation of code to point-free form, so that programmers may apply point-free techniques to their code with variables. Specifically, we present here:

Pointless: a library for point-free programming, allowing programmers to type-check and execute point-free code with recursion patterns, parameterized by data types. 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.

DrHylo: a tool that allows programmers to automatically convert Haskell code to point-free form with recursion patterns. In particular, we employ the well-known equivalence between simply typed λ-calculi and cartesian closed categories suggested by Lambek [12]. This serves as the basis for the translation of a core functional language to categorical combinators, extended by the first author [3] to cover sum types. A second component here is the application of a standard algorithm that converts recursive functions to *hylomorphisms* of adequate regular data-types, thus removing explicit recursion.

SimpliFree: a tool for manipulating point-free code. Two different aspects of its use will be exemplified here:

- for the simplification of the very verbose terms produced by **DrHylo**;
- for program transformation by applying fold fusion.

These componentes are all freely available as part of the **UMinho Haskell Software** distribution.

Organization of the Paper. Section 2 introduces the languages used in the paper, and Section 3 reviews notions of point-free equational reasoning, with the help of an example. **Pointless**, **DrHylo**, and **Simplifree** are described in sections 4, 5 and 6. Finally Section 7 concludes the paper.

2 The Pointwise and Point-free Styles of Programming

In both styles, types are defined according to the following syntax.

$$A, B ::= 1 \mid A \to B \mid A \times B \mid A + B \mid \mu F$$
$$F, G ::= \mathsf{Id} \mid \underline{A} \mid F \otimes G \mid F \oplus G \mid F \circledcirc G$$

We assume a standard domain-theoretic semantics, where types are pointed complete partial orders, with least element $\bot$. 1 is the single element type, $A \to B$ is the type of continuous functions from A to B, $A \times B$ is the cartesian product, $A + B$ is the separated sum (with distinguished least element), and μF is a recursive (regular) type defined as the fixed point of a functor.

F and G range over functors, with Id denoting the identity functor, $\underline{A}$ the constant functor that always returns A, $\otimes$ and $\oplus$ the lifted product and sum bifunctors, and $\circledcirc$ composition of functors. For example, booleans can be defined as $\mathsf{Bool} = 1 + 1$, natural numbers as $\mathsf{Nat} = \mu(\underline{1} \oplus \mathsf{Id})$, and lists with elements of type A as $\mathsf{List}\ A = \mu(\underline{1} \oplus \underline{A} \otimes \mathsf{Id})$.

$$\frac{}{\Gamma \vdash \star : 1} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma[x \mapsto A] \vdash M : B}{\Gamma \vdash \lambda x.\ M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Gamma \vdash L : A + B \quad \Gamma \vdash M : A \to C \quad \Gamma \vdash N : B \to C}{\Gamma \vdash \mathsf{case}\ L\ M\ N : C}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\ M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\ M : B} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl}\ M : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{inr}\ M : A + B}$$

$$\frac{\Gamma \vdash M : F\ (\mu F)}{\Gamma \vdash \mathsf{in}_{\mu F}\ M : \mu F} \qquad \frac{\Gamma \vdash M : \mu F}{\Gamma \vdash \mathsf{out}_{\mu F}\ M : F\ (\mu F)} \qquad \frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathsf{fix}\ M : A}$$

Fig. 1. Typing rules

Pointwise Language. The syntax of terms with variables is generated by the following grammar.

$$\begin{aligned} L, M, N ::= {} & \star \mid x \mid M\ N \mid \lambda x.\ M \mid \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \\ & \mathsf{case}\ L\ M\ N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{in}_{\mu F}\ M \mid \mathsf{out}_{\mu F}\ M \mid \mathsf{fix}\ M \end{aligned}$$

Apart from variable, abstraction, and application, we find $\star$, which is the unique inhabitant of the terminal type (as such, it equals $\bot_1$); fst and snd are projections from a product type and inl and inr are injections into a sum type; $\langle \cdot, \cdot \rangle$ is a pairing construct, and case performs case-analysis on sums. Associated with each recursive type μF are two unique strict functions $\mathsf{in}_{\mu F}$ and $\mathsf{out}_{\mu F}$, that are each other's inverse. These provide the means to, respectively, construct and inspect values of the given type. Whenever clear from context, the subscripts will be omitted.

The typing rules are presented in Figure 1. We now show examples of terms in this language.

$$\begin{array}{llll} \mathsf{true} : \mathsf{Bool} & \mathsf{false} : \mathsf{Bool} & \mathsf{zero} : \mathsf{Nat} & \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{true} = \mathsf{inl}\ \star & \mathsf{false} = \mathsf{inr}\ \star & \mathsf{zero} = \mathsf{in}\ (\mathsf{inl}\ \star) & \mathsf{succ} = \lambda x.\ \mathsf{in}\ (\mathsf{inr}\ x) \\ \mathsf{nil} : \mathsf{List}\ A & \mathsf{cons} : A \to \mathsf{List}\ A \to \mathsf{List}\ A & & \mathsf{null} : \mathsf{List}\ A \to \mathsf{Bool} \\ \mathsf{nil} = \mathsf{in}\ (\mathsf{inl}\ \star) & \mathsf{cons} = \lambda h t.\ \mathsf{in}\ (\mathsf{inr}\ \langle h, t \rangle) & & \mathsf{null} = \lambda l.\mathsf{case}\ (\mathsf{out}\ l)\ (\lambda x.\mathsf{true})\ (\lambda x.\mathsf{false}) \\ \mathsf{swap} : A \times B \to B \times A & & \mathsf{distr} : A \times (B + C) \to (A \times B) + (A \times C) & \\ \mathsf{swap} = \lambda x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x \rangle & & \mathsf{distr} = \lambda x.\ \mathsf{case}\ (\mathsf{snd}\ x)\ (\lambda y.\ \mathsf{inl}\ \langle \mathsf{fst}\ x, y \rangle)\ (\lambda y.\ \mathsf{inr}\ \langle \mathsf{fst}\ x, y \rangle) & \end{array}$$

Recursive functions are defined explicitly using fix. For example, assuming that $\mathsf{mult} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$, the factorial and length functions can be defined as follows.

$$\begin{array}{l} \mathsf{fact} : \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{fact} = \mathsf{fix}\ (\lambda f.\ \lambda x.\ \mathsf{case}\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{succ}\ \mathsf{zero})\ (\lambda y.\ \mathsf{mult}\ \langle \mathsf{succ}\ y, f\ y \rangle)) \\ \mathsf{length} : \mathsf{List}\ A \to \mathsf{Nat} \\ \mathsf{length} = \mathsf{fix}\ (\lambda f.\ \lambda l.\ \mathsf{case}\ (\mathsf{out}\ l)\ (\lambda x.\ \mathsf{zero})\ (\lambda x.\ \mathsf{succ}\ (f\ (\mathsf{snd}\ y)))) \end{array}$$

Point-free Language. The set of combinators that is of interest to us come from universal constructions in *almost bicartesian closed categories*, that is, categories with products, non-empty sums, exponentials, and terminal object. See for instance [13] for a thorough treatment of the subject.

The point-free language contains the constants fst, snd, inl, inr, in, and out, with the obvious types, and also the set of combinators given below. To convey the meaning of each combinator, we give its definition in the pointwise language.

$$\begin{array}{ll} (\cdot \circ \cdot) : (B \to C) \to (A \to B) \to A \to C & \mathsf{id} : A \to A \\ (\cdot \circ \cdot) = \lambda f g x.\ f\ (g\ x) & \mathsf{id} = \lambda x.\ x \\ (\cdot \vartriangle \cdot) : (A \to B) \to (A \to C) \to A \to (B \times C) & \mathsf{bang} : A \to 1 \\ (\cdot \vartriangle \cdot) = \lambda f g x.\ \langle f\ x, g\ x \rangle & \mathsf{bang} = \lambda x.\ \star \\ (\cdot \triangledown \cdot) : (A \to C) \to (B \to C) \to (A + B) \to C & \mathsf{ap} : (A \to B) \times A \to B \\ (\cdot \triangledown \cdot) = \lambda f g x.\ \mathsf{case}\ x\ (\lambda y.\ f\ y)\ (\lambda y.\ g\ y) & \mathsf{ap} = \lambda x.\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x) \\ \overline{\cdot} : (A \times B \to C) \to A \to B \to C & \\ \overline{\cdot} = \lambda f x y.\ f\ \langle x, y \rangle & \end{array}$$

It is also convenient to have derived combinators corresponding to the operation of the product, sum, and exponentiation bifunctors on functions. These can be defined, respectively, as $f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd}$, $f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g$, and $f^{\bullet} = \overline{f \circ \mathsf{ap}}$.

The point-free language contains only values of functional type. As such, elements of a non-functional type A are denoted by functions of the isomorphic type $1 \to A$. The previous examples can be written in the point-free language as follows.

$$\begin{array}{llll}
\mathsf{true} : 1 \to \mathsf{Bool} & \mathsf{false} : 1 \to \mathsf{Bool} & \mathsf{zero} : 1 \to \mathsf{Nat} & \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{true} = \mathsf{inl} & \mathsf{false} = \mathsf{inr} & \mathsf{zero} = \mathsf{in} \circ \mathsf{inl} & \mathsf{succ} = \mathsf{in} \circ \mathsf{inr} \\
\mathsf{nil} : 1 \to \mathsf{List}\ A & \mathsf{cons} : A \to \mathsf{List}\ A \to \mathsf{List}\ A & & \mathsf{null} : \mathsf{List}\ A \to \mathsf{Bool} \\
\mathsf{nil} = \mathsf{in} \circ \mathsf{inl} & \mathsf{cons} = \overline{\mathsf{in} \circ \mathsf{inr}} & & \mathsf{null} = (\mathsf{true} \triangledown \mathsf{false} \circ \mathsf{bang}) \circ \mathsf{out} \\
\mathsf{swap} : A \times B \to B \times A & & \mathsf{distr} : A \times (B + C) \to (A \times B) + (A \times C) & \\
\mathsf{swap} = \mathsf{snd} \vartriangle \mathsf{fst} & & \mathsf{distr} = (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ \mathsf{swap} &
\end{array}$$

The language also contains a recursion operator: the *hylomorphism* recursion pattern. This was introduced with the first study of recursion patterns in a domain-theoretic setting [13], and was later proved to be powerful enough to allow for the definition of any fixpoint [14]. It is defined as follows.

$$\begin{array}{l}
\mathsf{hylo}_{\mu F} : (F\ B \to B) \to (A \to F\ A) \to A \to B \\
\mathsf{hylo}_{\mu F} = \lambda g.\ \lambda h.\ \mathsf{fix}(\lambda f.\ g \circ F f \circ h)
\end{array}$$

Function h is responsible for all computations prior to recursion, (i.e. 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. The recursion tree of a function defined as a hylomorphism is modeled by μF.

The factorial and length functions can then be defined in the point-free language as follows.

$$\begin{array}{l}
\mathsf{fact} : \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fact} = \mathsf{hylo}_{\mathsf{List\ Nat}}\ (\mathsf{zero} \triangledown \mathsf{mult})\ ((\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}) \\
\mathsf{length} : \mathsf{List}\ A \to \mathsf{Nat} \\
\mathsf{length} = \mathsf{hylo}_{\mathsf{Nat}}\ \mathsf{in}_{\mathsf{Nat}}\ ((\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}_{\mathsf{List}\ A})
\end{array}$$

Naturally, other derived operators can be defined using hylomorphism. The following correspond to the well-known *fold* and *unfold* recursion patterns:

$$\begin{array}{ll}
\mathsf{fold}_{\mu F} : (F\ A \to A) \to \mu F \to A & \mathsf{unfold}_{\mu F} : (A \to F\ A) \to A \to \mu F \\
\mathsf{fold}_{\mu F} = \lambda g.\ \mathsf{hylo}_{\mu F}\ g\ \mathsf{out}_{\mu F} & \mathsf{unfold}_{\mu F} = \lambda g.\ \mathsf{hylo}_{\mu F}\ \mathsf{in}_{\mu F}\ g
\end{array}$$

3 Point-free Program Transformation

The basic laws of the non-recursive calculus are given in appendix A. We will exemplify their use in the context of a non-trivial program transformation example taken from [4]. With respect to recursion, we resort to the following fold-fusion law:

$$f \circ (\!|g|\!)_F = (\!|h|\!)_F \quad \Leftarrow \quad f \text{ strict} \wedge f \circ g = h \circ F f \qquad \text{cata-FUSION}$$

where we use the compact notation $(\!|g|\!)_F$ for $\mathsf{fold}_{\mu F} g$ (strictness conditions are discussed in detail in [4]). Consider the function `isums::[Int]->[Int]` that computes the initial sums of a list.

```
isums []     = []
isums (x:xs) = map (x+) (0 : isums xs)
```

This function can be optimized by introducing an accumulating parameter to store at each point the sum of all previous elements in the list. We first define $\oplus : \mathsf{List\ Int} \times \mathsf{Int} \to \mathsf{List\ Int}$ as

$$\oplus\ (l, x) = \mathsf{map}_{\mathsf{List}}\ (\overline{\mathsf{plus}}\ x)\ l$$

Using this operator isums can be written as the fold $\mathsf{isums} = (\!|\underline{\mathsf{nil}} \triangledown \oplus \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{cons} \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id})|\!)$.

The optimized function isums_t can be calculated from the equation $\mathsf{isums}_t = \overline{\oplus} \circ \mathsf{isums}$ (or $\mathsf{isums}_t\ l\ y = \mathsf{map}_{\mathsf{List}}\ (\overline{\mathsf{plus}}\ y)\ (\mathsf{isums}\ l)$ pointwise), which plays the role of specification to the transformation. The reader can check that one obtains by fusion, with F the base functor of lists,

$$\mathsf{isums}_t = (\!|\underline{\underline{\mathsf{nil}}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{plus}} \times k)|\!)$$

if there exists a function k such that $\overline{\oplus} \circ \mathsf{cons} \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id} = k \circ \overline{\oplus}$ (the derived constant combinator $\underline{\cdot}$ is defined in appendix A). The following calculation allows to identify $k = \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \vartriangle \mathsf{id}$.

$$\begin{array}{ll} & \overline{\oplus} \circ \mathsf{cons} \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id} \\ = & \quad \{\, \mathsf{isums\text{-}Aux} \,\} \\ & \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id} \\ = & \quad \{\, \times\text{-Absor}, \mathsf{zero} \text{ is a left-identity of } \mathsf{plus} \,\} \\ & \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \vartriangle \overline{\oplus} \\ = & \quad \{\, \text{const-Fusion} \,\} \\ & \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \circ \overline{\oplus} \vartriangle \overline{\oplus} \\ = & \quad \{\, \times\text{-Fusion} \,\} \\ & \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \vartriangle \mathsf{id} \circ \overline{\oplus} \end{array}$$

We remark that this is far from being a simple calculation. The derived law isums-Aux, calculated in appendix C, introduces a new operator split that internalizes the $(\cdot \vartriangle \cdot)$ combinator in the point-free language. This is a useful technique for doing point-free calculations [4].

Substituting k and converting the resulting definition back to pointwise, one obtains at last the following linear time definition (note that isums runs in quadratic time).

```
isums_t :: [Int] -> Int -> [Int]
isums_t [] y     = []
isums_t (x:xs) y = (x+y) : isums_t xs (x+y)
```

4 *Pointless Haskell*: Programming Without Variables

Implementing the Basic Combinators. It is well known that the semantics of a real functional programming language like Haskell differs from the standard domain-theoretic characterization, 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 $(\bot, \bot) \neq \bot$, nor true categorical exponentials because $(\lambda x.\bot) \neq \bot$. Concerning products, any function defined using pattern matching, such as `\(_,_) -> 0`, can distinguish between $(\bot, \bot)$ and $\bot$. For exponentials, the examples are more subtle and typically involve using the standard `seq` function.

As discussed in [6], this fact complicates equational reasoning because the standard laws about products and functions no longer hold. In point-free however, as will be shown later, pairs can only be inspected using a standard set of combinators that cannot distinguish both elements, and thus Haskell pairs can safely be used to model products. If we prohibit the use of `seq`, the same applies to functions. This problem does not occur with sums because the separated sum also has a distinguished least element. Sums are modeled by the standard Haskell data type `Either`.

```
data Either a b = Left a | Right b
```

A second problem concerns the terminal object. Using the standard Haskell 98 it is not possible to define a type that implements 1, because any type declaration must have at least one constructor. The best approach would be to use the special predefined unit data type `()`, however, this still has two elements, namely `()` and `undefined`. 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 that allow for a data type without constructors to be declared.

```
data One
_L = undefined
```

The only element of this data type is `undefined` (with alias `_L`), thus it correctly implements `1`.

The definition of the point-free combinators in the **Pointless** library is trivial, and included in appendix B. Equipped with these definitions, non-recursive point-free expressions can be directly translated to Haskell. For example, the swap and distr functions 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) . app . ((curry inl \/ curry inr) >< id) . swap
```

Implementing Functors and Data Types. The implementation of recursive types in **Pointless** is based on the generic programming library PolyP [15]. This library also views data types as fixed points of functors, but instead of using an explicit fixpoint operator, a non-standard multi-parameter type class with a functional dependency [10] is used to relate a data type `d` with its base functor `f`.

```
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 each data type can have at most one. The main advantage of using `FunctorOf` is that predefined Haskell types can be viewed as fixed points of functors (the use of the primes will be explained later). A relevant subset of PolyP was reimplemented in **Pointless** according to our own design principles.

To avoid the explicit definition of the map functions, regular functors are described using a fixed set of combinators, according to the definition.

```
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 type for lists as the expected fixed point.

```
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)
```

Naturally, it is still possible to work with data types declared explicitly as fixed points of functors. The explicit fixpoint operator can be defined at the type level using `newtype`.

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

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
```

The following multi-parameter type class is used to convert values declared using the functor combinators into the corresponding standard Haskell types and vice-versa.

```
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 the identity and constant functors one has

```
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
```

For the case of products and sums, the types of the arguments should be computed prior to the resulting type. This evaluation order is guaranteed by using class constraints, as in

```
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`.

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

Note the annotations are compulsory since the same standard Haskell type can represent different functor combinations. This type-checking problem can be avoided by annotating the polytypic functions with the functor to which they should be specialized (similarly to the theoretical notation). Types cannot be passed as arguments to functions, and so this is achieved indirectly through the use of a "dummy" argument. 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 achieve an implicit coercion mechanism it suffices to insert the conversions in the functions that refer to functors, namely `inn'`, `out'`, and `fmap`. In fact, this was the reason why the primes were used in the declaration of the `FunctorOf` class. The following functions should be used instead.

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

Implementing Recursion. A polytypic hylomorphism operator can be defined using `pmap`.

```
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
```

Due to the use of implicit coercion it is now possible to program with hylomorphisms in a truly point-free style. For example, the definition of factorial from Section 2 can now be transcribed directly to Haskell. The same applies to derived recursion patterns. Notice the use of bottom as the dummy argument to indicate the type to which a polytypic function should be instantiated.

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

5 *DrHylo*: Deriving Point-free Hylomorphisms

DrHylo is a tool for deriving point-free definitions for a subset of Haskell. The resulting definitions can be executed with the **Pointless** library. It is based on the well-known equivalence between simply-typed λ-calculus and cartesian closed categories, first stated by Lambek [12]. One half of this correspondence is testified by a translation from pointwise terms to categorical combinators, later used by Curien to study a new implementation technique for functional languages – the *categorical abstract machine* [5]. This translation is also the starting point for our point-free derivation mechanism. We show here how it can be extended to handle sums and recursion.

The way variables are eliminated resembles the translation of the lambda calculus into the *de Bruijn notation*, where variables are represented by integers that measure the distance to their binding abstractions. Typing contexts are represented by left-nested pairs, as defined by the grammar $\Gamma ::= \star \mid \langle \Gamma, x : A \rangle$, with x a variable and A a type. Each variable will be 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, denoted by Φ, operates on typing judgments. A judgment is translated as $\Phi(\Gamma : B \vdash M : A) : B \to A$ according to the following rules (typing information is omitted).

$$\begin{array}{ll}
\Phi(\Gamma \vdash \star) & = \mathsf{bang} \\
\Phi(\Gamma \vdash x) & = \mathsf{path}(\Gamma, x) \\
\Phi(\Gamma \vdash MN) & = \mathsf{ap} \circ (\Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N)) \\
\Phi(\Gamma \vdash \lambda x.M) & = \overline{\Phi(\langle \Gamma, x\rangle \vdash M)} \\
\Phi(\Gamma \vdash \langle M, N\rangle) & = \Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N) \\
\Phi(\Gamma \vdash \mathsf{fst}\ M) & = \mathsf{fst} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{snd}\ M) & = \mathsf{snd} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{inl}\ M) & = \mathsf{inl} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{inr}\ M) & = \mathsf{inr} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{case}\ L\ M\ N) & = \mathsf{ap} \circ (\mathsf{either} \circ (\Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N)) \vartriangle \Phi(\Gamma \vdash L)) \\
\Phi(\Gamma \vdash \mathsf{in}\ M) & = \mathsf{in} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{out}\ M) & = \mathsf{out} \circ \Phi(\Gamma \vdash M)
\end{array}$$

The translation of a closed term $M : A \to B$ is a point of type $1 \to (A \to B)$, which can be converted into the expected function of type $A \to B$ as $\mathsf{ap} \circ (\Phi(\star \vdash M) \circ \mathsf{bang} \vartriangle \mathsf{id})$. For example, the swap function is translated as the following closed term of functional type.

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

We convert the result to a function of type $A \times B \to B \times A$ and simplify this as expected:

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

Concerning the translation of the case construct, first notice that $\mathsf{case}\ L\ M\ N$ is equivalent to $(M \triangledown N)\ L$. 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 $\mathsf{ap} \circ (\Phi(\Gamma \vdash M \triangledown N) \vartriangle \Phi(\Gamma \vdash L))$. The question remains of how to combine $\Phi(\Gamma \vdash M) : \Gamma \to (A \to C)$ and $\Phi(\Gamma \vdash N) : \Gamma \to (B \to C)$ into a function of type $\Gamma \to (A + B \to C)$. Our solution is based on the internalization of the uncurried version of the either combinator, that can be defined in point-free as follows.

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

As an example, consider the translation of the function coswap.

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

The following result is obtained, which (given some additional facts about either) can be easily simplified into the expected definition $\mathsf{inr} \triangledown \mathsf{inl}$.

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

It can be shown that the translation Φ is sound [5], i.e, all equivalences proved with an equational theory for the λ-calculus can also be proved using the equations that characterize the point-free combinators. Soundness of the translation of sums is proved in [3].

Translating Recursive Definitions. Two methods can be used for translating recursive definitions into hylomorphisms. The first is based on the direct encoding of fix by a hylomorphism, first proposed in [14]. The insight to this result is that $\mathsf{fix}\ f$ 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 $\mathsf{Stream}\ A = \mu(\underline{A} \otimes \mathsf{Id})$ with a single constructor $\mathsf{in} : A \times \mathsf{Stream}\ A \to \mathsf{Stream}\ A$. Given a function f, the hylomorphism builds the recursion tree $\mathsf{in}\ (f, \mathsf{in}\ (f, \mathsf{in}\ (f, \ldots)))$, and then just replaces in by ap. The operator and its straightforward translation are given as follows

$$
\begin{array}{l}
\mathsf{fix}\ :\ (A \to A) \to A \\
\mathsf{fix} = \mathsf{hylo}_{\mathsf{Stream}\ (A \to A)}\ \mathsf{ap}\ (\mathsf{id} \vartriangle \mathsf{id})
\end{array}
\qquad\qquad
\Phi(\Gamma \vdash \mathsf{fix}\ M) = \mathsf{fix} \circ \Phi(\Gamma \vdash M)
$$

Although complete, this translation yields definitions that are difficult to manipulate by calculation. 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. An algorithm that derives such hylomorphisms from explicitly recursive definitions has been proposed [9]. In the present context, the idea is to use this algorithm in a stage prior to the point-free translation: first, a pointwise hylomorphism is derived, and then the translation is applied to its parameter functions. **DrHylo** incorporates this algorithm, adapted to the present setting where data types are declared as fixed points, and pattern matching is restricted to sums. For the algorithm to work correctly, some restrictions must be imposed on the syntax used to define recursive functions, however these are broad enough to encompass most useful definitions.

Given a single-parameter recursive function defined as a fixpoint, three transformations are produced by the algorithm: 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 a third one for the function that is invoked prior to recursion ($\mathcal{C}$). The function $\mathsf{fix}\ (\lambda f.\ \lambda x.\ L) : A \to B$ is translated as the following hylomorphism.

$$\mathsf{hylo}_{\mu(\mathcal{F}(L))}\ (\lambda x.\ \mathcal{A}(L))\ (\lambda x.\ \mathcal{C}(L)) : A \to B$$

For example, the length function is converted into the following hylomorphism, which after being converted to the point-free style can easily be shown to be equal to the expected definition.

$$\begin{array}{l}\mathsf{length}\ :\ \mathsf{List}\ A \to \mathsf{Nat}\\ \mathsf{length} = \mathsf{hylo}_{\mu(\underline{1}\oplus \mathsf{Id})}\ (\lambda x.\ \mathsf{case}\ x\ (\lambda y.\mathsf{in}\ (\mathsf{inl}\ \star))\ (\lambda y.\ \mathsf{in}\ (\mathsf{inr}\ y)))\\ \qquad\qquad\qquad\qquad (\lambda x.\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{inl}\ \star)\ (\lambda y.\ \mathsf{inr}\ (\mathsf{snd}\ y)))\end{array}$$

Pattern-Matching. In order to apply this translation to realistic Haskell code, we still need to accommodate in our λ-calculus some form of pattern-matching, and data types defined by collections of constructors. Concerning data types, it is well-known how to implement an algorithm for defining `FunctorOf` instances for most user-defined data types [15]. This algorithm is incorporated in DrHylo, and since it replaces constructors by their equivalent fixpoint definitions it suffices to have pattern-matching over the generic constructor in, sums, pairs, and the constant $\star$.

We will now 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 pattern, and returns the respective right-hand side.

$$\begin{array}{rcl}P & ::= & \star \mid x \mid \langle P, P\rangle \mid \mathsf{in}\ P \mid \mathsf{inl}\ P \mid \mathsf{inr}\ P\\ M, N & ::= & \ldots \mid \mathsf{match}\ M\ \mathsf{with}\ \{P \to N;\ \ldots; P \to N\}\end{array}$$

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 [3]. We remark that since Haskell does not have true products, this rewrite relation can sometimes produce expressions whose semantic behaviour is different from the original. Consider the Haskell function `\(x,y) -> 0`. This function diverges when applied to `_L`, but returns zero if applied to `(_L,_L)`. This function can be directly encoded using match and translated into the core λ-calculus using the following rewrite sequence.

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

Since it no longer has pattern-matching, the resulting function is different from the original since it never diverges. Apart from this problem, with this pattern-matching construct it is now possible to translate into point-free many typical Haskell functions, such as the ones defined in appendix C.

Using this construct, it is now possible to define functions using a syntax more similar to that of Haskell. For example, distr and the length function can be defined as follows (list constructors are replaced by their point-wise definition given in Section 2).

$$\begin{array}{l}\mathsf{distr}\ :\ A \times (B + C) \to (A \times B) + (A \times C)\\ \mathsf{distr} = \lambda x.\mathsf{match}\ x\ \mathsf{with}\ \{\langle y, \mathsf{inl}\ z\rangle \to \mathsf{inl}\ \langle y, z\rangle;\ \langle y, \mathsf{inr}\ z\rangle \to \mathsf{inr}\ \langle y, z\rangle\}\\ \mathsf{length}\ :\ \mathsf{List}\ A \to \mathsf{Nat}\\ \mathsf{length} = \mathsf{fix}(\lambda f.\lambda l.\mathsf{match}\ l\ \{\mathsf{in}\ (\mathsf{inl}\ \star) \to \mathsf{in}\ (\mathsf{inl}\ \star);\ \mathsf{in}\ (\mathsf{inr}\ \langle h, t\rangle)\ \to \mathsf{in}\ (\mathsf{inr}\ (f\ t))\})\end{array}$$

6 *SimpliFree*: Implementing Program Transformations

In this section we describe the approach we have followed to automatically simplify point-free expressions with the tool SimpliFree, and then show examples of its use in two critical situations: first to simplify the output expressions generated by DrHylo (which are usually very verbose), and then to perform program transformations by applying fold fusion, as exemplified in Section 3. For full details on the tool and its implementation the reader is directed to [16].

Basic Principles. The tool is based on the notion of *active source*, in the same way as MAG [7]. The idea is to write a *Haskell* file containing the expressions to be simplified, along with rules to be used in their simplification. These rules are used together with those contained in a pre-existing repository, and their application is oriented by a *strategy*. Simplifications are implemented by *generic traversals* using Strafunski [11].

From this file, the tool produces a new Haskell file containing the code that executes the simplifications; it can either be compiled or interpreted to allow for the simplification to be examined step by step. The tool supplies default strategies that work in many cases.

An important characteristic of the tool is that it uses the host language's own pattern-matching mechanism; unlike MAG however, it does not possess a fixed strategy; instead, the user is free to use a number of strategy combinators to produce new strategies adequate for particular classes of simplifications, which we see as a great advantage.

Using the Tool. The *point-free* terms are parsed from a standard *Haskell* file, and the information about the traversals is read from special annotated blocks, where it is possible to state:

- Options (import a known set of strategies)
- Rules (name, conversion and possible conditions)
- Strategies (combination of rules and/or strategies)
- Optimisations (association of *point-free* terms to strategies)

When importing a known strategy (in a special annotated block or by passing the corresponding argument), that strategy will be applied by default to all *point-free* terms, unless stated otherwise.

The output of the tool is a new *Haskell* code file where:

- The strategies are now built with functions defined in the SimpliFreelibrary, which apply the traversal schemes;
- The rules are functions with type `Term -> Maybe Term`, that try to apply a transformation to a term or to the *prefix* (in a composition) of a term;
- The application of a strategy to a term returns a computation, with the intermediate calculations and list of applied rules;
- The `main` function prints the original file, where the simplified terms replace the old ones.

The Library. A *computation* is defined as a final *point-free* term together with a list of intermediate results (pairs of $term \times rule$). Strafunski has *type preserving* and *type unifying* strategy combinators. To calculate a computation it is necessary to combine both strategies, in order to change a term (type preservation) and to collect all changed terms and rules applied (type unification).

For example, the *and* combinator is defined in the SimpliFreelibrary as:

```
{-| Tries the first strategy and then tries a second one -}
andPF :: MonadPlus m => TU Computation m -> TU Computation m -> TU Computation m
s1 `andPF` s2 = s1 `passTU` \(lst1,t1) ->
                  (constTP t1) `seqTU` s2 `passTU` \(lst2,t2) ->
                     (constTU (lst1++lst2,t2))
  where constTP t = adhocTP idTP (\_->return t)
```

A special traversal to preserve the composition invariant (composition associates to the right) was defined and is applied at the begining and after the aplication of any rule.

The strategy combinators defined in the library are: *rule*, *many*, *or*, *and*, *oneOrMore*, *optional* and *fail*. Other auxiliary functions were alse defined inside the library.

Construction of Rules. This is a central problem for the tool. A rule is a monadic function that applies a transformation to a term or to a *prefix* of a term, by pattern-matching a point-free term. Several important issues were succesfully overcome:

- Composition has different patterns in the end and anywhere else;
- When a variable pattern is found at the left of a composition there might be the need to try different associations;
- When a condition fails, the next match has to be tested.
- Rules have to be produced automatically, based on the left and right-hand sides, conditions, and rule names.

The usability was improved with the automatic inference of conditions in the presence of repeated variables, with the creation of special rules that originate new rules (*e.g.*, describing the associativity property, or defining the fold and unfold of rules), and with the possibility of calling strategies in the right-hand side of a rule. Consider a file containing two rules as follows

```
{- Rules:
prodCancel : fst . (f /\ g) -> f
prodFusInv : (f.h) /\ (g.h) -> (f/\g) . h
-}
```

Note that in `prodCancel` the whole term and the prefix have to be tested if the pattern match, and in `prodFusInv` there are repeated variables in the l.h.s (converted to conditions), and the variables `f` and `g` occur on the left of a composition. SimpliFreeoutputs the following:

```
prodCancel (FST :.: (f :/\: g)) = return (f)
prodCancel (FST :.: ((f :/\: g) :.: x)) = return (f :.: x)
prodCancel _ = fail "rule prodCancel1 not applied"

prodFusInv (f :/\: g)
  | and [success (xx_g g), success (xx_f f), verifyCond xxi] =
    return
      ((\ g x_h f h -> getEnd xxe ((f :/\: g) :.: h)) (getTerm 0 xxe)
         (getTerm 1 xxe)
         (getTerm 2 xxe)
         (getTerm 3 xxe))
  where xxp  = joinVars [xx_g g, xx_f f]
        xxi  = testCond (\ [g, x_h, f, h] -> h == x_h) xxp
        xxe  = getIndex xxi xxp
        xx_g (g :.: x_h)
          = addTerm g (addTerm x_h (emptyVar)) ++ addComp g (xx_g (x_h))
        xx_g (xx_g0 :.: xx_g1)
          | success (xx_g xx_g1) = addComp xx_g0 (xx_g xx_g1)
        xx_g _ = noVar
        xx_f (f :.: h)
          = addTerm f (addTerm h (emptyVar)) ++ addComp f (xx_f (h))
        xx_f (xx_f0 :.: xx_f1)
          | success (xx_f xx_f1) = addComp xx_f0 (xx_f xx_f1)
        xx_f _ = noVar
prodFusInv _ = fail "rule prodFusInv not applied"
```

Strategies. The main strategy in the rules repository is `base_strat`. Other strategies in the repository are extensions to `base_strat`, obtained by redefining some auxiliary strategies set initially to `fail`. The `base_strat` strategy consists in applying a set of rules several times, unfolding the macros, and repeating the process while possible. In the end the known macros are again folded.

```
{- Strategies:
base_strat : base_unMacro and (many base_fMacro)
```

```
base_unMacro : base_simplify and
                  (opt ((oneOrMore base_unfMacro) and base_unMacro))
base_simplify : many (base_pre or base_rules or base_pos)

base_pre      : fail
base_pos      : fail
macros_fold   : fail
macros_unfold : fail

base_rules :  natId1 or natId2 or prodCancel1 ...
base_unfMacro : exp_unfold or pnt_unfold or swap_unfold ..
base_fMacro   : exp_fold or  pnt_fold or swap_fold ...
-}

{- Rules:
natId1 : id . f -> f
natId2 : f . id -> f
prodCancel1 : fst . (f /\ g) -> f
...

exp_unfold  : 'exp' [f] -> curry (f . app)
pnt_unfold  : 'pnt' [f] -> curry (f.snd)
swap_unfold : 'swap'    -> snd /\ fst
...
exp_fold  : curry (f . app) -> 'exp' [f]
pnt_fold  : curry (f.snd)   -> 'pnt' [f]
swap_fold : snd /\ fst      -> 'swap'
...
-}
```

We end this section by showing how the **SimpliFree** tool effectively simplifies the point-free definitions derived in Section 5 for swap and coswap. The simplifications are performed using the strategy adv_strat, which is an elaboration of the basic strategy. Appendix C contains a further example of using SimpliFree, to perform a program transformation by fusion.

```
*Main> swap_adv_strat
app.((curry ((snd.snd) /\ (fst.snd)).bang) /\ id)
  = { expCancAdv3 }
((snd.snd) /\ (fst.snd)).(bang /\ id)
  = { prodFus }
(snd.snd.(bang /\ id)) /\ (fst.snd.(bang /\ id))
  = { prodCancel2 }
(snd.id) /\ (fst.snd.(bang /\ id))
  = { natId2 }
snd /\ (fst.snd.(bang /\ id))
  = { prodCancel2 }
snd /\ (fst.id)
  = { natId2 }
snd /\ fst
  = { swap_fold }
'swap'
```

```
*Main> coswap_adv_strat
app.((curry (app.(('eithr'.(curry (inr.snd) /\ curry (inl.snd))) /\ snd)).bang) /\ id)
  = { eitherConst }
app.((curry (app.(curry ((inr \/ inl).snd) /\ snd)).bang) /\ id)
  = { expCancAdv3 }
app.(curry ((inr \/ inl).snd) /\ snd).(bang /\ id)
  = { prodFus }
```

```
app.((curry ((inr \/ inl).snd).(bang /\ id)) /\ (snd.(bang /\ id)))
  = { prodCancel2 }
app.((curry ((inr \/ inl).snd).(bang /\ id)) /\ id)
  = { expCancAdv3 }
(inr \/ inl).snd.(bang /\ id /\ id)
  = { prodCancel2 }
(inr \/ inl).id
  = { natId2 }
inr \/ inl
  = { coswap_fold }
'coswap'
```

7 Conclusions and Future Work

We have focused on the most important aspects of each component of the framework; more documentation can be found at `http://wiki.di.uminho.pt/wiki/bin/view/PURe/PUReSoftware` (the UMinho Haskell Software pages).

While **Pointless** has reached a stable stage of development, there are still many points for improvement in the other components. In **DrHylo**, the translation of recursive functions must be improved with the automatic translation to other standard recursion patterns such as folds, unfolds, and paramorphisms, rather than always resorting to the all-encompassing hylomorphisms.

In **SimpliFree**, we plan to incorporate other laws for recursive functions, such as unfold-fusion. An immediate goal is to make the fusion mechanism more powerful, to cover at least all the transformations that can be done with MAG.

A significant improvement will be the introduction of truly generic laws: in the current version different folds are used for different data types, which require different laws. There is a mismatch with the theoretical formulation, where recursion patterns and laws are truly generic.

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. Alcino Cunha. *Point-free Program Calculation*. PhD thesis, Departamento de Informática, Universidade do Minho, 2005.
4. Alcino Cunha and Jorge Sousa Pinto. Point-free program transformation. *Fundamenta Informaticae*, 66(4):315–352, 2005. Special Issue on Program Transformation.
5. Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhuser, 2nd edition, 1993.
6. 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.
7. Oege de Moor and Ganesh Sittampalam. Generic program transformation. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Proceedings of the 3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 116–149. Springer-Verlag, 1999.
8. 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.
9. 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.
10. Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

11. Ralf Laemmel and Joost Visser. Typed combinators for generic traversal. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 137–154, London, UK, 2002. Springer-Verlag.
12. 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.
13. 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.
14. 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.
15. 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.
16. José Proença. Point-free simplification. Technical Report DI-PURe-05.06.01, Universidade do Minho, 2005.

A Equational Laws

$$\langle \pi_1, \pi_2 \rangle = \mathsf{id} \qquad \times\text{-Reflex}$$
$$\pi_1 \circ \langle f, g \rangle = f \;\wedge\; \pi_2 \circ \langle f, g \rangle = g \qquad \times\text{-Cancel}$$
$$\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle \qquad \times\text{-Fusion}$$
$$(f \times g) \circ \langle h, i \rangle = \langle f \circ h, g \circ i \rangle \qquad \times\text{-Absor}$$
$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-Functor}$$
$$\langle f, g \rangle = \langle h, i \rangle \quad \Leftrightarrow \quad f = h \;\wedge\; g = i \qquad \times\text{-Equal}$$
$$f \vartriangle g \text{ strict} \quad \Leftrightarrow \quad f \text{ strict} \;\wedge\; g \text{ strict} \qquad \times\text{-Strict}$$

$$[i_1, i_2] = \mathsf{id} \qquad +\text{-Reflex}$$
$$[f, g] \circ i_1 = f \;\wedge\; [f, g] \circ i_2 = g \qquad +\text{-Cancel}$$
$$f \circ [g, h] = [f \circ g, f \circ h] \quad \Leftarrow \quad f \text{ strict} \qquad +\text{-Fusion}$$
$$[f, g] \circ (h + i) = [f \circ h, g \circ i] \qquad +\text{-Absor}$$
$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad +\text{-Functor}$$
$$[f, g] = [h, i] \quad \Leftrightarrow \quad f = h \;\wedge\; g = i \qquad +\text{-Equal}$$
$$\forall f, g \cdot f \triangledown g \text{ strict} \qquad +\text{-Strict}$$

$$\overline{\mathsf{ap}} = \mathsf{id} \qquad \wedge\text{-Reflex}$$
$$f = \mathsf{ap} \circ (\overline{f} \times \mathsf{id}) \qquad \wedge\text{-Cancel}$$
$$\overline{f \circ (g \times \mathsf{id})} = \overline{f} \circ g \qquad \wedge\text{-Fusion}$$
$$f^A \circ \overline{g} = \overline{f \circ g} \qquad \wedge\text{-Absor}$$
$$(f \circ g)^A = f^A \circ g^A \qquad \wedge\text{-Functor}$$
$$\overline{f} = \overline{g} \quad \Leftrightarrow \quad f = g \qquad \wedge\text{-Equal}$$
$$\overline{f} \text{ strict} \quad \Leftrightarrow \quad f \text{ left-strict} \qquad \wedge\text{-Strict}$$

$$\underline{f} = \overline{f \circ \pi_2} \qquad \text{const-Def}$$
$$\underline{f} \circ g = \underline{f} \qquad \text{const-Fusion}$$

B Implementing Point-free Combinators

Some combinators, such as `fst`, `snd` and `curry` are already predefined in the Haskell standard prelude. The remaining are implemented as follows.

```
_L :: a
_L = undefined

bang :: a -> One
bang _ = _L

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

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

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

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

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 advantage of a similar graphical notation.

C isums Example – Auxiliary Calculations

The auxiliary calculation for the example in section 3 introduces the split operator

$$\begin{array}{l} \mathsf{split} \;:\; (B^A \times C^A) \to (B \times C)^A \\ \mathsf{split} = \overline{(\mathsf{ap} \times \mathsf{ap}) \circ \pi_1 \times \mathsf{id} \;\vartriangle\; \pi_2 \times \mathsf{id}} \end{array} \qquad \text{split-DEF}$$

The required auxiliary law is

$$\overline{\oplus} \circ \mathsf{cons} = \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \qquad \text{isums-AUX}$$

which can be proved by the following calculation.

$$
\begin{array}{ll}
\lceil & \overline{\oplus} \circ \mathsf{cons} \\
= & \quad \{ \wedge\text{-Fusion} \} \\
& \overline{\oplus \circ (\mathsf{cons} \times \mathsf{id})} \\
= & \quad \{ \oplus \circ (\mathsf{cons} \times \mathsf{id}) = \mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ \pi_1 \times \mathsf{id} \vartriangle \pi_2 \times \mathsf{id} \} \\
& \overline{\mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ \pi_1 \times \mathsf{id} \vartriangle \pi_2 \times \mathsf{id}} \\
= & \quad \{ \wedge\text{-Cancel}, \times\text{-Functor}, \times\text{-Absor} \} \\
& \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ \overline{\mathsf{plus}} \circ \pi_1 \times \mathsf{id} \vartriangle \overline{\oplus} \circ \pi_2 \times \mathsf{id}} \\
= & \quad \{ \times\text{-Cancel}, \text{def. of } \times \} \\
& \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ \pi_1 \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id} \vartriangle \pi_2 \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id}} \\
= & \quad \{ \times\text{-Functor}\times\text{-Fusion} \} \\
& \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ \pi_1 \times \mathsf{id} \vartriangle \pi_2 \times \mathsf{id} \circ ((\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id})} \\
= & \quad \{ \wedge\text{-Fusion}, \wedge\text{-Absor} \} \\
& \mathsf{cons}^\bullet \circ \overline{(\mathsf{ap} \times \mathsf{ap}) \circ \pi_1 \times \mathsf{id} \vartriangle \pi_2 \times \mathsf{id}} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \\
= & \quad \{ \mathsf{split}\text{-Def} \} \\
\lfloor & \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus})
\end{array}
$$

This gives an example of a slightly longer simplification in **SimpliFree**:

```
('exp' ['cons']).'split'.(curry 'plus' >< curry 'mapPlus')
 = { exp_unfold }
curry ('cons'.app).'split'.(curry 'plus' >< curry 'mapPlus')
 = { split_unfold }
curry ('cons'.app).curry ((app.(fst >< id)) /\ (app.(snd >< id))).
(curry 'plus' >< curry 'mapPlus')
 = { expFus }
curry ('cons'.app.((curry ((app.(fst >< id)) /\ (app.(snd >< id))).
(curry 'plus' >< curry 'mapPlus')) >< id))
 = { expCancAdv1 }
curry ('cons'.((app.(fst >< id)) /\ (app.(snd >< id))).
((curry 'plus' >< curry 'mapPlus') >< id))
 = { prodFus }
curry ('cons'.((app.(fst >< id).
((curry 'plus' >< curry 'mapPlus') >< id)) /\ (app.(snd >< id).
((curry 'plus' >< curry 'mapPlus') >< id))))
 = { prodFun }
curry ('cons'.((app.((fst.(curry 'plus' >< curry 'mapPlus')) >< (id.id)))
 /\ (app.(snd >< id).
((curry 'plus' >< curry 'mapPlus') >< id))))
 = { natId1 }
curry ('cons'.((app.((fst.(curry 'plus' >< curry 'mapPlus')) >< id))
 /\ (app.(snd >< id).
((curry 'plus' >< curry 'mapPlus') >< id))))
 = { prodCancel1' }
curry ('cons'.((app.((curry 'plus'.fst) >< id)) /\ (app.(snd >< id).
((curry 'plus' >< curry 'mapPlus') >< id))))
 = { prodFun }
curry ('cons'.((app.((curry 'plus'.fst) >< id)) /\
(app.((snd.(curry 'plus' >< curry 'mapPlus')) >< (id.id)))))
 = { natId1 }
curry ('cons'.((app.((curry 'plus'.fst) >< id)) /\
(app.((snd.(curry 'plus' >< curry 'mapPlus')) >< id))))
 = { prodCancel2' }
curry ('cons'.((app.((curry 'plus'.fst) >< id)) /\
(app.((curry 'mapPlus'.snd) >< id))))
 = { expCancAdv1 }
curry ('cons'.(('plus'.(fst >< id)) /\ (app.((curry 'mapPlus'.snd) >< id))))
 = { expCancAdv1 }
```

```
curry ('cons'.(('plus'.(fst >< id)) /\ ('mapPlus'.(snd >< id))))
 = { xx }
curry ('mapPlus'.('cons' >< id))
 = { expFus_fold }
curry 'mapPlus'.'cons'
```

The full calculation of isums_t by fusion can be done automatically as follows. Note the inclusion of the above auxiliary result as a rule.

```
isums_t = curry mapPlus .
         cataList ((pnt nil) \/ (mapPlus . swap . (id >< (cons.((pnt zero) /\ id)))))
{- Rules:
cataList_rules : plusDualAssoc or isums_hip or isums_help
plusDualAssoc : (curry 'mapPlus') . 'mapPlus'
            -> 'comp' . ((curry 'mapPlus') >< (curry 'plus'))
isums_hip : (curry 'mapPlus') . 'cons'
        -> ('exp' ['cons']) . ('split' . ((curry 'plus') >< (curry 'mapPlus')))
isums_help : (f . ('pnt' [g])) /\ (curry 'mapPlus')
         -> ((f . ('pnt' [g])) /\ id ) . (curry 'mapPlus')
-}
```

Due to its size we show just the initial and final lines of the resulting computation.

```
curry 'mapPlus'.('cataList' [('pnt' ['nil']) \/
('mapPlus'.'swap'.(id >< ('cons'.(('pnt' ['zero']) /\ id))))])
 = { ... }
'cataList' [(curry 'mapPlus'.curry ('nil'.snd)) \/
('comp'.(curry ('cons'.(('plus'.(curry ('zero'.snd) >< id)) /\
(app.(snd >< id)))) /\ (curry 'plus'.fst)))]
```