Fundamenta Informaticae XX (2005) 1–38
IOS Press

Point-free Program Transformation

Alcino Cunha
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga, Portugal
alcino@di.uminho.pt

Jorge Sousa Pinto
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga, Portugal
jsp@di.uminho.pt

Keywords: Functional Programming, Program Transformation, Program Calculation, Point-free Programming, Accumulation Strategy

Abstract. Functional programs are particularly well suited to formal manipulation by equational reasoning. In particular, it is straightforward to use calculational methods for program transformation. Well-known transformation techniques, like tupling or the introduction of accumulating parameters, can be implemented using calculation through the use of the *fusion* (or *promotion*) strategy. In this paper we revisit this transformation method, but, unlike most of the previous work on this subject, we adhere to a pure point-free calculus that emphasizes the advantages of equational reasoning. We focus on the accumulation strategy initially proposed by Bird, where the transformed programs are seen as higher-order folds calculated systematically from a specification. The machinery of the calculus is expanded with higher-order point-free operators that simplify the calculations. A substantial number of examples (both classic and new) are fully developed, and we introduce several shortcut optimization rules that capture typical transformation patterns.

1. Introduction

Functional programming is particularly appropriate for equational reasoning. This has been known for a long time, at least since Burstall and Darlington [8] introduced the fold/unfold technique, and Backus [2] proposed his calculational methodology.

In fold/unfold program transformation one applies a number of semantically sound rules to an initial program, with the aim of arriving at a better, equivalent transformed program. “Better” here may have different interpretations: time and space complexity improvements are obvious criteria, but removal of recursion is also a common goal (allowing us to convert programs into purely iterative forms). This is an activity that involves steps that are not easily automated, and as such typically requires human

intervention. For instance, the transformation rules include the possibility of defining new auxiliary functions, and then folding them in the body of the program being manipulated (see section 3.1 for an example). In fact, fold/unfold can be best described as a framework for program transformation, based on which a number of transformation strategies and techniques have been studied. *Tupling* [8] and *deforestation* [34] are examples of such techniques.

In this paper we use a different framework for reasoning about functional programs – *by calculation*. Some classic strategies for program transformation have been introduced using this framework, such as Bird's *accumulation* strategy [4]. Essentially a program calculus consists of a collection of equational laws allowing us to prove semantic equivalence between programs, or else to derive programs from other programs, or from their *specifications*. Quoting Backus [2]:

> *Associated with the functional style of programming is an algebra of programs [...] This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high-school algebra.*

One advantage of the calculational approach is that one can use the programming language itself to express properties and reason about the programs, rather than having a different formalism. Although not so general as the fold/unfold technique, this approach is also easier to mechanize because it only implies a local program analysis and the application of simple rewrite rules (typically with simple or no side-conditions to verify), and since it does not require any global analysis it can also be implemented in a modular way [33].

The program calculus used in this paper is built upon two basic ingredients. The first is a set of *recursion patterns*, higher-order operators that encapsulate typical patterns of recursion, such as the well-known *fold* operator on lists. These operators enjoy a nice set of equational laws, and their importance to functional programming has been compared to the abandoning of arbitrary *gotos* in favour of structured control primitives in the imperative setting. The second is a *point-free* style of programming in which programs are expressed as combinations of simpler functions, without ever mentioning their arguments. The calculus uses a reduced set of combinators, derived from standard categorical constructions, again characterized by a rich set of equational laws.

While recursion patterns are already widely used by functional programmers, the same cannot be said about point-free programming. Although there are some obvious advantages in using this style – the absence of variables and lambda abstractions simplifies the presentation and implementation of reduction rules – most authors still resort to pointwise, both for programming and for calculation, arguing that the intuitive meaning of point-free programs may be easily lost (it has even been jokingly called the *pointless* style). Quoting Gibbons on the advantages of calculating in the point-free style [13]:

> *This is the point of pointless calculations: when you travel light – discarding variables that do not contribute to the calculation – you can sometimes step lightly across the surface of the quagmire.*

In this paper we use a purely point-free style only for calculations. Our initial definitions will be expressed in pointwise Haskell [21], which must be converted into the language of the calculus (point-free style with recursion patterns) before being transformed by calculation. At the end of the transformation, the resulting functions will be converted back into the original domain. A useful comparison here is that of mathematical transforms such as the Fourier transform or the Laplace transform, which allow us to express functions in different domains in which certain manipulations are easier to perform.

The main goal of this paper is to revisit some classic work in the area of program transformation using pure point-free calculations. The paper is focused on the transformation of programs by introducing new accumulating parameters, according to the strategy initially proposed by Bird [4], where the transformed programs are seen as higher-order folds calculated systematically from specifications. We present a systematic approach to this program transformation technique, together with a substantial number of examples. This systematization leads to a set of generic transformation schemes, that could be used as shortcut optimization rules in an automatic program transformation system.

A second goal is the improvement of the machinery that is used to perform point-free calculations in a higher-order setting. Quoting Gibbons again [12]:

> *We are interested in extending what can be calculated precisely because we are not interested in the calculations themselves [...]*

In other words, we aim at extending the calculus with new useful operators that help reducing the burden of proofs just to the creative parts.

Organization of the paper. Section 2 contains all the necessary background material on recursion patterns and the point-free program calculus; acquaintance with basic notions of program semantics is assumed; rudiments of category theory will be helpful but not essential. This section briefly presents the historic development of the field. Section 3 introduces the theme of the paper by presenting a classic example fully worked in three styles: (i) using unfold-fold transformation; (ii) using point-level calculation; (iii) using the point-free calculus. This example also introduces the use of accumulations and tail-recursion, to be used throughout the paper. In section 4 it is shown how accumulations can be systematically calculated for programs over lists, binary trees, and rose trees in a point-free setting. In section 5 we extend this approach to functions with two accumulating parameters. Although specifications are typically folds (the simplest form of structural recursion), this approach can be generalized to other forms of recursion, as shown in section 6. Section 7 reviews some related approaches, and section 8 concludes the paper. Appendix A contains an extended set of laws concerning the point-free combinators introduced in section 2.

2. Point-free Programming with Recursion Patterns

In his 1977 ACM Turing Award lecture, John Backus proposed a new functional style of programming whose main features were the absence of variables and the use of *functional forms* to combine existing functions into new functions [2]. The main idea was to develop a calculus of programs that could be used for program transformation. The choice of the functional forms was based not only on their programming power, but also on the power of the associated algebraic laws. Most of the now standard point-free combinators (presented later in this section) were already introduced by Backus.

This approach was later endorsed by Bird and Meertens, who popularized a style of programming (the so-called "Bird-Meertens formalism") where final programs were derived from their specifications (typically, an inefficient combination of easy to understand functions) through a set of equational laws [4, 5, 24]. The now well-known notions of folding and *fusion* (or *promotion*) over lists were presented in this work, enabling for the first time the effective use of the calculational approach in program transformation. The main difference with respect to the initial approach by Backus was the occasional use of

the pointwise style. As Backhouse pointed out [1], the importance of the Bird-Meertens formalism lies not on the foundational concepts *per se* (at the time already known), but on their application to develop a concise calculational method for program transformation.

Malcolm [23] later showed that the concepts introduced by Bird and Meertens arise naturally for any data type when viewed in a categorical setting. The categorical approach to data types and functional programming in general had been previously clarified by Hagino [17]; category theory turned out to be a natural setting for defining the basic building blocks of data types (including sophisticated concepts such as mutually recursive data types, types defined using other parameterized data types, and infinite data types). As will be shown below, the definitions of most of the combinators used in the point-free style of programming are immediate from standard categorical constructions.

The generalization proposed by Malcolm was done in the context of total functions and totally defined elements, but later Meijer, Fokkinga, and Paterson [11, 25] extended it to the domain of partial functions and elements, thus enabling the power of full recursion, and providing a more appropriate semantic domain to modern lazy programming languages, like Haskell.

There are many introductory texts to this style of programming, covering the subject of this section [25, 7, 15].

2.1. Basic Combinators and Functors

The semantic characterization of modern lazy programming languages (like Haskell) is usually based on pointed complete partial orders (sets equipped with a partial order, a least element denoted $\bot$, and closed under limits of ascending chains), and continuous functions (monotonic functions that preserve limits). We remark however that most of the research done in program transformation in a calculational setting uses a less natural semantics based on sets and total functions that makes difficult the treatment of arbitrary recursion and partiality.

The study of algebraic programming in this setting was pioneered by Meijer, Fokkinga and Paterson [25], and the presentation of the material in this section (based on a categorical account of the denotational semantics, with functions modeled by arrows in the $\mathbf{Cpo}$ category, and types by objects in that category) is strongly influenced by their work.

Functors. A *functor* F is a mapping between categories (it maps objects to objects and arrows to arrows) such that

$$F\ f : F\ A \to F\ B \quad \Leftarrow \quad f : A \to B \qquad \text{functor-TYPE}$$

$$F(f \circ g) = F\ f \circ F\ g \qquad \text{functor-COMPOSE}$$

$$F\ \mathsf{id}_A = \mathsf{id}_{FA} \qquad \text{functor-ID}$$

For our purposes (and in general in the context of programming language semantics), *endofunctors* in $\mathbf{Cpo}$ will be used, mapping types to types, and functions to functions.

The simplest functor is the identity functor Id, whose action on types is defined as $\mathsf{Id}\ A = A$, and on functions as $\mathsf{Id}\ f = f$. Also important is the constant functor: given a type A, the functor $\underline{A}$ is defined on types as $\underline{A}\ B = A$, and on functions as $\underline{A}\ f = \mathsf{id}_A$.

A *bifunctor* is a mapping from a pair of categories to a category, with the expected requirements concerning composition and identity. Given two functors F and G and a bifunctor $\star$, a new functor

$F \hat{\star} G$ can be defined by *lifting* $\star$ as follows:

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

Given a type A, a functor $A\star$ can also be defined by *sectioning* $\star$ as $A\star = \underline{A} \hat{\star} \mathsf{Id}$, which corresponds to treating as a constant the first parameter of the functor.

In the following we introduce a number of type constructors, such as products and coproducts. Each constructor comes equipped with its own set of combinators and laws. In the categorical setting, type constructors are universal constructions, and the laws can all be derived from their universal properties.

Products. The product of two types is defined as the cartesian product:

$$A \times B = \{(x, y) \mid x \in A, y \in B\}$$

We also define the *projection* functions and the *split* function combinator, denoted $\langle \cdot, \cdot \rangle$.

$$\pi_1\ (x, y) = x \qquad \langle f, g \rangle\ x = (f\ x, g\ x)$$
$$\pi_2\ (x, y) = y$$

The fact that the cartesian product is a categorical product in $\mathbf{Cpo}$ is justified by the following uniqueness law.

$$f = \langle g, h \rangle \quad \Leftrightarrow \quad \pi_1 \circ f = g \wedge \pi_2 \circ f = h \qquad \times\text{-UNIQ}$$

It is also useful to define a product function combinator as:

$$f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle \qquad \times\text{-DEF}$$

Observe that this definition allows us to see product as a bifunctor. Some useful functions that can be defined on products are the following natural isomorphisms.

$$\begin{aligned} \mathsf{swap} &: A \times B \to B \times A \\ \mathsf{swap} &= \langle \pi_2, \pi_1 \rangle \\ (f \times g) \circ \mathsf{swap} &= \mathsf{swap} \circ (g \times f) \qquad \text{swap-NAT} \\ \mathsf{assocr} &: (A \times B) \times C \to A \times (B \times C) \\ \mathsf{assocr} &= \langle \pi_1 \circ \pi_1, \pi_2 \times \mathsf{id} \rangle \\ (f \times (g \times h)) \circ \mathsf{assocr} &= \mathsf{assocr} \circ ((f \times g) \times h) \qquad \text{assocr-NAT} \end{aligned}$$

Strictness. From the point of view of program calculation, the major difference between using $\mathbf{Set}$ and $\mathbf{Cpo}$ as underlying category is that some of the laws that characterize the basic combinators will have strictness side-conditions. As we will shortly see, this is due to the fact that the separated sum is not a categorical coproduct in $\mathbf{Cpo}$. Strictness is defined as follows:

$$f \text{ strict} \quad \Leftrightarrow \quad f \circ \bot = \bot \qquad \text{strict-DEF}$$

Sometimes the notion of left-strictness will also be required:

$$f \text{ left-strict} \quad \Leftrightarrow \quad f \circ (\bot \times \mathsf{id}) = \bot \qquad \text{lstrict-DEF}$$
$$f \text{ strict} \quad \Leftarrow \quad f \text{ left-strict} \qquad \text{lstrict-STRICT}$$

Sums. In lazy functional languages coproducts are typically implemented as separated sums, with a new bottom element added:

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

Related to this definition we have the *injection* functions and the *either* combinator, denoted $[\cdot, \cdot]$.

$$\begin{array}{ll} i_1\ x = (0, x) & [f, g]\ \bot = \bot \\ i_2\ x = (1, x) & [f, g]\ (0, x) = f\ x \\ & [f, g]\ (1, x) = g\ x \end{array}$$

The separated sum is not a categorical coproduct in $\mathbf{Cpo}$ because the uniqueness law only holds for strict functions:

$$f = [g, h] \quad \Leftrightarrow \quad f \circ i_1 = g \ \wedge\ f \circ i_2 = h \ \wedge\ f \text{ strict} \qquad +\text{-UNIQ}$$

Likewise to products, the separated sum can be turned into a functor by defining its operation on arrows, which corresponds to introducing the sum function combinator:

$$f + g = [i_1 \circ f, i_2 \circ g] \qquad +\text{-DEF}$$

Exponentials. The exponentiation of type B to type A is defined as the set of all functions with domain A and codomain B:

$$B^A = \{f \mid f : A \to B\}$$

Associated to exponentials, are the *apply* function and the *curry* combinator (denoted $\overline{\cdot}$).

$$\begin{array}{rcl} \mathsf{ap}\ (f, x) & = & f\ x \\ \overline{f}\ x\ y & = & f\ (x, y) \end{array}$$

The following uniqueness law guarantees that this notion of exponentiation is a categorical exponentiation in $\mathbf{Cpo}$:

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

Finally, the definition of the exponentiation combinator allows us to turn this operation into a functor:

$$f^A = \overline{f \circ \mathsf{ap}} \qquad \wedge\text{-DEF}$$

Notice that when the type in superscript is not relevant we will use the symbol $\bullet$.

Miscellanea. The one element type is denoted by 1, and its unique element also by 1. Since the types A and $1 \to A$ are isomorphic, an element $e : A$ can be seen as a function $e : 1 \to A$. Given a function $f : A \to B$ (possibly an element if $A = 1$), we define the constant operator that always returns f as follows.

$$\begin{array}{rcl} \underline{f} & : & C \to B^A \\ \underline{f} & = & \overline{f \circ \pi_2} \end{array} \qquad \text{const-DEF}$$

Constants enjoy the following fusion property:

$$\underline{f} \circ g = \underline{f} \qquad \text{const-FUSION}$$

To facilitate the point-free treatment of conditional expressions, it is useful to define the *guard* combinator associated to a given predicate $p : A \to \mathsf{Bool}$. Assuming that $\mathsf{Bool} = 1 + 1$:

$$\begin{array}{rcl} p? & : & A \to A + A \\ p? & = & (\pi_1 + \pi_1) \circ \mathsf{distr} \circ \langle \mathsf{id}, p \rangle \end{array} \qquad \text{guard-DEF}$$

Note that if p returns $\perp$ for some input, then $p?$ will also return $\perp$. The function distr distributes a value with respect to a sum, and has type $A \times (B + C) \to A \times B + A \times C$.

2.2. Recursive Data Types

In a typed functional programming language a new data type is defined by declaring its constructors and the respective types. Theoretically, a data type is defined as the fixed point of a pattern (or *base functor*) that captures the signature of the constructors.

For example, suppose that the base functor of a data type T is $F_1 \hat{+} \ldots \hat{+} F_n$, such that

$$T = \mu(F_1 \hat{+} \ldots \hat{+} F_n)$$

This means that T has n constructors, each with type $C_i : F_i\ T \to T$. For instance, lists of type A can be defined by

$$\mathsf{List}\ A \quad = \quad \mu(\underline{1} \hat{+} \underline{A} \hat{\times} \mathsf{Id})$$

that captures the two typical constructors of this data type (recall that a function of type $1 \to \mathsf{List}\ A$ corresponds to an element of type $\mathsf{List}\ A$).

$$\begin{array}{rcl} \mathsf{nil} & : & \mathsf{List}\ A \\ \mathsf{cons} & : & A \times \mathsf{List}\ A \to \mathsf{List}\ A \end{array}$$

Since the functor of lists will be used many times during the paper, we will denote it just by FList_A for lists of type A, or simply FList when the type of the elements is clear from context. The action of this functor on types is given by $\mathsf{FList}_A\ B = 1 + A \times B$, and on functions by $\mathsf{FList}_A\ f = \mathsf{id} + \mathsf{id} \times f$. As a second example of a recursive type, we present the definition of binary leaf trees:

$$\begin{array}{rcl} \mathsf{LTree}\ A & = & \mu(\underline{A} \hat{+} \mathsf{Id} \hat{\times} \mathsf{Id}) \\ \mathsf{leaf} & : & A \to \mathsf{LTree}\ A \\ \mathsf{branch} & : & \mathsf{LTree}\ A \times \mathsf{LTree}\ A \to \mathsf{LTree}\ A \end{array}$$

In Haskell these data types can be implemented as follows (we use the standard Haskell definition of lists).

```
data [a] = [] | a : [a]
data LTree a = Leaf a | Branch (LTree a) (LTree a)
```

A *polynomial functor* is either the identity functor, the constant functor, or one of the liftings $F \hat{\times} G$ and $F \hat{+} G$, where F and G are polynomial functors. A *regular functor* can additionally be built from type functors (defined in the next section). Reynolds [29] proved that in $\mathbf{Cpo}$, given a locally continuous and strictness-preserving base functor F, there exists a unique data type μF and two unique strict functions $\mathsf{in}_F : F(\mu F) \to \mu F$ and $\mathsf{out}_F : \mu F \to F(\mu F)$ that are each other's inverse. Fokkinga and Meijer [11] showed that all polynomial, and even all regular functors, are locally continuous and strictness-preserving. This guarantees that, for example, all the above data types are well defined.

Notice that in_F is defined as the "either" of all constructors of μF. For example, for lists we have

$$\begin{array}{lcl} \mathsf{in}_{\mathsf{FList}} & : & 1 + A \times \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{in}_{\mathsf{FList}} & = & [\underline{\mathsf{nil}}, \mathsf{cons}] \end{array}$$

2.3. Catamorphisms and Maps

Each regular data type is characterized by a number of standard ways of traversing it recursively in order to produce a result. The most basic of these recursion patterns is iteration, which computes the result by replacing the constructors of the input data type by other functions. Such functions are called *folds* or *catamorphisms*, and can be written (without explicit recursion) using a higher-order operator that is written once and for all.

Given a function $g : F\ A \to A$ (sometimes called a *gene*), the catamorphism generated by g is generically defined as follows.

$$\begin{array}{lcl} (\!|g|\!)_F & : & \mu F \to A \\ (\!|g|\!)_F & = & \mu(\lambda f \cdot g \circ F f \circ \mathsf{out}_F) \end{array} \qquad \text{cata-DEF}$$

This function obeys the following uniqueness law, proved in [11] using fixpoint induction.

$$f = (\!|g|\!)_F \ \wedge\ g \text{ strict} \quad \Leftrightarrow \quad f \circ \mathsf{in}_F = g \circ F f \ \wedge\ f \text{ strict} \qquad \text{cata-UNIQ}$$

Instantiating the catamorphism definition to the particular case of lists results in the well known foldr operator on lists. To see that this is so, let us assume $g = [\underline{z}, f] : 1 + A \times B \to B$ where $z : B$ and $f : A \times B \to B$, and proceed with the following calculation. For simplicity, we omit the strictness conditions.

$$\begin{array}{ll} & \mathsf{foldr} = (\!|[\underline{z}, f]|\!)_{\mathsf{FList}} \\ = & \quad \{\ \text{cata-UNIQ}\ \} \\ & \mathsf{foldr} \circ \mathsf{in}_{\mathsf{FList}} = [\underline{z}, f] \circ \mathsf{FList}\ \mathsf{foldr} \\ = & \quad \{\ \text{definitions of FList and } \mathsf{in}_{\mathsf{FList}}\ \} \\ & \mathsf{foldr} \circ [\underline{\mathsf{nil}}, \mathsf{cons}] = [\underline{z}, f] \circ (\mathsf{id} + \mathsf{id} \times \mathsf{foldr}) \\ = & \quad \{\ +\text{-FUSION}, +\text{-ABSOR}, \times\text{-FUNCTOR}\ \} \\ & [\mathsf{foldr} \circ \underline{\mathsf{nil}}, \mathsf{foldr} \circ \mathsf{cons}] = [\underline{z}, f \circ (\mathsf{id} \times \mathsf{foldr})] \\ = & \quad \{\ +\text{-EQUAL}\ \} \\ & \mathsf{foldr} \circ \underline{\mathsf{nil}} = \underline{z} \ \wedge\ \mathsf{foldr} \circ \mathsf{cons} = f \circ (\mathsf{id} \times \mathsf{foldr}) \\ = & \quad \{\ \eta\text{-expansion}\ \} \\ & (\mathsf{foldr} \circ \underline{\mathsf{nil}})\ x = \underline{z}\ x \ \wedge\ (\mathsf{foldr} \circ \mathsf{cons})\ (x, xs) = (f \circ (\mathsf{id} \times \mathsf{foldr}))\ (x, xs) \\ = & \quad \{\ \text{definition of composition, product, and constants}\ \} \\ & \mathsf{foldr}\ \mathsf{nil} = z \ \wedge\ \mathsf{foldr}\ (\mathsf{cons}\ (x, xs)) = f\ (x, \mathsf{foldr}\ xs) \end{array}$$

These two equations are the same that define the Haskell function `foldr`: the constant z is returned when the list is empty, and f is used to combine the head of the list with the result of recursively processing the tail. The only difference is that `foldr` uses curried parameters.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

An example of a very simple catamorphism is the function that, given a list of naturals, returns the sum of all the elements in the list ($\mathsf{zero} : \mathsf{Int}$ denotes the natural 0 and $\mathsf{plus} : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ the sum function).

$$\begin{array}{lcl} \mathsf{sum} & : & \mathsf{List}\ \mathsf{Int} \to \mathsf{Int} \\ \mathsf{sum} & = & (\!|[\underline{\mathsf{zero}}, \mathsf{plus}]|\!) \end{array}$$

Catamorphisms obey the following fusion law (easily derived from the uniqueness law), very useful in program transformation, and which forms the basis for all the techniques presented in this paper.

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

The following strictness property can be derived from cata-UNIQ.

$$(\!|f|\!)_F \text{ strict} \quad \Leftarrow \quad f \text{ strict} \qquad \text{cata-STRICT}$$

Type Functors. The base functor of a parameterized data type $T\ A$ can be seen as the sectioning of a bifunctor $\star$. This allows us to see a parameterized data type as a *type functor*. Its action on objects is given by the definition $T\ A = \mu(A\star)$, and its action on functions corresponds to the standard map function of the type, that can be defined generically using a catamorphism:

$$\mathsf{map}_T\ f = (\!|\mathsf{in} \circ (f \star \mathsf{id})|\!)_{A\star} \quad \Leftarrow \quad T\ A = \mu(A\star) \qquad \text{map-DEF}$$

For example, the bifunctor that characterizes lists is defined by

$$\begin{array}{rcl} A\ \star_{\mathsf{List}}\ B & = & 1 + A \times B \\ f\ \star_{\mathsf{List}}\ g & = & \mathsf{id} + f \times g \end{array}$$

and allows us to define $\mathsf{List}\ A = \mu(A \star_{\mathsf{List}}) = \mu(\mathsf{FList}_A)$. As expected, the map function for lists can be defined as follows (obtained from map-DEF after some simple calculations).

$$\mathsf{map}_{\mathsf{List}}\ f = (\!|[\underline{\mathsf{nil}}, \mathsf{cons} \circ (f \times \mathsf{id})]|\!)$$

The following is a classic law relating fold and map.

$$(\!|f|\!)_{A\star} \circ \mathsf{map}_T\ g = (\!|f \circ (g \star \mathsf{id})|\!)_{A\star} \quad \Leftarrow \quad T\ A = \mu(A\star) \wedge f \text{ strict} \qquad \text{fold-map-FUSION}$$

This law can be used, for instance, to optimize (by calculation) the following two-pass function for computing the sum of the squares of a list. In the following $\mathsf{sq} = \mathsf{mult} \circ \langle \mathsf{id}, \mathsf{id} \rangle$, where $\mathsf{mult} : \mathsf{Int} \times \mathsf{Int} \rightarrow \mathsf{Int}$ is the function that implements the integer product.

$$\begin{array}{lcl} \mathsf{sumsq} & : & \mathsf{List\ Int} \rightarrow \mathsf{Int} \\ \mathsf{sumsq} & = & \mathsf{sum} \circ \mathsf{map}_{\mathsf{List}}\ \mathsf{sq} \end{array}$$

Applying fold-map-FUSION and +-ABSOR we get the following single pass implementation.

$$\begin{array}{lcl} \mathsf{sumsq} & : & \mathsf{List\ Int} \rightarrow \mathsf{Int} \\ \mathsf{sumsq} & = & (\!|[\underline{\mathsf{zero}}, \mathsf{plus} \circ (\mathsf{sq} \times \mathsf{id})]|\!) \end{array}$$

This example in itself is well-known in the program transformation community. It can be achieved for instance using fold/unfold transformation.

3. A Motivating Example

Consider the reverse function on lists. Obtaining the accumulator-based linear time version of this function from the single-argument quadratic time version is a classic example of a program transformation.

In this section we use this example to briefly review different transformation techniques for optimizing programs by introducing accumulating parameters. The resulting functions are called *accumulations*. We also introduce a composition operator, which enriches the point-free calculus allowing us to express certain properties (such as the associativity of a binary operator) in a higher-order setting.

3.1. Transformation with Fold/Unfold Rules

Our initial definition of reverse in Haskell is

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]
```

We will now apply fold/unfold rule-based transformation (see for instance [28] for a review) in order to obtain the efficient version.

```
    reverse l
=     { definition of reverse (using conditionals instead of pattern-matching) }
    if (l == []) then [] else reverse (tail l) ++ [head l]
=     { unfold }
    if (l == []) then []
    else (if (tail l == []) then []
          else reverse (tail (tail l)) ++ [head (tail l)]) ++ [head l]
=     { distributing (++ [head l]) over the conditional and applying associativity of ++ }
    if (l == []) then []
    else if (tail l == []) then [] ++ [head l]
         else reverse (tail (tail l)) ++ ([head (tail l)] ++ [head l])
```

These transformation steps are oriented by the so-called *forced folding* (or *need-for-folding*) principle [10], which states that after the unfold step, the program should be manipulated so that a folding step can be applied to a different sub-expression. Hopefully these manipulations will lead to improvements at all levels of the recursion tree.

Continuing with our example, at this point one would like to be able to fold the last expression above using the definition of reverse; however, the presence of the expression `[head l]` in both cases of the conditional expression prevents this step. We must appeal to the *generalization strategy* [8], according to which we introduce a new function definition

```
aux (l,y) = (reverse l) ++ y
```

This definition can indeed be transformed until a fold step is performed.

```
    aux (l,y)
=     { definition of aux }
    (reverse l) ++ y
=     { unfold }
    (if (l == []) then [] else (reverse (tail l) ++ [head l])) ++ y
=     { distributing ++ y over the conditional and applying associativity of ++ }
    if (l == []) then [] ++ y else reverse (tail l) ++ ([head l] ++ y)
=     { fold }
    if (l == []) then [] ++ y else aux (tail l, [head l] ++ y)
```

Next we use the definition of ++ to simplify the above expression, yielding the definition

```
aux (l,y) = if (l == []) then y else aux (tail l, (head l):y)
```

Finally, we notice that the ++ operator has a right-identity, allowing us to write

```
    reverse l
=     { right-identity of ++ }
    (reverse l) ++ []
=     { definition of aux }
    aux (l,[])
```

The definition of aux, together with this last equation, is the final result of the transformation. In Haskell one would write

```
reverse :: [a] -> [a]
reverse l = aux (l, [])

aux :: ([a], [a]) -> [a]
aux ([], y)   = y
aux (x:xs, y) = aux (xs, x:y)
```

Remark. Notice that the transformed program is *tail-recursive*, i.e. the result of the recursive call is passed directly as the result of the invoking call. Linear tail-recursive functions can be converted into iterative code (i.e. with recursion totally removed) using a straightforward transformation scheme.

Removal of recursion is a major goal of program transformation, even when it can be only partially achieved, as is the case with functions over trees. In section 4.4 it will be seen, for the case of binary trees, that only one of the two recursive calls is made tail-recursive.

The asymptotic improvement in the execution time is a somewhat casual side-effect of the transformation – it is a consequence of the associativity property of the append operator, and the fact that it runs in linear time on the size of its first argument. In section 4.1 we consider the transformation of the function which calculates the product of the numbers in a list. Since arithmetic product is calculated in constant time, this transformation does not alter the asymptotic execution time; it is however still a useful transformation since it produces a tail-recursive definition.

3.2. Transformation by Calculation

The first application of the calculational approach to program transformation, as popularized by Richard Bird and Lambert Meertens in the mid-80s, was precisely the derivation of functions with accumulations from inefficient specifications [4]. In this seminal paper, Bird introduced the fundamental idea behind this method for transformation: first the recursive functions are specified using a standard recursion pattern; then fusion is used together with the generalization strategy (as used in fold/unfold transformation), in order to derive a hopefully more efficient implementation with an accumulating parameter. We remark that fusion was here called *promotion* and the fold recursion pattern had not yet been isolated in a higher-order operator such as `foldr`.

The functions resulting from such transformations have two arguments (the second of which is the accumulator). In order to be able to write them using the fold recursion pattern, Bird resorted to currying: accumulations are written as higher-order folds, returning a function as result. Apart from some refinements in the basic laws and notation, this same technique was later used by several authors [25, 19, 26, 31]. However, in none of these works the calculations were done in pure point-free style, and in some of them the generic fusion law for catamorphisms presented in section 2.3 was not used. Instead, they use the pointwise specialization of this law for particular data types, such as the following for the foldr operator on lists.

$$\begin{array}{rl} & f\ (\mathsf{foldr}\ g\ e\ l) = \mathsf{foldr}\ h\ c\ l \\ & \Leftarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{foldr-FUSION} \\ f\ \text{strict} \wedge\ & f\ e = c \wedge \forall x, r \cdot f\ (g\ x\ r) = h\ x\ (f\ r) \end{array}$$

Turning now to our running example, we will start from where the application of the generalization strategy led us in the previous section (except that the new function is in curried style). Notice that in the remaining of the paper the name f_t will be used to denote the accumulation obtained by transforming f.

$$\mathsf{reverse}_t\ l\ y = (\mathsf{reverse}\ l)\ \mathsf{++}\ y$$

Following the approach just described, to obtain the desired accumulation we must now fuse the concatenation operator with the reverse function, using law foldr-FUSION. In order to do that it is necessary to redefine reverse using the foldr operator (notice that $\mathsf{wrap}\ x = \mathsf{cons}(x, \mathsf{nil})$):

$$\begin{array}{lcl} \mathsf{reverse} & : & \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{reverse} & = & \mathsf{foldr}\ (\lambda x r.\ r\ \mathsf{++}\ (\mathsf{wrap}\ x))\ \mathsf{nil} \end{array}$$

Dropping the accumulating parameter from our specification we get the following definition, where the Haskell prefix notation $(\mathsf{++})$ is used for the concatenation operator.

$$\mathsf{reverse}_t\ l = (\mathsf{++})\ (\mathsf{foldr}\ (\lambda xr.\ r\ \mathsf{++}\ (\mathsf{wrap}\ x))\ \mathsf{nil}\ l)$$

This is a suitable expression to apply foldr-FUSION, where f is the curried version of the concatenation operator. Given that ++ is a strict function in its first argument, there remain two premises of this law to verify, which will in turn allow us to determine c and h:

$$\begin{gathered}(\mathsf{++})\ \mathsf{nil} = c \\ \lambda xr.\ (\mathsf{++})\ (r\ \mathsf{++}\ (\mathsf{wrap}\ x)) = \lambda xr.\ h\ x\ ((\mathsf{++})\ r)\end{gathered}$$

where λ-abstraction has been used to encode universal quantification.

Since nil is a left-identity of concatenation, then $c = \mathsf{id}$. In order to determine h we calculate:

$$\begin{array}{ll} & \lambda xr.\ (\mathsf{++})\ (r\ \mathsf{++}\ (\mathsf{wrap}\ x)) \\ = & \quad \{\ \eta\text{-expansion}\ \} \\ & \lambda xr.\lambda y.\ (r\ \mathsf{++}\ (\mathsf{wrap}\ x))\ \mathsf{++}\ y \\ = & \quad \{\ \text{associativity of ++}\ \} \\ & \lambda xr.\lambda y.\ r\ \mathsf{++}\ ((\mathsf{wrap}\ x)\ \mathsf{++}\ y) \\ = & \quad \{\ \text{definitions of ++, wrap}\ \} \\ & \lambda xr.\lambda y.\ r\ \mathsf{++}\ (\mathsf{cons}(x, y)) \end{array}$$

It is now clear that h can be defined as

$$h\ x\ z = \lambda y.\ z\ (\mathsf{cons}(x, y))$$

The result of applying the fusion law is thus the following higher-order fold:

$$\begin{array}{lcl} \mathsf{reverse}_t & : & \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{reverse}_t & = & \mathsf{foldr}\ (\lambda xzy.\ z\ (\mathsf{cons}(x, y)))\ \mathsf{id} \end{array}$$

After expanding the definition of foldr we obtain the curried version of the function in the previous section, to be invoked as `reverse l = reverse_t l []`:

```
reverse_t :: [a] -> [a] -> [a]
reverse_t [] y     = y
reverse_t (x:xs) y = reverse_t xs (x:y)
```

To sum up, the creative step involved in this technique is exactly the same as when using fold/unfold transformations – writing the specification corresponds to using the generalization strategy. However, for the particular technique of accumulations, experienced functional programmers should have no problem in writing specifications directly. A major advantage of the calculational approach is that by structuring recursion in fixed patterns, it is possible, as will be largely exemplified in this paper, to define laws that combine in a single *shortcut* step whole sequences of transformation steps.

3.3. Transformation in the Point-free Style

The third method, which will be used extensively in the remaining of the paper, consists in applying the same transformation principle as in the previous section, except that all the calculations are done in the point-free style with generic recursion patterns. Before this can be done, the initial specification needs to be written according to the principles exposed in section 2. Programmers with experience in the point-free style may be able to write this definition directly. In the following we present (using our running example) a systematic method for converting recursive functions into the point-free style.

First, we state the pointwise equations implicit in the Haskell definition. We denote the uncurried version of the concatenation operator by cat.

$$\begin{array}{lcl} \mathsf{reverse\ nil} & = & \mathsf{nil} \\ \mathsf{reverse}\ (\mathsf{cons}(x, xs)) & = & \mathsf{cat}\ (\mathsf{reverse}\ xs, \mathsf{wrap}\ x) \end{array}$$

The next step eliminates the variables in order to obtain a point-free specification. This is the most creative step, and usually implies the (sometimes not so trivial) introduction of "housekeeping" functions, since the parameters must be equally structured in both sides of each equation. In this particular case we use swap to get the following definition (recall that a constant can be seen as a function with domain 1).

$$\begin{array}{lcl} (\mathsf{reverse} \circ \underline{\mathsf{nil}})\ 1 & = & \underline{\mathsf{nil}}\ 1 \\ (\mathsf{reverse} \circ \mathsf{cons})\ (x, xs) & = & (\mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{reverse}))\ (x, xs) \end{array}$$

After eliminating the arguments we proceed with standard point-free calculations until we get a specification with the form $\mathsf{reverse} \circ \mathsf{in}_{\mathsf{FList}} = g \circ \mathsf{FList\ reverse}$, which by the uniqueness law of catamorphisms allows us to define $\mathsf{reverse} = (\!|g|\!)$.

$$\begin{array}{ll} & \mathsf{reverse} \circ \underline{\mathsf{nil}} = \underline{\mathsf{nil}} \ \wedge\ \mathsf{reverse} \circ \mathsf{cons} = \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{reverse}) \\ = & \quad \{ +\text{-EQUAL} \} \\ & [\mathsf{reverse} \circ \underline{\mathsf{nil}}, \mathsf{reverse} \circ \mathsf{cons}] = [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{reverse})] \\ = & \quad \{ +\text{-FUSION}, \times\text{-FUNCTOR} \} \\ & \mathsf{reverse} \circ [\underline{\mathsf{nil}}, \mathsf{cons}] = [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{reverse})] \\ = & \quad \{ +\text{-ABSOR} \} \\ & \mathsf{reverse} \circ [\underline{\mathsf{nil}}, \mathsf{cons}] = [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \mathsf{reverse}) \\ = & \quad \{ \text{definitions of FList and } \mathsf{in}_{\mathsf{FList}} \} \\ & \mathsf{reverse} \circ \mathsf{in}_{\mathsf{FList}} = [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \circ \mathsf{FList\ reverse} \end{array}$$

Now, since reverse is a strict function, cata-UNIQ can be applied to produce the desired point-free definition.

$$\begin{array}{lcl} \mathsf{reverse} & : & \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{reverse} & = & (\!|[\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})]|\!)_{\underline{1}\hat{+}\underline{A}\hat{\times}\mathsf{Id}} \end{array}$$

Turning to the specification that will allow us to derive the accumulation, it can now be written in point-free style as

$$\mathsf{reverse}_t = \overline{\mathsf{cat}} \circ \mathsf{reverse}$$

The derivation will be based on the generic fusion law cata-FUSION. According to this law (and since $\overline{\mathsf{cat}}$ is strict), in order to obtain the desired definition of $\mathsf{reverse}_t$ as a catamorphism $\mathsf{reverse}_t = (\!|h|\!)_{\mathsf{FList}}$,

we must find a function h such that

$$\overline{\mathsf{cat}} \circ [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] = h \circ \mathsf{FList}\ \overline{\mathsf{cat}} = h \circ (\mathsf{id} + \mathsf{id} \times \overline{\mathsf{cat}})$$

In both fold/unfold and the pointwise calculational transformations seen in the previous sections, one of the major steps was the application of the associativity property of cat (this is in general the case for all transformations involving accumulations). So the question arises of how to express this property in point-free style.

Consider an arbitrary operator $\oplus$. One possibility for expressing its associativity is to use the equation

$$\oplus \circ (\mathsf{id} \times \oplus) \circ \mathsf{assocr} = \oplus \circ (\oplus \times \mathsf{id})$$

This formulation is not very practical because the operator that will be fused is in curried form. Bearing in mind that it is our goal in this paper to enrich the calculus so as to simplify the derivations as much as possible, we need to introduce operators particularly tailored to express properties in a higher-order setting. For the particular case of associativity, it suffices to introduce an uncurried composition operator defined as

$$\mathsf{comp}\ (f, g) = f \circ g$$

or in point-free style (using, as expected, the exponential combinators):

$$\begin{array}{lcl} \mathsf{comp} & : & (C^B \times B^A) \to C^A \\ \mathsf{comp} & = & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr}} \end{array} \qquad \text{comp-DEF}$$

Using this combinator, associativity of $\oplus$ can be expressed more usefully by the following equation.

$$\overline{\oplus} \circ \oplus = \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \qquad \oplus\text{-ASSOC}$$

The following calculation shows that the latter formulation is a consequence of the former.

$$\begin{array}{ll} & \overline{\oplus} \circ \oplus \\ = & \quad \{\ \wedge\text{-FUSION}\ \} \\ & \overline{\oplus \circ (\oplus \times \mathsf{id})} \\ = & \quad \{\ \oplus \circ (\mathsf{id} \times \oplus) \circ \mathsf{assocr} = \oplus \circ (\oplus \times \mathsf{id})\ \} \\ & \overline{\oplus \circ (\mathsf{id} \times \oplus) \circ \mathsf{assocr}} \\ = & \quad \{\ \wedge\text{-CANCEL}, \times\text{-FUNCTOR}\ \} \\ & \overline{\mathsf{ap} \circ (\overline{\oplus} \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{ap}) \circ (\mathsf{id} \times (\overline{\oplus} \times \mathsf{id})) \circ \mathsf{assocr}} \\ = & \quad \{\ \times\text{-FUNCTOR}\ \} \\ & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ (\overline{\oplus} \times (\overline{\oplus} \times \mathsf{id})) \circ \mathsf{assocr}} \\ = & \quad \{\ \mathsf{assocr}\text{-NAT}\ \} \\ & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr} \circ ((\overline{\oplus} \times \overline{\oplus}) \times \mathsf{id})} \\ = & \quad \{\ \wedge\text{-FUSION}\ \} \\ & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr}} \circ (\overline{\oplus} \times \overline{\oplus}) \\ = & \quad \{\ \mathsf{comp}\text{-DEF}\ \} \\ & \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \end{array}$$

Equipped with this formulation of associativity, calculating the accumulation becomes very simple. Notice that the fact that nil is a left-identity of cat can be expressed by the equation $\overline{\mathsf{cat}} \circ \underline{\mathsf{nil}} = \underline{\mathsf{id}}$, and the effect of concatenation with the singleton list can be described by the equation $\overline{\mathsf{cat}} \circ \mathsf{wrap} = \overline{\mathsf{cons}}$.

$$
\begin{array}{ll}
 & \overline{\mathsf{cat}} \circ [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \\
= & \quad \{ +\text{-FUSION} \} \\
 & [\overline{\mathsf{cat}} \circ \underline{\mathsf{nil}}, \overline{\mathsf{cat}} \circ \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \\
= & \quad \{ \mathsf{nil} \text{ is a left-identity of } \mathsf{cat}, \text{ cat-ASSOC} \} \\
 & [\underline{\mathsf{id}}, \mathsf{comp} \circ (\overline{\mathsf{cat}} \times \overline{\mathsf{cat}}) \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \\
= & \quad \{ \text{swap-NAT}, \times\text{-FUNCTOR} \} \\
 & [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cat}} \circ \mathsf{wrap} \times \overline{\mathsf{cat}})] \\
= & \quad \{ \text{concatenation with a singleton list}, \times\text{-FUNCTOR} \} \\
 & [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id}) \circ (\mathsf{id} \times \overline{\mathsf{cat}})] \\
= & \quad \{ +\text{-ABSOR} \} \\
 & [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \overline{\mathsf{cat}})
\end{array}
$$

The result of the transformation is thus

$$
\begin{array}{lcl}
\mathsf{reverse}_t & : & \mathsf{List}\ A \to (\mathsf{List}\ A \to \mathsf{List}\ A) \\
\mathsf{reverse}_t & = & (\!|[\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})]|\!)_{\underline{1} \hat{+} \underline{A} \hat{\times} \mathsf{Id}}
\end{array}
$$

To see that this is the expected point-free definition of $\mathsf{reverse}_t$, we convert it back to pointwise. The calculation is similar to the one concerning `foldr` in section 2.3.

$$
\begin{array}{ll}
 & \mathsf{reverse}_t = (\!|[\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})]|\!)_{\mathsf{FList}} \\
= & \quad \{ \text{cata-UNIQ, definitions of FList and } \mathsf{in}_{\mathsf{FList}} \} \\
 & \mathsf{reverse}_t \circ [\underline{\mathsf{nil}}, \mathsf{cons}] = [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \mathsf{reverse}_t) \\
= & \quad \{ +\text{-FUSION}, \times\text{-FUNCTOR} \} \\
 & [\mathsf{reverse}_t \circ \underline{\mathsf{nil}}, \mathsf{reverse}_t \circ \mathsf{cons}] = [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{reverse}_t)] \\
= & \quad \{ +\text{-EQUAL} \} \\
 & \mathsf{reverse}_t \circ \underline{\mathsf{nil}} = \underline{\mathsf{id}} \ \wedge \ \mathsf{reverse}_t \circ \mathsf{cons} = \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{reverse}_t) \\
= & \quad \{ \eta\text{-expansion, definitions of the basic combinators} \} \\
 & \mathsf{reverse}_t\ \mathsf{nil} = \mathsf{id} \ \wedge \ \mathsf{reverse}_t\ (\mathsf{cons}\ (x, xs)) = (\mathsf{reverse}_t\ xs) \circ (\overline{\mathsf{cons}}\ x) \\
= & \quad \{ \eta\text{-expansion, definitions of the basic combinators} \} \\
 & \mathsf{reverse}_t\ \mathsf{nil}\ y = \mathsf{id}\ y \ \wedge \ \mathsf{reverse}_t\ (\mathsf{cons}\ (x, xs))\ y = \mathsf{reverse}_t\ xs\ (\mathsf{cons}\ (x, y))
\end{array}
$$

Remark. The following calculation shows that this solution to the premises of cata-FUSION is not unique.

$$
\begin{array}{ll}
 & \overline{\mathsf{cat}} \circ [\underline{\mathsf{nil}}, \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \\
= & \quad \{ +\text{-FUSION} \} \\
 & [\overline{\mathsf{cat}} \circ \underline{\mathsf{nil}}, \overline{\mathsf{cat}} \circ \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \\
= & \quad \{ \mathsf{nil} \text{ is a left-identity of } \mathsf{cat}, \wedge\text{-CANCEL} \} \\
 & [\underline{\mathsf{id}}, \overline{\mathsf{cat}} \circ \mathsf{ap} \circ (\overline{\mathsf{cat}} \times \mathsf{id}) \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \\
= & \quad \{ \mathsf{swap}\text{-NAT}, \times\text{-FUNCTOR} \} \\
 & [\underline{\mathsf{id}}, \overline{\mathsf{cat}} \circ \mathsf{ap} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \overline{\mathsf{cat}})] \\
= & \quad \{ \times\text{-FUNCTOR}, +\text{-ABSOR} \} \\
 & [\underline{\mathsf{id}}, \overline{\mathsf{cat}} \circ \mathsf{ap} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \overline{\mathsf{cat}})
\end{array}
$$

This leads to the following definition of the accumulation.

$$
\begin{array}{lcl}
\mathsf{reverse}_t & : & \mathsf{List}\ A \to (\mathsf{List}\ A \to \mathsf{List}\ A) \\
\mathsf{reverse}_t & = & (\!|[\underline{\mathsf{id}}, \overline{\mathsf{cat}} \circ \mathsf{ap} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})]|\!)
\end{array}
$$

and in pointwise:

```
reverse_t :: [a] -> [a] -> [a]
reverse_t [] y     = y
reverse_t (x:xs) y = (reverse_t xs [x]) ++ y
```

This is of course a useless transformation – the resulting function runs in quadratic time and is not tail-recursive. This shows that some notion of a strategy is necessary for the calculations to be relevant for our goals. The distinctive feature of a useful transformation in this particular case is its use of the associativity property of append, not used in the latter transformation.

4. Calculating Accumulations in the Point-free Style

The methodology presented for deriving accumulations using the point-free style is still not very amenable for mechanization: the calculations require human intervention, not only to decide which law to apply at each point, but also to identify a good target to guide the derivation. As we have seen, it is possible to derive accumulations that are not "better" than the original functions. As such, in this section we will present a set of transformation schemes, categorized by data type, whose derivation is performed once and for all, and that guarantee the usefulness of the transformation. To apply these transformation schemes one has to prove very few side conditions (typically, just the associativity of some operator). These schemes could thus be used as *shortcut* optimization rules in an automatic transformation system. We also demonstrate the application of these rules in a substantial number of examples.

We first present a transformation scheme that encapsulates the methodology for deriving accumulations in the calculational style using fusion. Among others, it is presented also in [7, 19]. We adapt it here to the **Cpo** setting. This scheme is too general to be useful, but will later be instantiated for specific data types.

Proposition 4.1. Let $\oplus : A \times B \to B$ be a left-strict operator with right-identity $e : B$; then for all x

$$(\!|f|\!)_F\ x = (\!|g|\!)_F\ x\ e \quad \Leftarrow \quad \overline{\oplus} \circ f = g \circ F\ \overline{\oplus}$$

Proof:
This equation can be expressed in point-free as $(\!|f|\!) = \mathsf{ap} \circ \langle (\!|g|\!), \underline{e} \rangle$, and proved as follows. Notice that $(\!|g|\!) = \overline{\oplus} \circ (\!|f|\!)$ is a direct consequence of the hypothesis and cata-FUSION (strictness of $\overline{\oplus}$ results from the left-strictness of $\oplus$ and $\wedge$-STRICT).

$$
\begin{array}{ll}
 & (\!|f|\!) \\
= & \quad \{\, e \text{ is a right-identity of } \oplus \,\} \\
 & \oplus \circ \langle \mathsf{id}, \underline{e} \rangle \circ (\!|f|\!) \\
= & \quad \{\, \times\text{-FUSION, const-FUSION} \,\} \\
 & \oplus \circ \langle (\!|f|\!), \underline{e} \rangle \\
= & \quad \{\, \wedge\text{-CANCEL} \,\} \\
 & \mathsf{ap} \circ (\overline{\oplus} \times \mathsf{id}) \circ \langle (\!|f|\!), \underline{e} \rangle \\
= & \quad \{\, \times\text{-ABSOR} \,\} \\
 & \mathsf{ap} \circ \langle \overline{\oplus} \circ (\!|f|\!), \underline{e} \rangle \\
= & \quad \{\, (\!|g|\!) = \overline{\oplus} \circ (\!|f|\!) \,\} \\
 & \mathsf{ap} \circ \langle (\!|g|\!), \underline{e} \rangle
\end{array}
$$

□

4.1. Tail-recursive Accumulations over Lists: Associative Operators

We start with the most classic example of applying the accumulation strategy: optimizing the iteration of an associative operator over a list.

Proposition 4.2. Given a left-strict associative operator $\oplus : B \times B \to B$ with right identity e, an element $c : B$, a function $f : A \to B$, and two functions defined over lists as

$$
\begin{array}{lcllcl}
h & : & \mathsf{List}\ A \to B & \qquad h_t & : & \mathsf{List}\ A \to B \to B \\
h & = & (\!|[\underline{c}, \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id})]|\!) & \qquad h_t & = & (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \mathsf{id})]|\!)
\end{array}
$$

Then $h\ l = h_t\ l\ e$.

Proof:
The following calculation allows us to apply proposition 4.1, with F instantiated to the base functor of lists.

$$
\begin{array}{ll}
 & \overline{\oplus} \circ [\underline{c}, \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id})] \\
= & \quad \{\, +\text{-FUSION, } \oplus \text{ left-strict, } \wedge\text{-STRICT} \,\} \\
 & [\overline{\oplus} \circ \underline{c}, \overline{\oplus} \circ \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id})] \\
= & \quad \{\, \oplus\text{-ASSOC} \,\} \\
 & [\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \circ \mathsf{swap} \circ (f \times \mathsf{id})] \\
= & \quad \{\, \mathsf{swap}\text{-NAT, } \times\text{-FUNCTOR} \,\} \\
 & [\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \overline{\oplus})] \\
= & \quad \{\, \times\text{-FUNCTOR, } +\text{-ABSOR} \,\} \\
 & [\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \overline{\oplus})
\end{array}
$$

□

In pointwise this proposition captures the transformation of the function

$$\begin{array}{lcl} h\ \mathsf{nil} & = & c \\ h\ (\mathsf{cons}(x, xs)) & = & (h\ xs) \oplus (f\ x) \end{array}$$

into the tail-recursive

$$\begin{array}{lcl} h_t\ \mathsf{nil}\ y & = & c \oplus y \\ h_t\ (\mathsf{cons}(x, xs))\ y & = & h_t\ xs\ ((f\ x) \oplus y) \end{array}$$

Remark. This proposition is strongly related to the *first duality theorem* of [6] that states the conditions under which a `foldr` can be converted into a `foldl`. The latter function is well known in the functional programming community, and encodes precisely a (restricted) notion of tail-recursive accumulations over lists. It is defined in the standard libraries of Haskell as follows.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

In the point-free calculus it is easier to write the following version of foldl that takes its two last arguments in the reverse order, and where $\oplus : A \times B \to B$ has the arguments swapped with respect to `f`.

$$\begin{array}{lcl} \mathsf{foldl}\ \oplus & : & \mathsf{List}\ A \to B \to B \\ \mathsf{foldl}\ \oplus & = & (\!|[\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \times \mathsf{id})]|\!) \end{array}$$

Given this definition, and the definition of foldr of section 2.3, the first duality theorem says that, given an associative operator $\oplus$ with unit e, we have

$$\mathsf{foldr}\ (\oplus \circ \mathsf{swap})\ e = \mathsf{foldl}\ \oplus\ e$$

From this theorem and some facts about maps, we could easily derive the following alternative formulation of proposition 4.2.

$$\mathsf{foldr}\ (\oplus \circ \mathsf{swap})\ c\ (\mathsf{map}_{\mathsf{List}} f\ l) = c \oplus (\mathsf{foldl}\ \oplus\ (\mathsf{map}_{\mathsf{List}} f\ l)\ e)$$

Example 4.1. (Reverse)
It is immediate to see that the reverse function of section 3.3 can be transformed directly by applying proposition 4.2, with the expected result.

Example 4.2. (Product)
We now want to apply proposition 4.2 to derive a tail-recursive implementation of the function that multiplies all the numbers in a list.

```
product :: [Int] -> Int
product []     = 1
product (x:xs) = x * (product xs)
```

In point-free style this can be written as

$$\begin{array}{lcl} \mathsf{product} & : & \mathsf{List\ Int} \to \mathsf{Int} \\ \mathsf{product} & = & (\!|[\underline{\mathsf{one}}, \mathsf{mult}]|\!) \end{array}$$

At first examination it seems that proposition 4.2 cannot be applied; however, since mult is a commutative operator and $\mathsf{id}_{\mathsf{Int}} \times \mathsf{id}_{\mathsf{Int}} = \mathsf{id}_{\mathsf{Int}\times\mathsf{Int}}$ the above definition is equivalent to the following.

$$\begin{array}{lcl} \mathsf{product} & : & \mathsf{List\ Int} \to \mathsf{Int} \\ \mathsf{product} & = & (\!|[\underline{\mathsf{one}}, \mathsf{mult} \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{id})]|\!) \end{array}$$

Now proposition 4.2 can be applied straightforwardly, resulting in the following accumulation (notice that since one is the unit of mult then $\overline{\mathsf{mult}} \circ \underline{\mathsf{one}} = \underline{\mathsf{id}}$).

$$\begin{array}{lcl} \mathsf{product}_t & : & \mathsf{List\ Int} \to \mathsf{Int} \to \mathsf{Int} \\ \mathsf{product}_t & = & (\!|[\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{mult}} \times \mathsf{id})]|\!) \end{array}$$

The final tail-recursive implementation in Haskell is

```
product :: [Int] -> Int
product l = product_t l 1

product_t :: [Int] -> Int -> Int
product_t [] y     = y
product_t (x:xs) y = product_t xs (x*y)
```

Notice that we could also use `foldr` to define the accumulations. In this case we would have:

```
product_t :: [Int] -> Int -> Int
product_t = foldr (\x r y -> r (x*y)) id
```

This example shows that it is not always immediate to apply the transformation rule of proposition 4.2. For many operators (namely for commutative operators) the redefinition of the initial catamorphism is trivial; the next example presents a situation where this is not so obvious.

Example 4.3. (Insertion Sort)
Consider the following definition of insertion sort.

```
isort :: (Ord a) => [a] -> [a]
isort []     = []
isort (x:xs) = insert x (isort xs)
```

where `insert` is the function that performs an ordered insertion in a list. This function can be written as the catamorphism

$$\mathsf{isort} = (\!|[\underline{\mathsf{nil}}, \mathsf{insert}]|\!)$$

where the type of insert is $A \times \mathsf{List}\ A \to \mathsf{List}\ A$. This is clearly not an associative operator (nor would it be with a different order of the arguments). In order to apply the transformation scheme, the definition has to be considerably modified. First, we notice that

$$\mathsf{insert} = \mathsf{merge} \circ (\mathsf{wrap} \times \mathsf{id})$$

where merge is the (associative) merge function on sorted lists, which has the empty list as its right identity. Taking into account that merge is also commutative, insertion sort can be redefined as follows.

$$\mathsf{isort} = (\!|[\underline{\mathsf{nil}}, \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})]|\!)$$

This definition is suitable for transformation using proposition 4.2, with the result (notice that $\overline{\mathsf{insert}} = \overline{\mathsf{merge}} \circ \mathsf{wrap}$)

$$\mathsf{isort}_t = (\!|[\mathsf{id}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{insert}} \times \mathsf{id})]|\!)$$

Thus the tail-recursive definition:

```
isort :: (Ord a) => [a] -> [a]
isort l = isort_t l []

isort_t :: (Ord a) => [a] -> [a] -> [a]
isort_t [] y     = y
isort_t (x:xs) y = isort_t xs (insert x y)
```

This example illustrates that the application of proposition 4.2 may require the introduction of new functions (as in the generalization strategy) with which to write the initial catamorphism. These may possibly be later eliminated after the shortcut is applied.

4.2. Tail-recursive Accumulations over Lists: Associative Dual Operators

The previous result can be generalized in order to be applicable to a slightly more general class of programs. A binary operator $\oplus$ is said to have an *associative dual* operator $\odot$ [3] if

$$(x \oplus y) \oplus z = x \oplus (y \odot z)$$

In point-free notation the above equality can be written as

$$\overline{\oplus} \circ \oplus = \mathsf{comp} \circ (\overline{\oplus} \times \overline{\odot})$$

Proposition 4.3. Given a left-strict operator $\oplus : B \times C \to B$ with right identity e and associative dual operator $\odot : C \times C \to C$, an element $c : B$, a function $f : A \to C$, and two functions defined as

$$\begin{array}{lcllcl} h & : & \mathsf{List}\ A \to B & \qquad h_t & : & \mathsf{List}\ A \to C \to B \\ h & = & (\!|[\underline{c}, \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id})]|\!) & \qquad h_t & = & (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\odot} \circ f \times \mathsf{id})]|\!) \end{array}$$

Then $h\ l = h_t\ l\ e$.

Proof:
Similar to proposition 4.2 (the associative dual law has a similar formulation to associativity). □

Example 4.4. (Tree Sorts)
Consider again the definition of insertion sort from example 4.3:

$$\mathsf{isort} = (\!|[\underline{\mathsf{nil}}, \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})]|\!)$$

There is no reason why an accumulator of a different type cannot be used, such as a binary tree. Without looking at the details of how such trees are implemented, we consider an abstract data type $\mathsf{Tree}\ A$ to be used as accumulator. This type comes equipped with the following functions.

$\mathsf{treeToList} : \mathsf{Tree}\ A \to \mathsf{List}\ A$	(obtains a sorted list of the elements stored in the tree)
$\mathsf{mkTree} : A \to \mathsf{Tree}\ A$	(produces a tree from a single element)
$\mathsf{mergeTree} : \mathsf{Tree}\ A \times \mathsf{Tree}\ A \to \mathsf{Tree}\ A$	(merges two trees)

Using these we define the function that inserts an element in a tree as

$$\begin{array}{lcl} \mathsf{insertTree} & : & A \times \mathsf{Tree}\ A \to \mathsf{Tree}\ A \\ \mathsf{insertTree} & = & \mathsf{mergeTree} \circ (\mathsf{mkTree} \times \mathsf{id}) \end{array}$$

In order to use trees as accumulators, we now simply try to rewrite the definition of isort such that wrap can be replaced by mkTree. The following calculation uses as hypothesis a property that we can reasonably expect to be verified by our implementation of trees, namely $\mathsf{treeToList} \circ \mathsf{mkTree} = \mathsf{wrap}$.

$$\begin{array}{ll} & \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\ = & \quad \{\ \mathsf{treeToList} \circ \mathsf{mkTree} = \mathsf{wrap}\ \} \\ & \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{treeToList} \circ \mathsf{mkTree} \times \mathsf{id}) \\ = & \quad \{\ \times\text{-FUNCTOR}\ \} \\ & \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{treeToList} \times \mathsf{id}) \circ (\mathsf{mkTree} \times \mathsf{id}) \\ = & \quad \{\ \mathsf{swap}\text{-NAT}\ \} \\ & \mathsf{merge} \circ (\mathsf{id} \times \mathsf{treeToList}) \circ \mathsf{swap} \circ (\mathsf{mkTree} \times \mathsf{id}) \end{array}$$

If we define $\oplus = \mathsf{merge} \circ (\mathsf{id} \times \mathsf{treeToList})$ we may then write

$$\mathsf{isort} = (\!|[\underline{\mathsf{nil}}, \oplus \circ \mathsf{swap} \circ (\mathsf{mkTree} \times \mathsf{id})]|\!)$$

and moreover the reader will have no difficulty in accepting that the intended implementation of trees should be such that mergeTree is the associative dual operator of $\oplus$, i.e.

$$(l \oplus t_1) \oplus t_2 = l \oplus (\mathsf{mergeTree}\ t_1\ t_2)$$

The conditions of proposition 4.3 are then verified, yielding the tail-recursive function:

$$\mathsf{sort}_t = (\!|[\overline{\oplus} \circ \underline{\mathsf{nil}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{mergeTree}} \circ \mathsf{mkTree} \times \mathsf{id})]|\!)$$

This definition may be further simplified taking into account $\overline{\oplus} \circ \underline{\mathsf{nil}} = \underline{\mathsf{treeToList}}$ and the definition of insertTree.

$$\mathsf{sort}_t = (\!|[\underline{\mathsf{treeToList}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{insertTree}} \times \mathsf{id})]|\!)$$

In Haskell we have

```
sort_t :: (Ord a) => [a] -> Tree a -> [a]
sort_t [] y     = treeToList y
sort_t (x:xs) y = sort_t xs (insertTree x y)
```

Possible implementations of trees include ordinary binary search trees (with the obvious ordered insertion operation and treeToList implemented by an inorder traversal) and leaf trees (with treeToList implemented as a fold that for each node converts both left and right sub-trees to sorted lists and then merges them together). In both cases, if insertion operations are designed to preserve a balanced shape, $\mathcal{O}(n\ \lg n)$ sorting algorithms result.

4.3. Other Accumulations over Lists

We will now further generalize our transformation scheme, to allow for transformations that, while still based on compositions with associative operators (or operators that have an associative dual), do not result in tail-recursive functions.

Proposition 4.4. Given a left-strict operator $\oplus : B \times C \to B$ with right identity e and associative dual operator $\odot : C \times C \to C$, an element $c : B$, a function $f : A \to C$, and two functions defined as

$$\begin{array}{lcl@{\qquad}lcl} h & : & \mathsf{List}\ A \to B & h_t & : & \mathsf{List}\ A \to C \to B \\ h & = & (\!|[\underline{c}, \oplus \circ \mathsf{swap} \circ (f \times g)]|\!) & h_t & = & (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\odot} \circ f \times k)]|\!) \end{array}$$

where g and k are functions such that $\overline{\oplus} \circ g = k \circ \overline{\oplus}$, then $h\ l = h_t\ l\ e$.

Proof:
Again, this is a similar proof to proposition 4.2, where the equality $\overline{\oplus} \circ g = k \circ \overline{\oplus}$ is used . □

The next example illustrates the application of this shortcut law; it also introduces a new higher-order point-free operator (in the same spirit as comp). Again, it becomes clear that enriching the calculus with such operators simplifies the calculations considerably.

Example 4.5. (Initial Sums)
Consider the following function (a slight variation of an example from [19]) that computes the initial sums of a list.

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

This definition can be optimized by introducing an accumulating parameter that at each point will store the sum of all previous elements in the list. This accumulation can be calculated by fusion from the equation $\mathsf{isums}_t = \overline{\oplus} \circ \mathsf{isums}$, where

$$\begin{array}{rcl} \oplus & : & \mathsf{List\ Int} \times \mathsf{Int} \to \mathsf{List\ Int} \\ \oplus\ (l, x) & = & \mathsf{map}_{\mathsf{List}}\ (\overline{\mathsf{plus}}\ x)\ l \end{array}$$

Instead of applying fusion directly, we will use proposition 4.4. We begin by defining isums in the point-free style as a catamorphism using the operator $\oplus$.

$$\begin{array}{lcl} \mathsf{isums} & : & \mathsf{List\ Int} \rightarrow \mathsf{List\ Int} \\ \mathsf{isums} & = & (\!|[\underline{\mathsf{nil}}, \oplus \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{cons} \circ \langle \underline{\mathsf{zero}}, \mathsf{id} \rangle)]|\!) \end{array}$$

In order to apply the transformation we must identify the right identity of $\oplus$ and its associative dual. The former is obviously zero; the latter is $\odot = \mathsf{plus}$ since the following property holds.

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

To keep the presentation short, rather than expressing the operator $\oplus$ in the point-free style and proving certain obvious properties about it, we will take these for granted and concentrate on the part of the point-free proof that is of interest to us. We must identify a function k such that

$$k \circ \overline{\oplus} = \overline{\oplus} \circ \mathsf{cons} \circ \langle \underline{\mathsf{zero}}, \mathsf{id} \rangle$$

For that, we need to express the equation $(\mathsf{cons}\ (x, l)) \oplus y = \mathsf{cons}\ (\mathsf{plus}\ (x, y), l \oplus y)$ in the point-free calculus. The obvious choice is to express this as $\oplus \circ (\mathsf{cons} \times \mathsf{id}) = \mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ \langle \pi_1 \times \mathsf{id}, \pi_2 \times \mathsf{id} \rangle$. However, likewise to associativity, a formulation of this property involving the curried version of the operator will ease the burden of the calculations. This implies introducing a new split combinator defined pointwise as $\mathsf{split}\ (f, g) = \langle f, g \rangle$, and in point-free by the following equation.

$$\begin{array}{lcl} \mathsf{split} & : & (B^A \times C^A) \rightarrow (B \times C)^A \\ \mathsf{split} & = & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ \langle \pi_1 \times \mathsf{id}, \pi_2 \times \mathsf{id} \rangle} \end{array} \qquad \text{split-DEF}$$

Using this combinator, the above property can be expressed by the equation

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

as the following calculation shows.

$$\begin{array}{ll} & \overline{\oplus} \circ \mathsf{cons} \\ = & \quad \{\ \wedge\text{-FUSION}\ \} \\ & \overline{\oplus \circ (\mathsf{cons} \times \mathsf{id})} \\ = & \quad \{\ \oplus \circ (\mathsf{cons} \times \mathsf{id}) = \mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ \langle \pi_1 \times \mathsf{id}, \pi_2 \times \mathsf{id} \rangle\ \} \\ & \overline{\mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ \langle \pi_1 \times \mathsf{id}, \pi_2 \times \mathsf{id} \rangle} \\ = & \quad \{\ \wedge\text{-CANCEL}, \times\text{-FUNCTOR}, \times\text{-ABSOR}\ \} \\ & \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ \langle \overline{\mathsf{plus}} \circ \pi_1 \times \mathsf{id}, \overline{\oplus} \circ \pi_2 \times \mathsf{id} \rangle} \\ = & \quad \{\ \times\text{-CANCEL}, \times\text{-DEF}\ \} \\ & \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ \langle \pi_1 \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id}, \pi_2 \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id} \rangle} \\ = & \quad \{\ \times\text{-FUNCTOR}\times\text{-FUSION}\ \} \\ & \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ \langle \pi_1 \times \mathsf{id}, \pi_2 \times \mathsf{id} \rangle \circ ((\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id})} \\ = & \quad \{\ \wedge\text{-FUSION}, \wedge\text{-ABSOR}\ \} \\ & \mathsf{cons}^{\bullet} \circ \overline{(\mathsf{ap} \times \mathsf{ap}) \circ \langle \pi_1 \times \mathsf{id}, \pi_2 \times \mathsf{id} \rangle} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \\ = & \quad \{\ \mathsf{split}\text{-DEF}\ \} \\ & \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \end{array}$$

We can now show very easily that $k = \mathsf{cons}^\bullet \circ \mathsf{split} \circ \langle \underline{\mathsf{id}}, \mathsf{id} \rangle$.

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

Finally we can apply proposition 4.4 in order to get the desired accumulation (notice that $\overline{\oplus} \circ \underline{\mathsf{nil}} = \underline{\underline{\mathsf{nil}}}$).

$$
\begin{array}{lcl}
\mathsf{isums}_t & : & \mathsf{List\ Int} \to \mathsf{Int} \to \mathsf{List\ Int} \\
\mathsf{isums}_t & = & (\!|[\underline{\underline{\mathsf{nil}}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{plus}} \times \mathsf{cons}^\bullet \circ \mathsf{split} \circ \langle \underline{\mathsf{id}}, \mathsf{id} \rangle)]|)
\end{array}
$$

After converting this definition to pointwise Haskell we get the following implementation. Although this is not a tail-recursive function, it runs in linear time rather than quadratic time as was the case for the initial specification.

```
isums :: [Int] -> [Int]
isums l = isums_t l 0

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

4.4. Accumulations over Leaf-labelled Trees

We now turn to a different inductive type, that of leaf-labeled binary trees. In general, folds over this type (functions whose result on a node is a function of the results on both left and right sub-trees) cannot be made fully tail-recursive; however one of the two recursive invocations can in certain circumstances be tail-recursive, if an accumulator is used. The current value of the accumulator is passed unchanged to one of the recursive calls, and the result of this call is then used as the new accumulator value for the second call. The next proposition shows how this pattern of computation can be calculated in the point-free calculus, and introduces a shortcut rule for it.

Proposition 4.5. Given a left-strict associative operator $\oplus : B \times B \to B$ with right identity e, a function $f : A \to B$, and two functions defined on leaf trees as

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
h & : & \mathsf{LTree}\ A \to B & h_t & : & \mathsf{LTree}\ A \to B \to B \\
h & = & (\!|[f, \oplus]|) & h_t & = & (\!|[\overline{\oplus} \circ f, \mathsf{comp}]|)
\end{array}
$$

Then $h\ t = h_t\ t\ e$.

Proof:
Direct consequence of proposition 4.1, with F instantiated to the base functor of leaf trees ($F\ \overline{\oplus} = \mathsf{id} + \overline{\oplus} \times \overline{\oplus}$), and the following calculation.

$$\begin{array}{ll} & \overline{\oplus} \circ [f, \oplus] \\ = & \quad \{ +\text{-FUSION}, \oplus \text{ left-strict}, \wedge\text{-STRICT} \} \\ & [\overline{\oplus} \circ f, \overline{\oplus} \circ \oplus] \\ = & \quad \{ \oplus\text{-ASSOC} \} \\ & [\overline{\oplus} \circ f, \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus})] \\ = & \quad \{ +\text{-ABSOR} \} \\ & [\overline{\oplus} \circ f, \mathsf{comp}] \circ (\mathsf{id} + \overline{\oplus} \times \overline{\oplus}) \end{array}$$

□

Example 4.6. (Leaves)
Proposition 4.5 can be used to optimize the definition of the $\mathcal{O}(n^2)$ left-to-right traversal function.

```
leaves :: LTree a -> [a]
leaves (Leaf x)     = [x]
leaves (Branch l r) = (leaves l) ++ (leaves r)
```

This function can be defined in the point-free style as follows.

$$\begin{array}{lcl} \mathsf{leaves} & : & \mathsf{LTree}\ A \to \mathsf{List}\ A \\ \mathsf{leaves} & = & (\!|[\mathsf{wrap}, \mathsf{cat}]|\!) \end{array}$$

Considering that $\overline{\mathsf{cat}} \circ \mathsf{wrap} = \overline{\mathsf{cons}}$ we obtain the following faster $\mathcal{O}(n)$ version with accumulations.

$$\begin{array}{lcl} \mathsf{leaves}_t & : & \mathsf{LTree}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{leaves}_t & = & (\!|[\overline{\mathsf{cons}}, \mathsf{comp}]|\!) \end{array}$$

The implementation of this optimized version with explicit recursion is

```
leaves :: LTree a -> [a]
leaves t = leaves_t t []

leaves_t :: LTree a -> [a] -> [a]
leaves_t (Leaf x) m     = x:m
leaves_t (Branch l r) m = leaves_t l (leaves_t r m)
```

Remark. This example has been presented by Bird and de Moor [7] using exactly the same point-free definitions. However, their calculation uses a pointwise definition of comp, which prevents them from reasoning about associativity in the point-free style.

4.5. Accumulations over Rose Trees

A different variety of binary trees is that in which the nodes, rather than the leaves, are labeled. Rather than considering that type here, we turn to a node-labeled type that allows for a variable branch factor. This type can be generated by the following regular functor, defined using the type functor of lists.

$$\begin{array}{rcl} \mathsf{Rose}\ A & = & \mu(\underline{A} \mathbin{\hat{\times}} \mathsf{List}) \\ \mathsf{node} & : & A \times \mathsf{List}(\mathsf{Rose}\ A) \to \mathsf{Rose}\ A \end{array}$$

This data type has a single constructor and can represent non-empty trees only. It can be implemented in Haskell as `data Rose a = Node a [Rose a]`. Notice that folds over this type are functions that combine the contents of each node with a list of recursive results.

Proposition 4.6. Given a left-strict associative operator $\oplus : B \times B \to B$ with right identity e, an element $c : B$, a function $f : A \to B$, and two functions defined on rose trees as

$$\begin{array}{rcl} h & : & \mathsf{Rose}\ A \to B \\ h & = & (\!|\oplus \circ \mathsf{swap} \circ (f \times (\!|[\underline{c}, \oplus]|\!))|\!) \end{array} \qquad \begin{array}{rcl} h_t & : & \mathsf{Rose}\ A \to B \to B \\ h_t & = & (\!|\mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp}]|\!))|\!) \end{array}$$

Then $h\ t = h_t\ t\ e$. Notice that, in both functions, the outer catamorphism traverses a rose tree (functor $\underline{A} \mathbin{\hat{\times}} \mathsf{List}$), while the inner one traverses a list (functor $\underline{1} \mathbin{\hat{+}} \underline{A} \mathbin{\hat{\times}} \mathsf{Id}$).

Proof:
Direct consequence of proposition 4.1 and the following calculation, with F instantiated to the base functor of rose trees ($F\ \overline{\oplus} = \mathsf{id} \times \mathsf{map}_{\mathsf{List}}\overline{\oplus}$).

$$\begin{array}{ll} & \overline{\oplus} \circ \oplus \circ \mathsf{swap} \circ (f \times (\!|[\underline{c}, \oplus]|\!)) \\ = & \quad \{\ \oplus\text{-ASSOC}\ \} \\ & \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \circ \mathsf{swap} \circ (f \times (\!|[\underline{c}, \oplus]|\!)) \\ = & \quad \{\ \mathsf{swap}\text{-NAT}, \times\text{-FUNCTOR}\ \} \\ & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \overline{\oplus} \circ (\!|[\underline{c}, \oplus]|\!)) \\ = & \quad \{\ \text{cata-FUSION}, \oplus \text{ left-strict}, \wedge\text{-STRICT}\ \} \\ & \left[\begin{array}{ll} & \overline{\oplus} \circ [\underline{c}, \oplus] \\ = & \quad \{\ +\text{-FUSION}, \oplus \text{ left-strict}, \wedge\text{-STRICT}\ \} \\ & [\overline{\oplus} \circ \underline{c}, \overline{\oplus} \circ \oplus] \\ = & \quad \{\ \oplus\text{-ASSOC}\ \} \\ & [\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus})] \\ = & \quad \{\ \times\text{-FUNCTOR}, +\text{-ABSOR}\ \} \\ & [\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ (\overline{\oplus} \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \overline{\oplus}) \end{array}\right. \\ & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp} \circ (\overline{\oplus} \times \mathsf{id})]|\!)) \\ = & \quad \{\ +\text{-ABSOR}\ \} \\ & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp}] \circ (\mathsf{id} + \overline{\oplus} \times \mathsf{id})|\!)) \\ = & \quad \{\ \text{fold-map-FUSION}\ \} \\ & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp}]|\!) \circ \mathsf{map}_{\mathsf{List}}\overline{\oplus}) \\ = & \quad \{\ \times\text{-FUNCTOR}\ \} \\ & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times (\!|[\overline{\oplus} \circ \underline{c}, \mathsf{comp}]|\!)) \circ (\mathsf{id} \times \mathsf{map}_{\mathsf{List}}\overline{\oplus}) \end{array}$$

□

Example 4.7. (Postorder)
Consider the following Haskell function that performs a postorder traversal of a rose tree.

```
post :: Rose a -> [a]
post (Node x l) = (aux l) ++ [x]
    where aux []     = []
          aux (x:xs) = (post x) ++ (aux xs)
```

This function can be expressed in point-free style as

$$\begin{array}{lcl} \mathsf{post} & : & \mathsf{Rose}\ A \to \mathsf{List}\ A \\ \mathsf{post} & = & (\!|\mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times (\!|[\underline{\mathsf{nil}}, \mathsf{cat}]|\!))|\!) \end{array}$$

where a catamorphism is used to process the list of results of recursive calls. Proposition 4.6 then allows us to transform this into the following linear time accumulation (we also use some obvious facts about cat to further simplify the result).

$$\begin{array}{lcl} \mathsf{post}_t & : & \mathsf{Rose}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{post}_t & = & (\!|\mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times (\!|[\underline{\mathsf{id}}, \mathsf{comp}]|\!))|\!) \end{array}$$

The optimized version can be implemented in Haskell as follows.

```
post :: Rose a -> [a]
post r = post_t r []

post_t :: Rose a -> [a] -> [a]
post_t (Node x l) m = aux l (x:m)
    where aux [] m     = m
          aux (x:xs) m = post_t x (aux xs m)
```

5. Functions with more than one accumulator

We now get back to the accumulation pattern seen in section 4.4 for leaf-trees. Recall that the value of the accumulator received when processing a node was passed directly to one of the recursive calls, and the result of this call used as the accumulator value for the second call. This is not however the only possibility; certain functions require that the value of the accumulator at the root be received also by the second recursive call. In this situation two accumulators have to be used.

This section presents a concrete example of such a derivation, for the function that determines the height of a binary tree. The example requires the introduction of another point-free operator in the calculus; this is an alternative exponentiation operator that implements a post composition: in addition to $f^A\ g = f \circ g$ we will also define ${}^Af\ g = g \circ f$.

Given $f : B \to C$, this combinator has the following point-free definition.

$$\begin{array}{lcl} {}^Af & : & A^C \to A^B \\ {}^Af & = & \overline{\mathsf{ap} \circ (\mathsf{id} \times f)} \end{array} \qquad \text{pexp-DEF}$$

Likewise to the normal exponentiation we will use $\bullet$ for the superscript when the information about the type is not relevant.

Example 5.1. (Height)
We begin with the following straightforward implementation.

```
height :: LTree a -> Int
height (Leaf x)     = 0
height (Branch l r) = 1 + max (height l) (height r)
```

This can be written as the catamorphism

$$\begin{array}{lcl} \mathsf{height} & : & \mathsf{LTree}\ A \to \mathsf{Int} \\ \mathsf{height} & = & (\!|[\underline{\mathsf{zero}}, \mathsf{succ} \circ \mathsf{max}]|\!) \end{array}$$

The specification of h_t uses two accumulators: the first, d, will store the depth of the current node while traversing the tree; the second, m, will store the maximum depth so far. The specification for fusion is thus, in pointwise and point-free respectively,

$$\mathsf{height}_t\ t\ d\ m = \mathsf{max}\ (\mathsf{plus}\ (\mathsf{height}\ t, d), m)$$
$$\mathsf{height}_t = \overline{\mathsf{max}}^{\bullet} \circ \overline{\mathsf{plus}} \circ \mathsf{height}$$

This specification allows us to apply fusion in two steps: first we fuse height with $\overline{\mathsf{plus}}$ to introduce the first accumulating parameter, and then we fuse the result with $\overline{\mathsf{max}}^{\bullet}$ for the second. For the first calculation the following properties about plus, max, and succ need to be expressed in point-free style.

$$\mathsf{plus}\ (\mathsf{max}\ (x, y), z) = \mathsf{max}\ (\mathsf{plus}\ (x, z), \mathsf{plus}\ (y, z))$$
$$\mathsf{plus}\ (\mathsf{succ}\ x, y) = \mathsf{plus}\ (x, \mathsf{succ}\ y)$$

The first is similar to the one that motivated the introduction of the split combinator in example 4.3; the second can be written using the new exponentiation combinator – a simple proof allows us to obtain this as a consequence of the straightforward $\mathsf{plus} \circ (\mathsf{succ} \times \mathsf{id}) = \mathsf{plus} \circ (\mathsf{id} \times \mathsf{succ})$.

$$\overline{\mathsf{plus}} \circ \mathsf{max} = \mathsf{max}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\mathsf{plus}}) \qquad \text{plus-MAX}$$
$$\overline{\mathsf{plus}} \circ \mathsf{succ} = {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{plus}} \qquad \text{plus-SUCC}$$

For the fusion with $\overline{\mathsf{plus}}$ we proceed with the following calculation. Notice that this operator is strict due to $\wedge$-STRICT and the left strictness of plus.

$$\begin{array}{ll} & \overline{\mathsf{plus}} \circ [\underline{\mathsf{zero}}, \mathsf{succ} \circ \mathsf{max}] \\ = & \quad \{ +\text{-FUSION}, \overline{\mathsf{plus}}\ \text{strict} \} \\ & [\overline{\mathsf{plus}} \circ \underline{\mathsf{zero}}, \overline{\mathsf{plus}} \circ \mathsf{succ} \circ \mathsf{max}] \\ = & \quad \{ \mathsf{zero}\ \text{is a left-identity of plus, plus-SUCC} \} \\ & [\underline{\mathsf{id}}, {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{plus}} \circ \mathsf{max}] \\ = & \quad \{ \text{plus-MAX} \} \\ & [\underline{\mathsf{id}}, {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\mathsf{plus}})] \\ = & \quad \{ +\text{-ABSOR} \} \\ & [\underline{\mathsf{id}}, {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split}] \circ (\mathsf{id} + \overline{\mathsf{plus}} \times \overline{\mathsf{plus}}) \end{array}$$

The result of the first fusion is then $\mathsf{height}_t = \overline{\mathsf{max}}^\bullet \circ (\!|[\underline{\mathsf{id}},\ ^\bullet\mathsf{succ} \circ \mathsf{max}^\bullet \circ \mathsf{split}]|\!)$. The next calculation uses the following laws about the exponentiation combinators and split. See [9] for proofs.

$$f^\bullet \circ \underline{g} = \underline{f \circ g} \qquad \text{const-EXP}$$
$$f^\bullet \circ {}^\bullet g = {}^\bullet g \circ f^\bullet \qquad \text{pexp-EXP}$$
$$\mathsf{split} \circ (f^\bullet \times g^\bullet) = (f \times g)^\bullet \circ \mathsf{split} \qquad \text{split-EXP}$$

Notice that $\overline{\mathsf{max}}^\bullet$ is strict due to the left-strictness of max, $\wedge$-STRICT, and the definition of the exponentiation operator.

$$\begin{array}{ll} & \overline{\mathsf{max}}^\bullet \circ [\underline{\mathsf{id}},\ ^\bullet\mathsf{succ} \circ \mathsf{max}^\bullet \circ \mathsf{split}] \\ = & \quad \{ +\text{-FUSION}, \overline{\mathsf{max}}^\bullet \text{ strict} \} \\ & [\overline{\mathsf{max}}^\bullet \circ \underline{\mathsf{id}}, \overline{\mathsf{max}}^\bullet \circ {}^\bullet\mathsf{succ} \circ \mathsf{max}^\bullet \circ \mathsf{split}] \\ = & \quad \{ \text{const-EXP, pexp-EXP} \} \\ & [\underline{\overline{\mathsf{max}}},\ ^\bullet\mathsf{succ} \circ \overline{\mathsf{max}}^\bullet \circ \mathsf{max}^\bullet \circ \mathsf{split}] \\ = & \quad \{ \wedge\text{-FUNCTOR, max-ASSOC} \} \\ & [\underline{\overline{\mathsf{max}}},\ ^\bullet\mathsf{succ} \circ (\mathsf{comp} \circ (\overline{\mathsf{max}} \times \overline{\mathsf{max}}))^\bullet \circ \mathsf{split}] \\ = & \quad \{ \wedge\text{-FUNCTOR, split-EXP} \} \\ & [\underline{\overline{\mathsf{max}}},\ ^\bullet\mathsf{succ} \circ \mathsf{comp}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{max}}^\bullet \times \overline{\mathsf{max}}^\bullet)] \\ = & \quad \{ +\text{-ABSOR} \} \\ & [\underline{\overline{\mathsf{max}}},\ ^\bullet\mathsf{succ} \circ \mathsf{comp}^\bullet \circ \mathsf{split}] \circ (\mathsf{id} + \overline{\mathsf{max}}^\bullet \times \overline{\mathsf{max}}^\bullet) \end{array}$$

This calculation yields an accumulation defined as follows.

$$\begin{array}{lcl} \mathsf{height}_t & : & \mathsf{LTree}\ A \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ \mathsf{height}_t & = & (\!|[\underline{\overline{\mathsf{max}}},\ ^\bullet\mathsf{succ} \circ \mathsf{comp}^\bullet \circ \mathsf{split}]|\!) \end{array}$$

After expanding the definitions of the combinators we get the following implementation, where one of the recursive calls has been made tail-recursive.

```
height :: LTree a -> Int
height t = height_t t 0 0

height_t :: LTree a -> Int -> Int -> Int
height_t (Leaf x) d m     = max d m
height_t (Branch l r) d m = height_t l (d+1) (height_t r (d+1) m)
```

Remark. The notion of post composition already appeared in [13] as a mean to express some properties about higher-order functions. Similarly to comp in [7], it was defined in pointwise style.

6. Transforming Hylomorphisms into Accumulations

The goal of this section is to show that the application of the techniques presented in this paper is not restricted to catamorphisms. In fact, almost every recursive definition can be expressed as a *hylomorphism* [25], a recursion pattern that corresponds to the composition of a catamorphism with an instance of another recursion pattern we have not yet mentioned: *anamorphisms*.

Anamorphisms. Although already known for a long time, anamorphisms are still not very popular among programmers [16]. They correspond to the dual construction of catamorphisms, in the sense that they encode the simplest way of producing values of a recursive type. Given a function of type $g : B \to F\ B$, the anamorphism generated by g (denoted $[\!(g)\!]$) is defined as follows.

$$\begin{array}{lcl} [\!(g)\!]_F & : & B \to \mu F \\ [\!(g)\!]_F & = & \mu(\lambda f \cdot \mathsf{in}_F \circ F f \circ g) \end{array} \qquad \text{ana-DEF}$$

Function g (sometimes called the *gene* of the pattern) is used to control the generation of values of type μF. As was the case for catamorphisms, a standard Haskell function (called `unfoldr`) exists that encodes this recursion pattern for the particular case of lists.

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of Nothing    -> []
                          Just (a,b) -> a : unfoldr f b
```

Notice that the type of the first parameter is isomorphic to the expected $B \to 1 + A \times B$, since the `Maybe` data type is declared in the standard Haskell libraries as `data Maybe a = Nothing | Just a`. When g returns `Nothing` the generation of the list stops; otherwise g returns a pair with a value for the head of the list and a seed used to recursively generate the tail of the list.

As an example of an anamorphism we give the following function that produces the list of all values from a given input n down to 1. $\mathsf{pred} : \mathsf{Int} \to \mathsf{Int}$ and $\mathsf{iszero} : \mathsf{Int} \to \mathsf{Bool}$ are, respectively, the predecessor and test for zero functions.

$$\begin{array}{lcl} \mathsf{from} & : & \mathsf{Int} \to \mathsf{List\ Int} \\ \mathsf{from} & = & [\!(\mathsf{id} + \langle \mathsf{id}, \mathsf{pred} \rangle) \circ \mathsf{iszero?}]\!] \end{array}$$

Hylomorphisms. By constructing an intermediate data-structure with an anamorphism and then processing it with a catamorphism, more complex forms of recursion can be captured. The following equivalence ensures that a single recursive definition results from such a composition.

$$(\!|g|\!)_F \circ [\!(h)\!]_F = \mu(\lambda f \cdot g \circ F f \circ h) \qquad \text{hylo-FUSION}$$

More precisely, this law states that the catamorphism and the anamorphism can be fused together in a single definition, avoiding the construction of the intermediate data structure. We remark that it is possible to automatically derive pointwise hylomorphisms from almost any explicitly recursive definition [18].

From the above law it is clear that the computations after and before recursion are performed, respectively, by the parameters of the catamorphism and the anamorphism. A consequence of this fact is that if, by using an accumulation parameter, the catamorphism can be transformed into a tail recursive definition, the resulting hylomorphism will also necessarily be tail recursive. The next example shows how to transform a function that does not fit the simple fold recursion pattern.

Example 6.1. (Factorial)
To exemplify the application of the accumulation technique to a function that cannot be directly defined as a catamorphism, consider the standard definition of factorial.

```
fact :: Int -> Int
fact 0     = 1
fact (n+1) = (fact n) * (n+1)
```

This function can be expressed as the hylomorphism

$$\begin{array}{lcl} \mathsf{fact} & : & \mathsf{Int} \to \mathsf{Int} \\ \mathsf{fact} & = & \mathsf{product} \circ \mathsf{from} \end{array}$$

where product is the catamorphism of example 4.2, which can be transformed into a tail recursive function $\mathsf{product}_t$ by introducing an accumulating parameter. By defining $\mathsf{fact}_t = \mathsf{product}_t \circ \mathsf{from}$, then applying hylo-FUSION and unfolding the fixpoint operator, we get the following equation.

$$\mathsf{fact}_t = [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{mult}} \times \mathsf{id})] \circ (\mathsf{id} + \mathsf{id} \times \mathsf{fact}_t) \circ (\mathsf{id} + \langle \mathsf{id}, \mathsf{pred} \rangle) \circ \mathsf{iszero}?$$

This can be simplified to $\mathsf{fact}_t = [\underline{\mathsf{id}}, \mathsf{comp} \circ \mathsf{swap} \circ \langle \overline{\mathsf{mult}}, \mathsf{fact}_t \circ \mathsf{pred} \rangle] \circ \mathsf{iszero}?$. The factorial function can then be rewritten as $\mathsf{fact}\ n = \mathsf{fact}_t\ n\ \mathsf{one}$, bearing in mind that one is the unit of mult . In pointwise Haskell, this corresponds, as expected, to the following tail-recursive implementation.

```
fact :: Int -> Int
fact n = fact_t n 1

fact_t :: Int -> Int -> Int
fact_t 0 y = y
fact_t n y = fact_t (n-1) (n * y)
```

7. Related Work

There have been some attempts to develop automatic systems for program transformation using shortcut fusion in pointwise style. One of the most successful is the MAG system developed by Sittampalam and de Moor [26, 31]. This system is not fully automatic, but relies on the notion of *active source*, that is, the original (inefficient) definitions are stored together with sufficient hints (namely, the specification that results from the generalization strategy, and the creative steps of the derivation) that enable the system to derive the efficient version. It has been applied to perform transformations using both the accumulation and tupling strategies.

At the core of this system lies a term rewriting mechanism that, given a set of transformation rules, tries to apply them from left to right in the order in which they appear in the active source, repeating this process until no rule can be applied. The use of a shortcut fusion rule instead of fold/unfold transformations makes it unnecessary to apply equations in more than one direction. In order to cope with the side-conditions of the fusion rule, this mechanism includes a higher-order matching algorithm that is capable of deriving mechanically new function definitions (like the h function in foldr-FUSION). MAG does not require the original functions to be defined directly as catamorphisms, since such definitions can be derived automatically. The technique that allows to do this was introduced in [22], and basically consists in trying to fuse the original definition with the identity fold (defined as $\mathsf{foldr}\ \overline{\mathsf{cons}}\ \mathsf{nil}$ for lists).

Among the drawbacks of this system we have the fact that it does not include a strictness analyzer, leaving to the user the manual verification of part of the side-conditions of the fusion rule. Another drawback is that the rewriting system is quite limited; for a transformation to succeed, the user must be careful about the order in which the transformation rules are stated in the active source. The same foresight applies to the definitions themselves. As the authors say [31]:

> *... to use MAG to mechanize a fusion derivation, one must first have some idea of what the derivation will be – what MAG does for the programmer is to deal with the details of the derivation, and to make it repeatable without needing to store it with the program.*

Hu, Iwasaki, and Takeichi have used a calculational approach to several program transformation techniques, including deforestation [18], tupling [20], and accumulations [19]. In this latter work, the authors present a methodology for deriving accumulations using fusion, where the expected structure of the parameters of catamorphisms is used to facilitate the derivations. Unlike in the MAG system, the generic definition of catamorphisms (and fusion) is used, but most of the expressions are still defined in the pointwise style. Although the authors suggest that their method is amenable to automation, they present no hints on how to do it. Due to the generality of the transformation laws and the use of pointwise definitions, it is likely that it would also require some form of higher-order matching.

Meijer, Fokkinga, and Paterson [25] have introduced a transformation rule quite similar to proposition 4.2, for deriving accumulations from functions defined over lists. Besides dealing with associative operators, it also covers operators with associative duals. This generality complicates the formalization of the rule by not making the associativity properties explicit, and introducing side-conditions whose mechanical verification is not trivial: to apply the rule one needs to discover new operators, which again implies the use of a higher-order matching algorithm. The rule is expressed in a mixed style that includes both point-free and pointwise definitions: the former is used for writing the catamorphisms, and the latter for defining the (associativity-like) properties of the operators.

The work of Sheard and Fegaras on the derivation of accumulations [30] also bears some similarities to ours (even though no fusion or point-free style are used). A syntactic transformation algorithm is defined for recognizing folds that are amenable to be implemented as accumulations, and automatically converts them into the higher-order folds that define them. For the particular case of lists, the transformation is similar to the one defined by proposition 4.2, with the occurrences of the associative operator being replaced by composition. The authors also acknowledge similarities between this transformation and the classic continuation-passing style transformation. The main advantage of this approach is that the transformation algorithm can be generically applied to folds over any data-type, as long as the involved operator is associative. As such it also covers for instance, proposition 4.5 for transformation of functions over leaf trees.

Accumulations are usually defined as higher-order catamorphisms. However, other approaches have been proposed using different recursion patterns. For instance, it is known that some accumulations can be expressed as regular first order anamorphisms. That is the case of the so-called *downwards accumulations*, functions that label each node of a data structure by applying a function to its ancestors (i.e. information flows in a top-down fashion). Malcolm used anamorphisms to define this kind of accumulations for infinite lists [23], and later, Gibbons presented a generic definition that works for any regular data type [14]. Given a binary operator $\oplus : B \times A \to A$, we could slightly change Malcolm's

definition to work on finite lists as follows.

$$\begin{array}{lcl} \mathsf{da}\ \oplus & : & \mathsf{List}\ B \times A \to \mathsf{List}\ A \\ \mathsf{da}\ \oplus & = & [\!(\pi_1 + \langle \pi_2, (\mathsf{id} \times \oplus) \circ \mathsf{assocr} \circ (\mathsf{swap} \times \mathsf{id}) \rangle) \circ \mathsf{distl} \circ (\mathsf{out} \times \mathsf{id})]\!] \end{array}$$

This function can be implemented in Haskell as follows (notice that it is very similar to the standard Haskell function `scanl`).

```
da :: ((b, a) -> a) -> ([b], a) -> [a]
da op ([], b)   = []
da op (x:xs, b) = b:(da op (xs, op (x, b)))
```

With da we can define the function that computes the initial (reversed) segments of a list as follows.

$$\begin{array}{lcl} \mathsf{inits} & : & \mathsf{List}\ A \to \mathsf{List}\ \mathsf{List}\ A \\ \mathsf{inits} & = & \mathsf{da}\ \mathsf{cons} \circ \langle \mathsf{id}, \underline{\mathsf{nil}} \rangle \end{array}$$

Using these definitions, Malcolm proves the following shortcut fusion rule, that can be used to improve the efficiency of a function of type List $B \to$ List A. The operator $\oplus$ has type $B \times A \to A$ and $e : A$. Gibbons also generalized this rule to work with any regular data type.

$$(\mathsf{map}_{\mathsf{List}}(\!|[\underline{e}, \oplus]|\!) \circ \mathsf{inits})\ l = \mathsf{da} \oplus (l, e)$$

Notice that this rule does not have side conditions (not even the associativity of $\oplus$ is required). The left side of the equation can be seen as a clear specification of downwards accumulations for the particular case of lists, while the right side is the expected efficient implementation. As an example, this rule can be used to transform the following (very easy to understand) specification of a function very similar to isums of example 4.5

$$\begin{array}{lcl} \mathsf{isums}' & : & \mathsf{List\ Int} \to \mathsf{List\ Int} \\ \mathsf{isums}' & = & \mathsf{map}_{\mathsf{List}}\mathsf{sum} \circ \mathsf{inits} \end{array}$$

into the efficient implementation $\mathsf{isums}'\ l = \mathsf{da\ plus}\ (l, \mathsf{zero})$.

Without resorting to higher-order, it is still possible to express a wide range of accumulations using the hylomorphism recursion pattern. Pardo followed this approach in order to define a generic accumulation operator that supersedes the one proposed by Gibbons [27]. Although no further specialization was carried out, he defines a general transformation rule similar to proposition 4.1 using the new operator. Although powerful, this definition of accumulations is not as expressive as the one used in this paper – for example, it is not possible to define the accumulation of example 4.6.

There is some research work in program transformation with accumulations that is not concerned with deriving accumulations from recursive definitions, but rather with studying fusion of functions already defined as accumulations [19, 14, 27, 32]. This work aims at developing the calculus in order to prove facts like $\mathsf{product}\ (\mathsf{reverse}_t\ l\ \mathsf{nil}) = \mathsf{product}_t\ l\ \mathsf{one}$.

8. Conclusions

In this paper we have shown how the classic accumulation strategy can be applied using calculation in a pure point-free style. We have briefly compared this approach with the standard fold/unfold transformations, and pointwise calculation. The main similarity between all these techniques is the need for a creative step for writing the initial specification that will be transformed (the generalization step of fold/unfold transformations). Our emphasis was on finding generic transformation schemes for various data types, that can be used as shortcut optimization rules in an automatic transformation system. We have also presented a point-free derivation of a function with two accumulating parameters, emphasizing the modularity of the calculational approach – the accumulating arguments were introduced in separate, simpler fusion steps. Although we have focused on a specific transformation strategy, it is our belief that exactly the same approach can be followed for other transformation techniques, such as tupling or deforestation.

In order to cope with calculations in a higher-order setting, we have felt the need to internalize uncurried versions of some of the basic combinators as point-free definitions. This was the case for the composition and split combinators. Fundamental properties, like the associativity of curried operators, can be succinctly expressed using these definitions, leading to major simplifications in the calculations. We have also introduced a new point-free exponentiation operator, equivalent to the right-sectioning of the composition combinator.

Other contributions of the paper include the generalization of the approach to a broader class of recursive definitions (by using hylomorphisms), and a clarification of the strictness side-conditions that characterize the accumulation strategy in the $\mathbf{Cpo}$ domain. As was shown, strictness analysis can also be made by calculation – the strictness side-conditions were derived from a basic set of laws concerning the strictness properties of the basic combinators and recursion patterns.

There are some limitations in our methodology: first, it is still not clear how to automatically derive point-free expressions from the typical pointwise definitions most programmers use; second, as shown in example 4.3, it is sometimes necessary to (non-trivially) change the initial definition of a program to enable the application of the transformation rules.

In the near future we intend to develop a transformation system for point-free programs, based on a term-rewriting approach. The idea is to take advantage of the pure point-free style to circumvent the need for a higher-order matching algorithm for finding the “unknowns” in the transformation rules. This means that we can follow a simpler approach, as in Bird’s functional calculator [6]. With this system we hope to provide practical evidence of the advantages of using a point-free style for program transformation by calculation.

Acknowledgements

The authors would like to thank Simão Melo de Sousa, Joost Visser, and the reviewers for their comments. This research was partially supported by FCT research project Pure POSI/CHS/44304/2002.

References

[1] Backhouse, R.: Making Formality Work For Us, *EATCS Bulletin*, **38**, 1989, 219–249.

[2] Backus, J.: Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, **21**(8), 1978, 613–641.

[3] Bauer, F. L., Wössner, H.: *Algorithmic Language and Program Development*, Springer-Verlag, 1982.

[4] Bird, R.: The Promotion and Accumulation Strategies in Transformational Programming, *ACM Transactions on Programming Languages and Systems*, **6**(4), October 1984, 487–504.

[5] Bird, R.: An Introduction to the Theory of Lists, in: *Logic of Programming and Calculi of Discrete Design* (M. Broy, Ed.), Springer-Verlag, 1987, 3–42.

[6] Bird, R.: *Introduction to Functional Programming using Haskell*, International Series in Computer Science, Prentice Hall, 1998.

[7] Bird, R., de Moor, O.: *Algebra of Programming*, Prentice Hall, 1997.

[8] Burstall, R. M., Darlington, J.: A Transformation System for Developing Recursive Programs, *Journal of the ACM*, **24**(1), January 1977, 44–76.

[9] Cunha, A., Pinto, J. S.: *Point-free Program Transformation*, Technical Report DI-PURe-04:02:03, Departamento de Informática, Universidade do Minho, February 2004, Available from `http://www.di.uminho.pt/pure`.

[10] Darlington, J.: An Experimental Program Transformation System, *Artificial Intelligence*, **16**, 1981, 1–46.

[11] Fokkinga, M., Meijer, E.: *Program Calculation Properties of Continuous Algebras*, Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[12] Gibbons, J.: An Introduction to the Bird-Meertens Formalism, *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, November 1994.

[13] Gibbons, J.: A Pointless Derivation of Radix Sort, *Journal of Functional Programming*, **9**(3), 1999, 339–346.

[14] Gibbons, J.: Generic Downwards Accumulations, *Science of Computer Programming*, **37**(1–3), 2000, 37–65.

[15] Gibbons, J.: Calculating Functional Programs, in: *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction* (R. Backhouse, R. Crole, J. Gibbons, Eds.), vol. 2297 of *LNCS*, chapter 5, Springer-Verlag, 2002, 148–203.

[16] Gibbons, J., Jones, G.: The Under-Appreciated Unfold, *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, ACM Press, 1998.

[17] Hagino, T.: *A Categorical Programming Language*, Ph.D. Thesis, University of Edinburgh, 1987.

[18] Hu, Z., Iwasaki, H., Takeichi, M.: Deriving Structural Hylomorphisms From Recursive Definitions, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, ACM Press, 1996.

[19] Hu, Z., Iwasaki, H., Takeichi, M.: Calculating Accumulations, *New Generation Computing*, **17**(2), 1999, 153–173.

[20] Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling Calculation Eliminates Multiple Data Traversals, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, ACM Press, 1997.

[21] Jones, S. P., Hughes, J., Eds.: *Haskell 98: A Non-strict, Purely Functional Language*, February 1999.

[22] Launchbury, J., Sheard, T.: Warm Fusion: Deriving Build-Catas from Recursive Definitions, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1995.

[23] Malcolm, G.: Data Structures and Program Transformation, *Science of Computer Programming*, **14**(2–3), October 1990, 255–279.

[24] Meertens, L.: Algorithmics – Towards Programming as a Mathematical Activity, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, North-Holland, 1986.

[25] Meijer, E., Fokkinga, M., Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)* (J. Hughes, Ed.), 523, Springer-Verlag, 1991.

[26] de Moor, O., Sittampalam, G.: Generic Program Transformation, *Proceedings of the 3rd International Summer School on Advanced Functional Programming* (D. Swierstra, P. Henriques, J. Oliveira, Eds.), 1608, Springer-Verlag, 1999.

[27] Pardo, A.: Generic Accumulations, *Proceedings of the 2002 IFIP TC2 Working Conference on Generic Programming* (J. Gibbons, J. Jeuring, Eds.), Kluwer Academic Publishers, Schloss Dagstuhl, Germany, 2003.

[28] Pettorossi, A., Proietti, M.: Rules and Strategies for Transforming Functional and Logic Programs, *ACM Computing Surveys*, **28**(2), 1996, 360–414.

[29] Reynolds, J.: Semantics of the Domain of Flow Diagrams, *Journal of the ACM*, **24**(3), July 1977, 484–503.

[30] Sheard, T., Fegaras, L.: A Fold for All Seasons, *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture*, 1993.

[31] Sittampalam, G., de Moor, O.: Mechanising Fusion, in: *The Fun of Programming* (J. Gibbons, O. de Moor, Eds.), chapter 5, Palgrave Macmillan, 2003, 79–104.

[32] Svenningsson, J.: Shortcut Fusion for Accumulating Parameters & Zip-like Functions, *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, ACM Press, 2002.

[33] Takano, A., Hu, Z., Takeichi, M.: Program Transformation in Calculational Form, *ACM Computing Surveys*, **30**(3), September 1998.

[34] Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees, *Proceedings of the European Symposium on Programming*, number 300 in LNCS, Springer-Verlag, 1988.

A. Some Laws About Products, Coproducts, and Exponentials

$$\langle \pi_1, \pi_2 \rangle = \mathsf{id} \qquad \times\text{-REFLEX}$$

$$\pi_1 \circ \langle f, g \rangle = f \ \wedge\ \pi_2 \circ \langle f, g \rangle = g \qquad \times\text{-CANCEL}$$

$$\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle \qquad \times\text{-FUSION}$$

$$(f \times g) \circ \langle h, i \rangle = \langle f \circ h, g \circ i \rangle \qquad \times\text{-ABSOR}$$

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

$$\langle f, g \rangle = \langle h, i \rangle \quad \Leftrightarrow \quad f = h \ \wedge\ g = i \qquad \times\text{-EQUAL}$$

$$\langle f, g \rangle \text{ strict} \quad \Leftrightarrow \quad f \text{ strict} \ \wedge\ g \text{ strict} \qquad \times\text{-STRICT}$$

$$[i_1, i_2] = \mathsf{id} \qquad +\text{-REFLEX}$$
$$[f, g] \circ i_1 = f \;\wedge\; [f, g] \circ i_2 = g \qquad +\text{-CANCEL}$$
$$f \circ [g, h] = [f \circ g, f \circ h] \quad \Leftarrow \quad f \text{ strict} \qquad +\text{-FUSION}$$
$$[f, g] \circ (h + i) = [f \circ h, g \circ i] \qquad +\text{-ABSOR}$$
$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad +\text{-FUNCTOR}$$
$$[f, g] = [h, i] \quad \Leftrightarrow \quad f = h \;\wedge\; g = i \qquad +\text{-EQUAL}$$
$$\forall f, g \cdot [f, g] \text{ strict} \qquad +\text{-STRICT}$$

$$\overline{\mathsf{ap}} = \mathsf{id} \qquad \wedge\text{-REFLEX}$$
$$f = \mathsf{ap} \circ (\overline{f} \times \mathsf{id}) \qquad \wedge\text{-CANCEL}$$
$$\overline{f \circ (g \times \mathsf{id})} = \overline{f} \circ g \qquad \wedge\text{-FUSION}$$
$$f^A \circ \overline{g} = \overline{f \circ g} \qquad \wedge\text{-ABSOR}$$
$$(f \circ g)^A = f^A \circ g^A \qquad \wedge\text{-FUNCTOR}$$
$$\overline{f} = \overline{g} \quad \Leftrightarrow \quad f = g \qquad \wedge\text{-EQUAL}$$
$$\overline{f} \text{ strict} \quad \Leftrightarrow \quad f \text{ left-strict} \qquad \wedge\text{-STRICT}$$